Contents

RenderWare V1.4 Help File

The following Help Topics are available:

For Help on Help, Press F1

You will be notified when the final release of this help file is available.

Using the Library

This section presents an overview of the main concepts of RenderWare. The following

- Error Handling.
- Debugging.

Related Topics

Overview
Coordinate Systems
Matrices
The Virtual Camera Model
Hierarchical Modeling
The Structure of a RenderWare Program
Error Reporting
Debugging

Overview

A **Clump** is a collection of **Polygons** and **Vertices**. Clump objects allow applications to handle large numbers of related polygons and vertices as a single, atomic entity. This greatly simplifies application construction.

Each polygon has an associated **Material** object, which defines the appearance of the polygon. Each material object may be shared by many polygons. Material objects encapsulate the following surface properties: **geometry sampling** type, **light sampling** type, **RGB color**; **ambient**, **diffuse** and **specular coefficients** of **reflection**, **opacity**, **texture** and **texture modes**.

A [illegible] objects in a scene. Three types of light source are supported:

- [illegible] rections from a specified position. A light bulb is an example of a point light source.
- A **conical light** source emits a cone of light centered about a specified axial direction from a specified position. A spot light is an example of a conical light source.

A **Camera** captures an image of the objects in a 3D scene projected onto an image plane (or view plane). The projection may be either perspective or parallel. A rectangular region of this plane (called the **view window**) is stored as a 2D image in the cameras **image buffer**. The rectangular region of pixels on the output device onto which a cameras view window is mapped is the cameras **viewport**.

A **Scene** is a collection of clumps and lights. A scene may be viewed through one or more cameras. At any instant, however, only one camera is active (the **current camera**) and the results of rendering are stored in the image buffer of this camera.

Coordinate Systems

RenderWare recognizes the following coordinate systems:

- Object Space

Each clump has a local coordinate system (or **Object Space**). This is the coordinate system in which geometry being added to a clump is specified. For example:

```
RwAddVertexToClump(Clump, 1.0, 1.0, 1.0);
```

adds a new vertex to Clump at (1.0, 1.0, 1.0) in the clumps Object Space. When constructing geometry using a script file or the RenderWare API, Object Space coordinates are transformed by the current transform matrix (CTM) before being added to the clump.

- World Space

The coordinate system of a scene is known as the world coordinate system (or **World Space**). This is the coordinate system that is used for specifying the positions of lights and cameras.

RenderWare uses a *right-handed world coordinate system*. This has the same orientation as that given by taking the thumb, first and second fingers of your right hand as the X, Y and Z axes respectively. The positive X axis points to the right, the positive Y axis points up, whilst the positive Z axis points forwards (out of the screen).

- Camera Space

Each camera has a viewing coordinate system (or **Camera Space**). Camera space has its origin at the cameras position. The positive Z axis of camera space is given by the view direction (or Look At vector). Unlike world space, *camera space is left-handed*. The units of camera space are the same as those of world space.

Camera space is primarily used in the API function **RwVCMoveCamera**() which moves the camera a certain number of units left, right, up, down, forwards or backwards relative to the current camera position.

- Device Space

The coordinate system of the display device (or **Device Space**) is a two dimensional coordinate space used when specifying the position of cameras viewports. Device space is closely related to viewport space as described below.

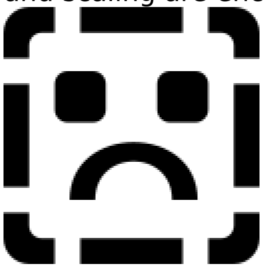
- Viewport Space

The coordinate system of a cameras viewport is the viewport coordinate system (or **Viewport Space**). Viewport space is used when picking, when damaging or undamaging the cameras viewport, when specifying a portion of the viewport into which a backdrop should be copied and when rendering user-draws. The units of viewport space are the same as those of device space. The terms device space units and viewport space units will be used interchangeably throughout this document.

To convert a point in device space to the viewport space of a camera, simply subtract the X and Y coordinates of the cameras viewport from the X and Y coordinates of the point. This conversion is often useful when attempting to pick an object under the mouse pointer. The mouse position will normally be returned to the application in device (screen or window) coordinates. However, viewport coordinates are needed for the pick operation and so the above conversion should be performed. This conversion is not necessary if the viewports origin is the same as the origin of device space (as is often the case).

Matrices

RenderWare uses 4 x 4 homogeneous matrices to represent 3D transformations. Rotation and scaling are encoded in the top-left 3 x 3 sub-matrix, and translation in the final row:



Such matrices are available to the application programmer through the opaque data type `RwMatrix4d`.

In addition to various high-level matrix operations (such as multiplication) the elements of a matrix can be set or retrieved either individually or as a whole. For example, the API function **`RwGetMatrixElements`**`()` copies the elements of a matrix into a 4 x 4 array of real numbers:

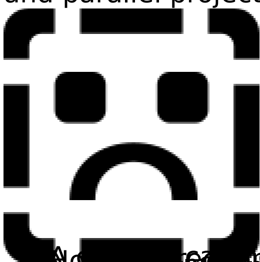
```
RwMatrix4d *matrix;
RwReal elements[4][4];

matrix = RwCreateMatrix();
RwGetMatrixElements(matrix, elements);
```

The first index of the elements array is the row number and the second is the column number. Thus, the X, Y and Z values of the translation component of `matrix` are `elements[3][0], elements[3][1]` and `elements[3][2]` respectively.

The Virtual Camera Model

The purpose of a camera is to project the objects in a 3D scene onto an image plane (or view plane), a rectangular region of which, called the view plane window (or view window), is output as a 2D image. The rectangular portion of the display surface onto which a cameras **view window** is mapped is called the **viewport**. RenderWare supports both perspective and parallel projections.



[illegible] camera can be positioned anywhere in world coordinate space and can be pointed in [illegible]

- Changing the view window is analogous to changing the setting of a zoom lens in a real camera and can simulate the effects of wide angle or telephoto lenses (by specifying larger and smaller extents for the view window respectively). To get objects to appear in their normal size, the view window extent should be set to one. Increasing the size of the view window widens the field of view, which means that a larger proportion of the scene will appear in the image and therefore the objects in the scene appear smaller.
- The user can also change the position of the front (near) and back (far) clipping planes (both of which are co-planar with the view plane) by specifying a distance from the camera position. The near clipping plane must be at a distance greater than `0.025` units from the camera.

Hierarchical Modeling

Hierarchical modeling is the process of building models that preserve the hierarchical structure of objects and allow the position and orientation of an object in the hierarchy to be specified relative to its parent.

Related Topics

Hierarchical Modeling in RenderWare
Constructing Hierarchical Models in Scripts
Traversing Clump Hierarchies
Finding a Particular Clump In a Hierarchy

Hierarchical Modeling in RenderWare

RenderWares hierarchical modeling support provides the ability to explicitly model articulation or joints connecting objects.
A clump may have a parent and zero or more children.
Each clump has its own independent, local coordinate system. A clump has three associated matrices: a modeling transformation matrix, a joint (or articulation) transformation matrix, and a local transformation matrix (LTM)
The modeling and joint transformations of a child clump together specify the mapping from its the local coordinate system to that of the parent clump. The joint transformation specifies the rotation of the child clump about its local origin; clumps are therefore always hinged about their local origin (the joint is always at the origin of the child clump). The modeling transformation specifies where the origin of the child clump is with respect to the origin of the parent clump.
Typically, the geometry being added to a clump is positioned with respect to that clumps origin by applying translation, rotation or scaling to the current transformation matrix.
The local transformation matrix (LTM) of a clump maps from the local coordinate system of the clump to the world coordinate system of the scene to which it belongs. It may be calculated by initialization to the identity matrix followed by an ascent from that clump to the root of its hierarchy with post transformation first by the joint and then the modeling transformation at each level. Note that a clumps local transformation is computed by RenderWare and can be retrieved by `RwGetClumpLTM()`. It cannot (and need not) be set directly by the application programmer.

Constructing Hierarchical Models in Scripts

This section gives a simple example of building a hierarchical model in a RenderWare script (.rwx) file. The example is equally applicable to RenderWares Object Builder API functions.

The following is an example of a script defining a simple hierarchical model forming part of a robots body:

```
ModelBegin
    ClumpBegin
        Color 1.0 0.0 0.0
        Surface 0.5 0.5 0.5

        Block 0.6 0.8 0.3

        Translate 0.3 0.4 0.15
        ClumpBegin
            Color 1.0 1.0 1.0

            Translate 0.1 -0.4 0.0
            Block 0.2 0.8 0.2
        ClumpEnd
    ClumpEnd
ModelEnd
```

The top-level **ClumpBegin** ... **ClumpEnd** block defines a clump representing the central section of a robots body. One arm of the robot is then modeled by a child of this clump.

The Translate keyword applies a translation to the current transformation matrix (CTM). When the definition of the child clump is begun (with **ClumpBegin**), the modeling matrix for the child clump is set to the CTM. The first **Translate** keyword positions the child clump in relation to its parent; the childs local origin is displaced by `[0.3 0.4 0.15]` from that of the parent.

Before adding the second block corresponding to the child clump, a translation is applied to the CTM. This specifies the position of the block relative to the origin of the clump.

The rotation of the arm about the shoulder joint requires only one additional line in the script: **RotateJointTM**. This keyword should be inserted immediately before the **ClumpBegin** ... **ClumpEnd** defining the child clump. The command specifies the direction of the axis of rotation about the joint (in the childs local space) and the angle of rotation in degrees:

```
RotateJointTM 1.0 0.0 0.0 30
```

The above rotates the arm about the X axis of the childs local space by 30 degrees.

Traversing Clump Hierarchies

There are two ways of traversing clump hierarchies:

1. Starting from any clump in a clump hierarchy, every other clump in that hierarchy may be visited with clump access functions such as **`RwGetClumpParent`**`()`, **`RwGetNextClump`**`()` and **`RwGetFirstChildClump`**`()`.
2. All clumps in a hierarchy may be iterated over by the API function **`RwForAllClumpsInHierarchy`**`()` and its variants. This family of functions is convenient in situations when the same operation is to be applied to each clump in a hierarchy.

Finding a Particular Clump In a Hierarchy

There are two main techniques for finding a particular clump:

1 RenderWare allows an integer tag to be attached to each clump. A clumps tag can be set by the API function **RwSetClumpTag**() or the script keyword **Tag** and retrieved by the API function **RwGetClumpTag**(). Tags are convenient for marking parts of a hierarchical model for identification and manipulation by an application program. A tagged clump may be found with **RwFindTaggedClump**().

2 The second, more general technique involves **RwFindClump**() or one of its variants. These functions apply a boolean (predicate) call-back function to each clump in a hierarchy in turn until the call-back function returns non-zero. Iteration is then terminated. The clump passed as the argument to the call-back function is returned.

The Structure of a RenderWare Program

The include file `rwlib.h` contains the prototypes for all RenderWare API functions. Any application program exploiting RenderWare should include this include file.

There are three other important include files: `rwtypes.h`, `rwmacros.h` and `rwerrors.h`. `rwtypes.h` contains the declarations of RenderWares data types; `rwmacros.h` contains macro functions for fixed-point arithmetic and `rwerrors.h` contains the RenderWare error codes. Each of these files is [illegible]d by `rwlib.h`, there is no need to explicitly include them.

Any [illegible] RenderWare [illegible] program must be [illegible] `RwOpen()` and `RwClose()` respectively.

- [illegible]
- creates and initializes a camera
- creates a scene
- creates and initializes one or more lights and adds them to a scene
- creates one or more clumps and add them to a scene.
- repeatedly renders a scene
- closes the library

Each of these tasks is discussed in more detail below. Code fragments are given to illustrate each operation. These code fragments are sections of a floating-point, RenderWare application targeted at the Microsoft Windows 3.1 operating system.

The code fragments are from a program which displays a clump read from a script file.

For the sake of clarity the following code fragments omit the various macros necessary for compatibility with the fixed-point RenderWare libraries. Furthermore, variable declarations and some error checking have been omitted.

Related Topics

Initializing the Library
Creating and Initializing a Camera
Creating a Scene
Creating a Light Source and adding it to a Scene
Creating a Clump and adding it to a Scene
Rendering a Scene
Closing the Library

Initializing the Library

RwOpen`()` initializes the library for a particular output device.

```
if (RwOpen(MSWindows, NULL))
{
          /*
           * Since the return value was non-zero,
           * the application can continue...
           */
```

Creating and Initializing a Camera

A new camera can be created with **`RwCreateCamera`**`()`. This requires the maximum width and height of the cameras viewport and a device-specific parameter as its arguments:

```
cam = RwCreateCamera(320, 200, NULL);
if (cam)
     {
          /*
           * As NULL was not returned the call
           * was successful and the application can
           * continue...
           */
```

The above code fragment creates a camera cam with a maximum viewport size of 320 by 200 pixels.

By default, the camera is positioned at the origin of world space, looking down the world space Z axis in the direction of decreasing Z. The camera has a view window of 1.0 unit by 1.0 unit and has a viewport background color of black (0.0, 0.0, 0.0). The default projection model is perspective.

To move the camera back 10.0 units down the Z axis from its initial position:

```
RwVCMoveCamera(cam, 0.0, 0.0, -10.0);
```

Important Note:
By default, the cameras viewport has a width and height of zero. The application program must explicitly set the viewport of each new camera to a non-zero width and height. This viewport should be set with the API function **RwSetCameraViewport**`()` as soon as the desired size is established.

The following code fragment sets the cameras viewport in response to a `WM_SIZE` message from MS Windows:

```
case WM_SIZE:
     width = LOWORD(lParam);
     height = HIWORD(lParam);
     RwSetCameraViewport(cam, 0, 0, width, height);
```

Creating a Scene

RwOpen`()` creates a scene (the default scene) which holds all clumps and lights when they are first created. The default scene can be viewed and rendered with a camera in the same way as any other scene. However, it is recommended that an application creates a specific scene for rendering and only uses the default scene to hold currently unused clumps and lights.

```
if (scene = RwCreateScene())
{
          /*
           * Since NULL was not returned,
           * the call was successful and the
           * application can continue...
           */
```

Creating a Light Source and adding it to a Scene

A light source is created by calling **`RwCreateLight`**`()`.

```
if (light = RwCreateLight(rwPOINT,
                    0.0, 10.0, 0.0, 1.0))
{
          /*
           * Since NULL was not returned,
           * the call was successful and the
           * application can continue...
           */
```

The above code fragment creates a point light source of maximum brightness `(1.0)` positioned at `(0.0, 10.0, 0.0)` in world space.

To add the light to the scene:

```
RwAddLightToScene(scene, light);
```

Creating a Clump and adding it to a Scene

A clump is read from a RenderWare script file as follows:

```
ball = RwReadShape(ball.rwx);
if (ball)
{
        /*
         * Since NULL was not returned,
         * the call was successful and the
         * application can continue...
         */
```

If the application wishes to replace the surface properties of the clump defined in the script file it can do so via the RenderWare API functions which deal with polygons and materials:

```
RwForAllPolygonsInClumpInt(ball,
     RwSetPolygonGeometrySampling, rwSOLID);

RwForAllPolygonsInClumpInt(ball,
     RwSetPolygonLightSampling, rwVERTEX);

RwForAllPolygonsInClumpReal(ball,
          RwSetPolygonAmbient, 0.7);

color.r = 1.0;
color.g = 0.0;
color.b = 1.0;
RwForAllPolygonsInClumpPointer(ball,
          RwSetPolygonColorStruct, &color);
```

The above code fragment sets the geometry sampling type, light sampling type, ambient reflection coefficient, and color of all the clumps polygons.

To add the clump to the scene:

```
RwAddClumpToScene(scene, ball);
```

Rendering a Scene

At some point, the application must render the scene and display the results of that rendering. Under Windows the application may do this in response to a `WM_PAINT` message:

```
case WM_PAINT:
     hdc = BeginPaint(window, &ps);

     RwBeginCameraUpdate(cam, (void *)window);
          RwClearCameraViewport(cam);
          RwRenderScene(scene);
     RwEndCameraUpdate(cam);
     RwShowCameraImage(cam, (void *)hdc);

     EndPaint(window, &ps);
```

Closing the Library

Finally, when all rendering is complete, the RenderWare library is closed:

```
RwClose();
```

Error Reporting

RenderWare reports errors by setting a global error status. The majority of API functions have a distinguished return value which indicates that the global error status has been set by that function.

The global error status may be interrogated by **RwGetError**(). This returns `E_RW_NOERROR` if no error has occurred. Otherwise, an error code is returned representing the first error encountered since the global error status was last cleared. The global error code is cleared to `E_RW_NOERROR` by calling **RwGetError**().

The global error status is set when an error occurs and the global error status is `E_RW_NOERROR`. The actual error code set indicates the type of the error. Once set, the error status is not set by any subsequent error until **RwGetError**() is called. This ensures that the first error encountered is not over written by subsequent errors

As previously described, the majority of API functions have a distinguished return value which indicates that the global error status has been set by that function. However, there are functions where this is not possible. For example, the function **RwGetClumpParent**() can return `NULL` as a legal value (when the clump is the root of a hierarchy). The documentation for any such function directs the application to call **RwGetError**() to determine whether an error occurred.

It is strongly recommend that application programs test the return values of library functions to check for errors. This is particularly important in the case of RenderWares constructor functions such as **RwCreateMatrix**() or **RwCreateCamera**().

The range of error codes that may be returned by **RwGetError**() is given in Appendix C. RenderWare does not include descriptive strings for each error type. Therefore, to determine the nature of an error it may be necessary to convert a numeric error code into a description. Appendix C contains a table which maps numeric error codes to error identifiers. Furthermore, when using the RenderWare debugging kernel, texture messages for each error generated are issued to the debugging stream. This can be extremely useful in tracking down errors.

Finally, note that application programs may set RenderWares global error status using **RwSetUserError**(). This signals that an error was encountered in an application supplied call back function. Its primary function is to prematurely terminate the iteration performed by one of RenderWares **RwForAll**...() functions.

Debugging

Two forms of the RenderWare library are available. The retail version, with which all final, retail quality systems should be linked and the debugging version which aids in the development of RenderWare applications.

Although slower than the retail version, the debugging version performs more error checking and issues messages to alert the application programmer to potential problems. A set of API functions are provided to control the type of tracing and debugging information generated. In the retail version of the library, these functions are simply null operations. Applications can, therefore, switch between debugging and production libraries by simply re-linking or, for DLL users, by simply switching DLLs.

Debugging messages are associated with three levels of severity:

- Informational — [illegible]
- Warning — A recoverable error was encountered.
- Error — A fatal error was encountered.

The above are listed in ascending order of severity. The items in the list correspond to values (`rwINFORM`, `rwWARNING`, and `rwERROR`) of the enumerated type `RwDebugSeverity`. The messages dispatched to the debugging stream may be filtered according to their severity by setting the minimum severity level. For example, setting the severity level to `rwWARNING` would filter out informational messages whilst retaining warning and error messages.

There are four categories of debugging messages:

- Assertion failure messages.
- Scripting trace messages.
- Miscellaneous messages.
- API function tracing messages.

All messages share the following format:

```
RW <severity level> [xx:yyyy:RwFunctionName] <text of the message>
```

The `<xx:yyyy>` is a code comprising a two- and four-digit number. This code is of no significance to the application programmer, but is informative to RenderWares technical support staff when investigating problems. If the function generating the message is a RenderWare API function then the function name will also appear in the message. All other internal functions will have a reported name of `RwInternal`.

I. Assertion Failure Messages

These messages report the failure of assertions made within the library functions. For example, the **`RwCreateCamera`**`()` function asserts that its maximum width and height arguments should have positive values. If either the width or height is negative the assertion fails and a message is issued.

An assertion failure message takes one of the following two forms:

```
RW INFORM  [xx:yyyy:RwCreateCamera] ASSERT FAILED

RW WARNING [xx:yyyy:RwCreateCamera] ASSERT FAILED
```

Assertion failure messages can have either informational or warning level severity. Assertion failure messages are informative to RenderWares technical support staff but contain no significant information for the application programmer.

Note that the execution of an API function continues after the detection of an assertion failure. The debugging version of the library behaves in the same way as the standard version.

II. Scripting Trace Messages

These messages trace the parsing of a script file. This is useful in finding problems in script files that fail to load.

The format of a scripting trace message is:

```
RW INFORM [xx:yyyy] SCRIPT Line <line number>: Parsing <scripting keyword>
```

For example:

```
RW INFORM [12:3324] SCRIPT Line:25 Parsing ClumpEnd
```

Note that scripting trace messages always have severity level `rwINFORM`.

III. Miscellaneous Messages

Several of the debugging and tracing messages (including all API level errors) fall into this category. Two examples are:

```
RW ERROR  [15:8463:RwSetLightBrightness] E_RW_INV_LIGHT: Invalid light type
passed to library function
```

```
RW INFORM [92:3425:RwCurrentMatrix] RwCurrentMatrix is obsolete. Please use
RwScratchMatrix.
```

Miscellaneous messages can be of any severity level. API level error messages are always at severity level `rwERROR`.

The above message classes can be enabled or disabled individually or collectively. By default, assertion failure and miscellaneous messages are enabled whilst script trace messages are disabled.

IV. API Function Tracing Messages

These messages trace function calls made to the RenderWare library. A message for each entry into and for each exit from an API function is produced.

Example API function tracing messages are:

```
RW INFORM [92:3425:RwCurrentMatrix] ENTER
```

```
RW INFORM [92:3425:RwCurrentMatrix] EXIT
```

Note that API function tracing messages always have severity level `rwINFORM`.

Some example scenarios for controlling debugging messages are given below:

Scenario 1: Report Everything

```
RwSetDebugSeverity(rwINFORM);
RwSetDebugOutputState(rwENABLE);
```

Scenario 2: Report Only API-level Errors

```
RwSetDebugSeverity(rwERROR);
RwSetDebugOutputState(rwENABLE);
```

Scenario 3: Report Only Script Tracing Information

```
RwSetDebugSeverity(rwINFORM);
RwSetDebugAssertionState(rwDISABLE);
RwSetDebugMessageState(rwDISABLE);
RwSetDebugTraceState(rwENABLE);
```

Scenario 4: Report All Warnings and Errors

```
RwSetDebugSeverity(rwWARNING);
RwSetDebugOutputState(rwENABLE);
```

This code is only valid when using a floating-point RenderWare library. The macro functions necessary for correct fixed-point operation have been omitted for clarity.

RwCreateMatrix() can fail and so its return value should be checked. However, for the sake of clarity this check can has been omitted.

The local transformation matrix (LTM) transforms local space into world space. Thus, to find the world space coordinates of a clump's vertex, transform the position of the vertex by the LTM.

RwForAllClumpsInHierarchyInt(), RwForAllClumpsInHierarchyLong(), RwForAllClumpsInHierarchyReal(), and RwForAllClumpsInHierarchyPointer().

RwFindClumpInt(), RwFindClumpLong(), RwFindClumpReal(), RwFindClumpPointer().

Certain versions of RenderWare have platform specific include files. These include files are not automatically included by rwlib.h. See Appendix B for more information.

Although these code fragments are taken from an MS Windows 3.1 application the modifications necessary for other platforms are trivial.

Subsequent errors are most often the direct result of the first error. Such error cascades are not significant. The first error encountered is the source of the problem.

Data Types

The RenderWare library exports a small number of data types and associated functions which operate there on.

All RenderWare API functions and data types are prefixed with Rw. For example:

```
RwRenderScene()
RwClump *
```

All RenderWare enumerated type values are prefixed with rw. For example:

```
rwWIREFRAME
```

From the programmers perspective, there are two major categories of data types within the RenderWare library: public types and opaque types.

Public types may be created and manipulated by the standard C language mechanisms. For example, a field in an object of the public structure type `RwV3d` can be set directly:

```
RwV3d point;
point.x = 1.0;
```

Unlike instances of the public types, opaque objects, such as matrices, cameras, clumps and lights, can only be created and accessed by RenderWare API functions which identify those objects by opaque object pointers. The implementation of opaque types is hidden from the application programmer in much the same manner as the C language standard library `FILE` structure. For example, a field in an object of the opaque type `RwClump` can *only* be set with the appropriate RenderWare API function:

```
RwClump *clump;

clump = RwCreateClump(10, 10);
RwSetClumpTag(clump, 23);
```

Related Topics

Public Types
Opaque Types

Public Types

This category consists of:

- Numeric types (such as `RwReal` and `RwInt32`).
- Structure types (such as `RwV3d`).
- Enumerated types (such as `RwCameraProjection`).
- Bitfield types (such as `RwClumpHints`).

Related Topics

Numeric Types
Structure Types
Enumerated Types
Bitfield Types

Numeric Types

Related Topics

RwReal

Integer Types

RwReal

There are two possible scenarios in which an application program may exploit the [illegible]

- If the application is only to be used with the fixed point version of the library, then:
  - (i) The RenderWare arithmetic macros, e.g., `RMul` and `RAdd`, must be used instead of Cs corresponding arithmetic operators.
  - (ii) Cs built-in types - `int`, `long`, `float` and `double` - should be converted to RenderWares `RwReal` with the conversion macros, e.g., `CREAL` and `INT2REAL`. Prototypes for RenderWare API functions use `RwReal *` rather than `float *`
  - (iii) [illegible]
- [illegible]
- If the floating point version of the library is to be used exclusively, then:
  - (iv) `RwReal` is interpreted by the library as `float`.
  - (v) there is no need to use either the conversion macros or the arithmetic macros

However, the conversion and arithmetic macros will perform correctly with the floating point version of the library and do not incur any run-time performance penalty. Application programmers are encouraged to apply the macros consistently to facilitate any future port between fixed and floating-point libraries.

To facilitate the use of the `RwReal` type, several conversion macros are provided:

- `REAL2INT`(x) takes an `RwReal` variable `x` as an argument and returns an `int`
- [illegible]
- `REAL2FL`(x) takes an `RwReal` variable `x` as an argument and returns a `float`

In addition, there are several arithmetic macros that take `RwReal` arguments and return [illegible]

- [illegible]
- `RAbs(x)     c.f. fabs(x)`
- `RSqrt(x)    c.f. sqrt(x)`

In floating-point versions of the library, the arithmetic macros handle over- and under-flow exactly as their C counterparts. In fixed-point versions of the library, `RAdd` and `RSub` work modulo 2^16 - so that overflow results in wrapping; `RMul` and `RDiv` return the value of greatest possible absolute value and correct sign on overflow; `RMul` returns zero on underflow whilst `RDiv` returns the numerator on division by zero.

Note that the arguments passed to the above type conversion and arithmetic macros may be evaluated multiple times. Therefore, applications should avoid passing arguments that have evaluation side-effects.

The following code fragment demonstrates the application of some of these macros. It is compatible with both fixed-and floating-point versions of the library:

```
#define PI 3.14159265358979323846

RwReal
sinangle(RwReal degrees)
{
     RwReal radians;
     RwReal sine;

     /*
      * Convert to radians.
      */
     radians = RMul(degrees, RDiv(CREAL(PI),CREAL(180)));

     /*
      * Find sin by call to double sin(double x).
      */
     sine = FL2REAL(sin(REAL2FL(radians)));

     return sine;
}
```

The smallest and largest positive real numbers that can be represented are available via the macros `REAL_MIN` and `REAL_MAX` respectively.

Integer Types

RenderWare supports a number of integer types to aid in portability across platforms. These types are defined as follows:

```
typedef short          RwInt16;   /* 16 bit signed integer */
typedef unsigned short RwUInt16;  /* 16 bit unsigned integer */
typedef long           RwInt32;   /* 32 bit signed integer */
typedef unsigned long  RwUInt32;  /* 32 bit unsigned integer */
typedef RwInt32        RwBool;    /* Boolean type */
```

These types should be used throughout a RenderWare application in preference to their underlying native type. This is particularly important when building 16-bit RenderWare applications using either Visual C++ or Borland C++. Certain RenderWare functions, such as **`RwGetCameraViewport`**`()`, require pointers to `RwInt32`s or pointers to arrays of `RwInt32`s. It is essential in 16-bit environments that the objects passed to these functions are declared `RwInt32` and not `int`.

Structure Types

Structure types are normal C structures. Their fields can be set and retrieved directly.
Structure types are always passed to RenderWare API functions by reference.

Related Topics

RwV3d
RwUV
RwRect
RwRGBColor
RwPaletteEntry
Pick Related Structures
RwOpenArgument

RwV3d

The type `RwV3d` represents points and vectors. It is defined as follows:

```
typedef struct
{
    RwReal x, y, z;
} RwV3d;
```

`x, y, z` — x, y and z coordinates of a 3D point or the x, y and z components of a 3D vector

RwUV

The type `RwUV` represents the texture coordinates of a vertex. It is defined as follows:

```
typedef struct
{
    RwReal u, v;          u and v texture coordinates in the range
                          CREAL(0.0)...CREAL(32.0)
} RwUV;
```

RwRect

The type `RwRect` represents a rectangular region. It is defined as follows:

```
typedef struct
{
    RwInt32 x, y, w, h;         x, y - offset; w, h - size
} RwRect;
```

RwRGBColor

The type `RwRGBColor` represents RGB color triples. It is defined as follows:

```
typedef struct
{
    RwReal r, g, b;         Intensities in the range
                            CREAL(0.0)...CREAL(1.0)
} RwRGBColor;
```

RwPaletteEntry

The type `RwPaletteEntry` represents a palette entry. It is defined as follows:

```
typedef struct
{
     unsigned char r, g, b;     Intensities in the range 0 - 255
     unsigned char flags;       Reserved
}
```

Pick Related Structures

Applications often need to identify the object lying under a specified position in the cameras viewport. This can be accomplished with RenderWares pick functions. These return a pointer to an `RwPickRecord` structure, which provides information about the object (if any) which was picked.

A `RwPickRecord` structure is defined as follows:

```
typedef struct
{
    RwPickObject type;                Type of object picked
    union
    {
        RwPickClumpData clump;        Picked clump data
        RwPickVertexData vertex;      Picked vertex data
    }object
} RwPickRecord
```

The `type` field describes the type of object picked. Depending on the value of this field, either the clump or vertex pick data records should be examined.

`RwPickVertexData` and `RwPickClumpData` are respectively defined as follows:

```
typedef struct
{
    RwInt32 vindex;      Index of vertex closest to the pick position
    RwInt32 d2;          Distance squared (in viewport space units) from the pick
                     position to closest vertex
} RwPickVertexData;
```

Fields `vindex` and `d2` specify the index of the closest viewport space vertex to the pick position and the square of distance (in viewport space units) from that vertex to the actual pick position.

```
typedef struct
{
    RwClump *clump;                 Pointer to the clump picked
    RwPolygon3d *polygon;           Pointer to the polygon picked
    RwPickVertexData vertex;        Picked vertex data
    RwV3d wcpoint;                  World coordinate of pick position
}   RwPickClumpData;
```

This structure specifies the clump picked, the polygon picked, the vertex picked, and the pick position on this polygon in world space.

RwOpenArgument

The type `RwOpenArgument` identifies a library configuration option when opening the library with the **`RwOpenExt`**`()` API function. It is defined as follows:

```
typedef struct
{
        RwOpenOption option;     Option identifier
        void *value;             Parameter value
} RwOpenArgument;
```

The field `option` identifies the open option and `value` is a device specific parameter associated with that option. The available options are described in Appendix B.

Enumerated Types

Enumerated types are used when one of a small range of options must be specified. It should be noted that for all the enumerated types defined by RenderWare, the first value in the list is reserved to indicate errors.

Where the list of possible options is fixed in a device and platform independent way the C `enum` construct defines the enumerated type. However, certain RenderWare enumerated types have options which are device or platform specific. In those cases the C `enum` construct is not adopted. A new type name is introduced based on a C integral type and the values of the type are defined as macros.

Related Topics

Axis Alignment Type
Camera Projection Type
Combination Type
Debug Severity Type
Debug State Type
Device Action Type
Device Information Type
Error Code Type
Geometry Sampling Type
Light Type
Light Sampling Type
Open Option Type
Pick Object Type
Search Mode Type
Spline Type
Spline Path Type
State Type
System Information Type
Texture Dither Mode Type
User-Draw Types

Axis Alignment Type

Clumps may have their axes aligned with the Look At, Look Right and Look Up vectors of the camera used to render that clump. This is mainly useful for creating sprites or decals (2D bitmaps aligned with the viewplane of the viewing camera). However, a clumps axis alignment type may be used to constrain the motion of any clump. The axis alignment type is defined as follows:

`RwAxisAlignment`

`rwNAAXISALIGNMENT`	Error code
`rwNOAXISALIGNMENT`	Do not align the clump with the viewing camera.
`rwALIGNAXISZORIENTX`	Preserve the horizontal orientation of the clump when aligning with the viewing camera.
`rwALIGNAXISZORIENTY`	Preserve the vertical orientation of the clump when aligning with the viewing camera.
`rwALIGNAXISXYZ`	Align the X, Y and Z axes of the clump with the Look Right, Look Up and Look At vectors of the viewing camera.

Camera Projection Type

Cameras project according to either a perspective or parallel model. The projection type is as follows:

`RwCameraProjection`

`rwNACAMERAPROJECTION`	Error code
`rwPERSPECTIVE`	Perspective projection
`rwPARALLEL`	Parallel projection

Combination Type

A combination operator can be applied to several objects, most commonly matrices. The operator determines the order in which one object is combined with another: pre-concatenation, post-concatenation, or replacement. The combination operator is as follows:

`RwCombineOperation`

`rwNACOMBINEOPERATION`	Error code
`rwREPLACE`	Assignment (replace existing value)
`rwPRECONCAT`	Pre-concatenation
`rwPOSTCONCAT`	Post-concatenation

Debug Severity Type

Debugging or trace messages from debugging libraries are output to a debugging stream. Each message can be issued at one of three levels of severity: informational, warning or error. The severity level is as follows:

`RwDebugSeverity`

`rwNADEBUGMESSAGESEVERITY`	Error code
`rwINFORM`	Control flow annotation
`rwWARNING`	Non-fatal exception
`rwERROR`	Fatal exception

Debug State Type

Previous versions of RenderWare provided a type (`RwDebugState`) to explicitly represent the debug trace state. In RenderWare V1.4 the debug trace state is represented by the new generic state type `RwState`. To maintain backwards compatibility `RwDebugState` has been retained as a synonym for `RwState`. As `RwDebugState` will be removed from the API in the next release of RenderWare, references to `RwDebugState` should be replaced by `RwState`.

`RwDebugState` is as follows:

`RwDebugState`

`rwNADEBUGMESSAGESTATUS`	Error code
`rwDISABLE`	Disable messages
`rwENABLE`	Enable messages

Device Action Type

The RenderWare API function **RwDeviceControl**() performs device specific actions. Values of the type RwDeviceAction identify what action **RwDeviceControl**() performs.

The device information type is as follows:

RwDeviceAction

rwNADEVICEACTION	Error code

Specific device drivers may define additional device actions. Appendix B documents any additional (platform specific) actions.

Device Information Type

The API function **`RwGetDeviceInfo`**`()` returns information about the current RenderWare device driver. Values of the type `RwDeviceInfo` identify what device information **`RwGetDeviceInfo`**`()` returns.

The device information type is as follows:

`RwDeviceInfo`

`rwNADEVICEINFO`	Error code
`rwRENDERDEPTH`	Current render depth
`rwINDEXEDRENDERING`	Rendering with an index color scheme (color table) or direct color.
`rwPALETTEBASED`	Does the output device have a palette that RenderWare will attempt to modify.

The following options only apply to palette based devices.

`rwPALETTE`	Fetch the RenderWare palette
`rwPALETTESIZE`	The number of entries in the entire palette
`rwFIRSTPALETTEENTRY`	Index of the first palette entry available for use by an application
`rwLASTPALETTEENTRY`	Index of the last palette entry available for use by an application

Specific device drivers may define additional information types. Appendix B documents any additional (platform specific) options.

Error Code Type

For a full description of the error code type, `RwErrorCode`, see Appendix C: Error Codes

Geometry Sampling Type

Geometry may be visualised in one of three ways: as a cloud of points representing polygon vertices, as a wireframe of polygon edges, or as a solid enclosed by filled polygons. The geometry sampling type is as follows:

`RwGeometrySampling`

`rwNAGEOMETRYSAMPLING`	Error code
`rwPOINTCLOUD`	Render vertices
`rwWIREFRAME`	Render edges
`rwSOLID`	Render polygons

Light Type

Lights may be directional, point, or conical sources. The light type is as follows:

`RwLightType`

`rwNALIGHTTYPE`	Error code
`rwDIRECTIONAL`	Directional light source
`rwPOINT`	Point light source
`rwCONICAL`	Conical light source

Light Sampling Type

The lighting of a polygon in a clump can be calculated in either of two ways: a single lighting sample per polygon at the polygons center, or several lighting samples per polygon - one at each polygon vertex. The former is called flat shading (shading is constant over a polygons entire surface). The latter is called smooth shading (shading may vary over the polygon to yield an apparently smooth surface). These options may be selected by setting the light sampling type of a polygons material to `rwFACET` or `rwVERTEX` respectively.

The light sampling type is as follows:

`RwLightSampling`

`rwNALIGHTSAMPLING`	Error code
`rwFACET`	Flat shading
`rwVERTEX`	Smooth shading

Open Option Type

Library configuration options may be specified when opening the library with the **`RwOpenExt`**`()` API function. Values of the type `RwOpenOption` identify the various options. The open options are as follows:

`RwOpenOption`

`rwNAOPENOPTION`	Error code
`rwNOOPENOPTION`	Null (ignored) option
`rwGAMMACORRECT`	Enable gamma correction

Specific device drivers may define additional open options. Appendix B documents any additional (platform specific) options.

Pick Object Type

RenderWares pick functions return information about a pick operation in a pick record (supplied by reference as an argument). The pick type identifies this record as a clump pick data record, and is as follows:

`RwPickObject`

`rwNAPICKOBJECT`	No Clump picked
`rwPICKCLUMP`	Clump picked

Search Mode Type

Several API functions operate on the texture dictionary stack. The behavior of some of these, e.g., **RwGetNamedTexture**(), is determined by the current search mode. This can have either of two values: `rwLOCAL`, specifying a search limited to the top dictionary on the stack or `rwGLOBAL`, specifying a search through all dictionaries on the stack. The search mode type is as follows:

`RwSearchMode`

`rwNASEARCHMODE`	Error code
`rwLOCAL`	Search locally (top-most dictionary only)
`rwGLOBAL`	Search globally (all dictionaries)

Spline Type

A spline is described by a set of control points through which the spline passes. The spline can be open or closed. In the former case the curve starts at the first control point and ends at the last point. In the later case the last point is joined to the first point. The spline type is as follows:

`RwSplineType`

`rwNASPLINETYPE`	Error code
`rwOPENLOOP`	Open spline (end points not joined)
`rwCLOSEDLOOP`	Closed spline (end points joined)

RenderWare splines are interpolatory, non-rational, cubic B-splines. The `rwOPENLOOP` type is open in having disjoint start and end points whilst the `rwCLOSEDLOOP` type is closed in forming a complete circuit.

Spline Path Type

Once created, a spline can be sampled as a parametric curve. Points on the spline may be computed by specifying a parameter in the range [`CREAL(0.0)` - `CREAL(1.0)`]. The spline path type specifies the manner of interpolation. An `rwSMOOTH` path is interpolated uniformly across the range, whilst a `rwNICEENDS` path will have zero differential at its ends. The spline path type is as follows:

`RwSplinePath`

`rwNASPLINEPATHTYPE`	Error code
`rwSMOOTH`	Uniform interpolation across the range.
`rwNICEENDS`	Zero differential at ends.

State Type

This is a generic state type. It is used whenever an attribute can be switched on or off. The state type is as follows:

`RwState`

`rwNASTATE`	Error code
`rwOFF`	Object is turned off
`rwON`	Object is turned on

System Information Type

The API function **`RwGetSystemInfo`**`()` returns information about the current RenderWare library. Values of the type `RwSystemInfo` identify the information returned by **`RwGetSystemInfo`**`()`.

The system information type is as follows:

`RwSystemInfo`

`rwNASYSTEMINFO`	Error code
`rwVERSIONSTRING`	A string including major and minor version numbers and the release string
`rwVERSIONMAJOR`	The major version number of the library
`rwVERSIONMINOR`	The minor version number of the library
`rwVERSIONRELEASE`	A string identifying a particular release of RenderWare
`rwFIXEDPOINTLIB`	Does the current library using fixed-point numerics
`rwDEBUGGINGLIB`	Does the current library use the RenderWare debugging kernel

Texture Dither Mode Type

The `RwTextureDitherMode` type controls whether the apparent color resolution of textures should be enhanced by dithering. It is important to note that textures are dithered when being read but not during rendering. These flags, therefore, only apply when a texture is first read from disk.

Dithering is only appropriate if the texture being read has not already been dithered. Previous versions of RenderWare attempted to avoid re-dithering by examining the size and depth of the texture being read. If the texture did not have to be resized and was already of the correct depth then it would not be dithered. Otherwise was dithering was performed. Current versions of RenderWare provide finer grain control over texture dithering. There are three texture dithering modes; `rwDITHERON` forces textures to be dithered when loaded, `rwDITHEROFF` prevents textures from being dithered and `rwAUTODITHER` provides backwards compatibility by deciding whether to dither according to the size and depth of the image read.

The texture dither mode is as follows:

`RwTextureDitherMode`

`rwNATEXTUREDITHER`	Error code
`rwDITHERON`	Always dither
`rwDITHEROFF`	Never dither
`rwAUTODITHER`	Dither if necessary

User-Draw Types

An application may supplement the image generated by RenderWares 3D rendering with 2D graphics such as labels. Such composition is supported via call-back functions known as User-Draws. The enumerated type `RwUserDrawType` represents the alignment of the user-draw object relative to the associated RenderWare object. A user-draw object may be aligned with the origin of the owning clump, with a vertex of the owning clump, with the bounding box of the owning clump or with a cameras viewport. The user-draw type is defined as follows:

`RwUserDrawType`

`rwNAUSERDRAWTYPE`	Error code
`rwCLUMPALIGN`	Align to a clumps origin
`rwVERTEXALIGN`	Align to a clumps vertex
`rwBBOXALIGN`	Align to a clumps viewport bounding box
`rwVPALIGN`	Align to a cameras viewport

Bitfield Types

Like enumerated types, bitfields are used when selecting from a small number of alternatives. Unlike enumerated types, however, bitfields allow the selection of several independent options simultaneously.

The bitfield types can be manipulated with the C bitwise manipulation operators: | (or), & (and), ~ (not) and ^ (xor). Furthermore, 0 is a valid value for any bitfield and indicates that none of the available options have been selected.

Related Topics

Raster Options
Palette Options
Clump Hints
Texture Modes
UserDraw Alignments

Raster Options

The bitfield type `RwRasterOptions` controls several aspects of raster loading, these options being resizing, dithering and gamma correction. These options are specified when loading a raster with **RwReadRaster**() or when creating a raster from a platform specific bitmap with **RwBitmapRaster**().

To understand the effect of raster options, the size and depth constraints of rasters should be appreciated. The depth of a raster is always equal to RenderWares current rendering depth (8 or 16 bit in version 1.4). If the source of a raster (either a bitmap file or a platform specific bitmap object) is of a different depth to the rendering depth, the source pixels will be converted to that depth. Furthermore, if a raster is to be employed as the pixel source for a texture map it may have to be changed in size. Single frame texture maps have a fixed width and height of 128 pixels. Multi-frame texture maps have a fixed width of 128 pixels and a height of n * 128 pixels (where n is the number of frames).

The raster options are as follows:

`rwFITRASTER` will resize the raster to texture map dimensions. Specify this option if the raster is to be selected into a texture with **RwCreateTexture**() or **RwSetTextureRaster**().

`rwGAMMARASTER` will gamma correct the raster. Do not specify this option if the source of the raster has already been gamma corrected.

`rwAUTODITHERRASTER` dithers a raster only if it has been resized (the source bitmap was not of texture map size and `rwFITIMAGE` was specified) or changed in depth. This option mirrors the `rwAUTODITHER` texture dither mode. `rwAUTODITHERRASTER` must not be specified with `rwDITHERRASTER`.

`rwDITHERASTER` forces dithering of a raster. If neither `rwAUTODITHERRASTER` nor `rwDITHERRASTER` is specified the raster will not be dithered.

`RwRasterOptions`

`rwAUTODITHERRASTER`	Dither the raster if necessary
`rwDITHERRASTER`	Dither the raster
`rwFITRASTER`	Resize the raster to the appropriate size for a texture
`rwGAMMARASTER`	Gamma correct the raster

Palette Options

The bitfield type `RwPaletteOptions` specifies the options that can be performed when setting the palette entries.

The palette options are as follows:

`rwGAMMAPALETTE` will gamma correct the palette. Do not specify this option if the source of the palette has already been gamma corrected. This option will normally be used when reading the palette from a bitmap which will be loaded via **`RwReadRaster`**`()` with the corresponding `rwGAMMARASTER` option set.

`RwPaletteOptions`	
`rwGAMMAPALETTE`	Gamma correct the palette.

Clump Hints

The bitfield type `RwClumpHints` optimizes rendering by passing hints to RenderWare about the nature of a clump and the environment in which it is to be rendered.

The clump hints are as follows:

`rwCONTAINER` marks the clump as a container - a clump which spatially contains other clumps. For example, a clump representing a room should normally have the `rwCONTAINER` hint set.

`rwHS` specifies that hidden surfaces should be removed when the clump is rendered.

`rwEDITABLE` marks the clump as being editable (its vertices may be moved and new vertices and polygons added).

`RwClumpHints`

`rwCONTAINER`	Clump is a container
`rwHS`	Apply hidden surface removal to the clump
`rwEDITABLE`	Clumps geometry may be edited

Texture Modes

The bitfield type `RwTextureModes` provides fine grain control over the rendering of textures. Three textures modes are defined: `rwLIT`, `rwFORESHORTEN` and `rwFILTER`.

If `rwLIT` is specified, the associated texture will be lit according to the current light sampling type of the material (`rwFACET` or `rwVERTEX`). If it is not specified, the texture will not be affected by lighting; its luminance will be as tabulated in the textures bitmap data.

`rwFORESHORTEN` controls the interpolation of texture coordinates. A texture image is applied by assigning a texel color to each pixel within a projected polygon from an associated position within that image. This position is specified by a pair of texture coordinates, conventionally called (u, v). These coordinates are each in the range [0.0 - 32.0] and measure how far to the right and below the upper left corner of the image (respectively) to take the texel color. Each vertex in a clump may be assigned a pair of texture coordinates. A textured image is "wrapped" over a polygon by interpolating texture coordinates defined at the vertices over the polygon's projection.

By default, the texture coordinates specified at the vertices are interpolated bilinearly over the screen projection of a polygon. This method of texture coordinate interpolation is analogous to the luminance interpolation of Gouraud shading, and gives high speed performance. For perspective views however, this bilinear interpolation is only mathematically correct if the polygon is at a constant depth from - and therefore lies in a plane parallel to -- the view plane. The bilinear screen space interpolation of texture coordinates over the projection of a polygon which is not parallel to the view plane will "drift" away from their true values when the interpolation is distant from the projected vertices, although the interpolation always synchronizes with exact values at the vertices. Such inaccuracies are often negligible, but can be noticeable if a polygon extends over a significant depth range relative to the screen or projects to a large screen area. A large depth range introduces inaccuracies into the bilinear interpolation, whilst a large projected area allows plenty of screen "real estate" away from vertices over which these may accumulate. Such inaccuracies manifest themselves by the applied texture not being foreshortened as would be expected in a perspective image.

The `rwFORESHORTEN` texture mode will ensure that exact texture coordinates are interpolated over the entire screen projection of a polygon, thereby rendering the expected foreshortening. However, the arithmetic underlying this interpolation can be significantly more involved than that for bilinear interpolation. The indiscriminate application of this texture mode can degrade performance to an unacceptable level. The `rwFORESHORTEN` texture mode should only applied when texturing polygons for which bilinear interpolation is significantly inaccurate -- those extending over a significant depth range relative to the screen plane and projecting to a large screen area. The interpolation adopted in the `rwFORESHORTEN` texture mode has been optimized for polygons projecting to a large screen area, so that visual and performance integrity may be ensured simultaneously.

The `rwFILTER` texture mode reduces texture aliasing artifacts arising from extreme texellation to make the texture map appear more continuous. By default, when "zooming in" to a textured polygon, the square texel boundaries become clearly visible, emphasizing discrete texel steps in the texture map. The texture map may be made to appear more continuous and hence realistic, since such sharp transitions tend not to occur in real world texture detail, by applying the `rwFILTER` texture mode. This mode smoothes the transition between adjacent texels to reduce the sharp edges which would otherwise appear.

`RwTextureModes`

`rwLIT`	Texture is illuminated by light sources.
`rwFORESHORTEN`	Texture is foreshortened in a perspectively correct manner.
`rwFILTER`	A filter will be applied to the texture to reduce the effect of pixelation due to aliasing.

UserDraw Alignments

The bitfield type `RwUserDrawAlignmentTypes` controls the justification of a user draw relative to the object with which it is associated - clump vertex, clump bounding box, clump origin or camera viewport. The four options, `rwALIGNTOP`, `rwALIGNBOTTOM`, `rwALIGNLEFT` and `rwALIGNRIGHT`, respectively justify the top, bottom, left or right edge of a user draw with the associated object. These options may be combined; for example, specifying `rwALIGNTOP | rwALIGNLEFT`, will align the top left hand corner of the user draw with the associated object. Two simplification options are provided for common types of alignment, `rwALIGNTOPLEFT` and `rwALIGNBOTTOMRIGHT`. Specifying both `rwALIGNTOP` and `rwALIGNBOTTOM` is not permitted. Specifying both `rwALIGNLEFT` and `rwALIGNRIGHT` is not permitted. If no options are selected the user draw will be centered about the associated object.

`RwUserDrawAlignmentTypes`

`rwALIGNTOP`	Align to top edge
`rwALIGNBOTTOM`	Align to bottom edge
`rwALIGNLEFT`	Align to left edge
`rwALIGNRIGHT`	Align to right edge
`rwALIGNTOPLEFT`	Align to top and left edges
`rwALIGNBOTTOMRIGHT`	Align to bottom and right edges

Opaque Types

Most RenderWare objects are opaque - their implementation is hidden from the user. These objects are created by RenderWare API functions which return pointers thereto. Such opaque objects can only be accessed through RenderWare API functions. These functions require that the target object is identified by its opaque object pointer. As such, opaque objects are similar to the C standard library `FILE` structure.

The opaque object types are:

`RwCamera *`	A camera
`RwClump *`	A collection of polygons and vertices
`RwLight *`	A light
`RwMaterial *`	A set of attributes defining a surface material
`RwMatrix4d *`	A 4 x 4 transformation matrix
`RwPolygon3d *`	A polygon
`RwRaster *`	A bitmap (used by textures or as a camera backdrop)
`RwScene *`	A scene
`RwSpline *`	A spline
`RwTexture *`	A texture (single or multi-frame)
`RwUserDraw *`	A 2D, application drawn object

Fixed-point compatibility macros omitted for clarity.

Error checking omitted for clarity.

Previous versions of RenderWare included a second real number type, RPARAM. However, with this release of RenderWare RPARAM has been removed and all real numbers are repredented by the RwReal type. References to RPARAM should be removed from all code.

Function Index Alphabetically

RwGetLightType()
RwGetLightVector()
RwGetMaterialAmbient()
RwGetMaterialColor()
RwGetMaterialDiffuse()
RwGetMaterialGeometrySampling()
RwGetMaterialLightSampling()
RwGetMaterialOpacity()
RwGetMaterialSpecular()
RwGetMaterialTexture()
RwGetMaterialTextureModes()
RwGetMatrixElement()
RwGetMatrixElements()
RwGetNamedTexture()
RwGetNextClump()
RwGetNumNamedTextures()
RwGetPaletteEntries()
RwGetPolygonAmbient()
RwGetPolygonCenter()
RwGetPolygonColor()
RwGetPolygonData()
RwGetPolygonDiffuse()
RwGetPolygonGeometrySampling()
RwGetPolygonLightSampling()
RwGetPolygonMaterial()
RwGetPolygonNormal()
RwGetPolygonNumSides()
RwGetPolygonOpacity()
RwGetPolygonOwner()
RwGetPolygonSpecular()
RwGetPolygonTag()
RwGetPolygonTexture()
RwGetPolygonTextureModes()
RwGetPolygonUV()
RwGetPolygonVertices()
RwGetRasterData()
RwGetRasterDepth()
RwGetRasterHeight()
RwGetRasterPixels()
RwGetRasterStride()
RwGetRasterWidth()
RwGetSceneData()
RwGetSceneNumClumps()
RwGetSceneNumLights()
RwGetShapePath()
RwGetSplineData()
RwGetSplineNumPoints()
RwGetSplinePoint()
RwGetSystemInfo()
RwGetTextureData()
RwGetTextureDictSearchMode()
RwGetTextureDithering()
RwGetTextureFrame()
RwGetTextureFrameStep()
RwGetTextureGammaCorrection()

Function Index by Category

Related Topics

Camera Functions
Clump Functions
Debug Functions
Error Functions
Light Functions
Material Functions
Matrix Functions
Object Builder Functions
Point / Vector Functions
Polygon Functions
Raster Functions
Scene Functions
Shape Functions
Spline Functions
Texture Functions
User Draw Functions
Other Functions

Camera Functions

RwBeginCameraUpdate() Changed
RwClearCameraViewport()
RwCreateCamera()
RwDamageCameraViewport()
RwDestroyCamera()
RwDuplicateCamera()
RwEndCameraUpdate()
RwGetCameraBackColor()
RwGetCameraBackdrop()
RwGetCameraBackdropOffset()
RwGetCameraBackdropViewportRect()
RwGetCameraData()
RwGetCameraFarClipping() New
RwGetCameraImage()
RwGetCameraLTM() New
RwGetCameraLookAt()
RwGetCameraLookRight()
RwGetCameraLookUp()
RwGetCameraNearClipping()
RwGetCameraPosition()
RwGetCameraProjection()
RwGetCameraViewOffset()
RwGetCameraViewport()
RwGetCameraViewportRaster()
RwGetCameraViewwindow()
RwInvalidateCameraViewport()
RwPanCamera()
RwPointCamera()
RwResetCamera()
RwRevolveCamera()
RwSetCameraBackColor()
RwSetCameraBackColorStruct()
RwSetCameraBackdrop()
RwSetCameraBackdropOffset()
RwSetCameraBackdropViewportRect()
RwSetCameraData()
RwSetCameraFarClipping() New
RwSetCameraLookAt()
RwSetCameraLookUp()
RwSetCameraNearClipping()
RwSetCameraPosition()
RwSetCameraProjection()
RwSetCameraViewOffset()
RwSetCameraViewport()
RwSetCameraViewwindow()
RwShowCameraImage() Changed
RwTiltCamera()
RwTransformCamera() Changed
RwTransformCameraOrientation()
RwUndamageCameraViewport()
RwVCMoveCamera()
RwWCMoveCamera()

Clump Functions

RwAddChildToClump()
RwAddHintToClump()
RwAddPolygonsToClump()
RwAddPolygonToClump()
RwAddVertexToClump()
RwCalculateClumpVertexNormal()
RwClumpDistance()
RwCreateClump()
RwCreateSprite()
RwCubicTexturizeClump()
RwDestroyClump()
RwDuplicateClump()
RwEnvMapClump()
RwFindClump()
RwFindClumpInt()
RwFindClumpLong() Obsolete
RwFindClumpPointer()
RwFindClumpReal()
RwFindTaggedClump()
RwForAllClumpsInHierarchy()
RwForAllClumpsInHierarchyInt()
RwForAllClumpsInHierarchyLong() Obsolete
RwForAllClumpsInHierarchyPointer()
RwForAllClumpsInHierarchyReal()
RwForAllPolygonsInClump()
RwForAllPolygonsInClumpInt()
RwForAllPolygonsInClumpLong() Obsolete
RwForAllPolygonsInClumpPointer()
RwForAllPolygonsInClumpReal()
RwGetClumpAxisAlignment()
RwGetClumpBBox()
RwGetClumpData()
RwGetClumpHints()
RwGetClumpJointMatrix()
RwGetClumpLocalBBox() New
RwGetClumpLTM()
RwGetClumpMatrix()
RwGetClumpNumChildren()
RwGetClumpNumPolygons()
RwGetClumpNumVertices()
RwGetClumpOrigin()
RwGetClumpOwner()
RwGetClumpParent()
RwGetClumpRoot()
RwGetClumpState()
RwGetClumpTag()
RwGetClumpVertex()
RwGetClumpVertexNormal()
RwGetClumpVertexUV()
RwGetClumpVertexViewportPosition()
RwGetClumpViewportRect()
RwGetFirstChildClump()
RwGetNextClump()

RwNormalizeClump()
RwPickClump()
RwRemoveChildFromClump()
RwRemoveHintFromClump()
RwRenderClump()
RwSetClumpAxisAlignment()
RwSetClumpData()
RwSetClumpHints()
RwSetClumpState()
RwSetClumpTag()
RwSetClumpVertex()
RwSetClumpVertexNormal()
RwSetClumpVertexUV()
RwSetClumpVertices()
RwSphericalTexturizeClump()
RwTransformClump()
RwTransformClumpJoint()

Debug Functions

RwCloseDebugStream()	
RwGetDebugAssertionState()	Changed
RwGetDebugMessageState()	Changed
RwGetDebugScriptState()	Changed
RwGetDebugSeverity()	
RwGetDebugTraceState()	New
RwOpenDebugStream()	
RwSetDebugAssertionState()	Changed
RwSetDebugMessageState()	Changed
RwSetDebugOutputState()	Changed
RwSetDebugScriptState()	Changed
RwSetDebugSeverity()	
RwSetDebugStream()	NonDLL
RwSetDebugTraceState()	New

Error Functions

RwGetError()
RwGetInternalError()
RwSetUserError()

Light Functions

RwCreateLight()	
RwDestroyLight()	
RwDuplicateLight()	
RwGetLightBrightness()	
RwGetLightColor()	New
RwGetLightConeAngle()	
RwGetLightData()	
RwGetLightOwner()	
RwGetLightPosition()	
RwGetLightState()	
RwGetLightType()	
RwGetLightVector()	
RwSetLightBrightness()	
RwSetLightColor()	New
RwSetLightColorStruct()	New
RwSetLightConeAngle()	
RwSetLightData()	
RwGetLightLTM()	New
RwSetLightPosition()	
RwSetLightState()	
RwSetLightVector()	
RwTransformLight()	Changed

Material Functions

RwAddTextureModeToMaterial()
RwCopyMaterial()
RwCreateMaterial()
RwCurrentMaterial()
RwDestroyMaterial()
RwDuplicateMaterial()
RwGetMaterialAmbient()
RwGetMaterialColor()
RwGetMaterialDiffuse()
RwGetMaterialGeometrySampling()
RwGetMaterialLightSampling()
RwGetMaterialOpacity()
RwGetMaterialSpecular()
RwGetMaterialTexture()
RwGetMaterialTextureModes()
RwPopCurrentMaterial()
RwPushCurrentMaterial()
RwRemoveTextureModeFromMaterial()
RwSetMaterialAmbient()
RwSetMaterialColor()
RwSetMaterialColorStruct()
RwSetMaterialDiffuse()
RwSetMaterialGeometrySampling()
RwSetMaterialLightSampling()
RwSetMaterialOpacity()
RwSetMaterialSpecular()
RwSetMaterialSurface()
RwSetMaterialTexture()
RwSetMaterialTextureModes()

Matrix Functions

Object Builder Functions

RwVertexExt()

Point / Vector Functions

RwAddVector()
RwCrossProduct()
RwDotProduct()
RwNormalize()
RwScaleVector()
RwSubtractVector()
RwTransformPoint()
RwTransformVector()

Polygon Functions

RwAddTextureModeToPolygon()
RwDestroyPolygon()
RwFindTaggedPolygon()
RwGetPolygonAmbient()
RwGetPolygonCenter()
RwGetPolygonColor()
RwGetPolygonData()
RwGetPolygonDiffuse()
RwGetPolygonGeometrySampling()
RwGetPolygonLightSampling()
RwGetPolygonMaterial()
RwGetPolygonNormal()
RwGetPolygonNumSides()
RwGetPolygonOpacity()
RwGetPolygonOwner()
RwGetPolygonSpecular()
RwGetPolygonTag()
RwGetPolygonTexture()
RwGetPolygonTextureModes()
RwGetPolygonUV()
RwGetPolygonVertices()
RwRemoveTextureModeFromPolygon()
RwReversePolygonFace()
RwSetPolygonAmbient()
RwSetPolygonColor()
RwSetPolygonColorStruct()
RwSetPolygonData()
RwSetPolygonDiffuse()
RwSetPolygonGeometrySampling()
RwSetPolygonLightSampling()
RwSetPolygonMaterial()
RwSetPolygonOpacity()
RwSetPolygonSpecular()
RwSetPolygonSurface()
RwSetPolygonTag()
RwSetPolygonTexture()
RwSetPolygonTextureModes()
RwSetPolygonUV()

Raster Functions

RwBitmapRaster()	Changed
RwCreateRaster()	
RwDestroyRaster()	
RwDuplicateRaster()	
RwGetRasterData()	
RwGetRasterDepth()	
RwGetRasterHeight()	
RwGetRasterPixels()	Changed
RwGetRasterStride()	
RwGetRasterWidth()	
RwReadMaskRaster()	
RwReadRaster()	
RwReleaseRasterPixels()	New
RwSetRasterData()	

Scene Functions

RwAddClumpToScene()	
RwAddLightToScene()	
RwCreateScene()	
RwDefaultScene()	
RwDestroyScene()	
RwForAllClumpsInScene()	
RwForAllClumpsInSceneInt()	
RwForAllClumpsInSceneLong()	Obsolete
RwForAllClumpsInScenePointer()	
RwForAllClumpsInSceneReal()	
RwForAllLightsInScene()	
RwForAllLightsInSceneInt()	
RwForAllLightsInSceneLong()	Obsolete
RwForAllLightsInScenePointer()	
RwForAllLightsInSceneReal()	
RwGetSceneData()	
RwGetSceneNumClumps()	
RwGetSceneNumLights()	
RwPickScene()	
RwRemoveClumpFromScene()	
RwRemoveLightFromScene()	
RwRenderScene()	
RwSetSceneData()	

Shape Functions

RwGetShapePath()
RwReadShape()
RwSetShapePath()
RwWriteShape()

Spline Functions

Texture Functions

RwAddTextureModeToSurface()
RwCreateTexture()
RwDestroyTexture()
RwFindNamedTexture()
RwForAllNamedTextures()
RwForAllNamedTexturesInt()
RwForAllNamedTexturesLong() Obsolete
RwForAllNamedTexturesPointer()
RwForAllNamedTexturesReal()
RwGetNamedTexture()
RwGetNumNamedTextures()
RwGetTextureData()
RwGetTextureDictSearchMode()
RwGetTextureDithering()
RwGetTextureFrame()
RwGetTextureFrameStep()
RwGetTextureGammaCorrection()
RwGetTextureName()
RwGetTextureNumFrames()
RwGetTextureRaster()
RwMaskTexture()
RwReadNamedTexture()
RwReadTexture()
RwRemoveTextureModeFromSurface()
RwSetTextureData()
RwSetTextureDictSearchMode()
RwSetTextureDithering()
RwSetTextureFrame()
RwSetTextureFrameStep()
RwSetTextureGammaCorrection()
RwSetTextureRaster()
RwTextureDictBegin()
RwTextureDictEnd()
RwTextureNextFrame()

User Draw Functions

RwAddUserDrawToClump()	NonDLL	
RwCreateUserDraw()	NonDLL	
RwDestroyUserDraw()	NonDLL	
RwDuplicateUserDraw()	NonDLL	
RwForAllUserDrawsInClump()	NonDLL	
RwForAllUserDrawsInClumpInt()	NonDLL	
RwForAllUserDrawsInClumpLong()	NonDLL	Obsolete
RwForAllUserDrawsInClumpPointer()	NonDLL	
RwForAllUserDrawsInClumpReal()	NonDLL	
RwGetClumpNumUserDraws()	NonDLL	
RwGetUserDrawAlignment()	NonDLL	
RwGetUserDrawCallback()	NonDLL	
RwGetUserDrawData()	NonDLL	
RwGetUserDrawOwner()	NonDLL	
RwGetUserDrawParentAlignment()	NonDLL	
RwGetUserDrawSize()	NonDLL	
RwGetUserDrawType()	NonDLL	
RwGetUserDrawVertexIndex()	NonDLL	
RwRemoveUserDrawFromClump()	NonDLL	
RwSetUserDrawAlignment()	NonDLL	
RwSetUserDrawCallback()	NonDLL	
RwSetUserDrawData()	NonDLL	
RwSetUserDrawOffset()	NonDLL	
RwSetUserDrawParentAlignment()	NonDLL	
RwSetUserDrawSize()	NonDLL	
RwSetUserDrawType()	NonDLL	
RwSetUserDrawVertexIndex()	NonDLL	

Other Functions

RwClose()	
RwDeviceControl()	Changed
RwGetDeviceInfo()	Changed
RwGetPaletteEntries()	New
RwGetSystemInfo()	Changed
RwOpen()	Changed
RwOpenExt()	Changed
RwRandom()	New
RwSetPaletteEntries()	New
RwSRandom()	New

RwClump *
RwAddChildToClump(RwClump *parent, RwClump *child);

Description

Makes the second clump a child of the first. If `child` is already a child of another clump, it will be removed from that clumps list of children before being added to parent.

Arguments

`parent`	Pointer to the parent clump.
`child`	Pointer to the child clump.

Return Value

The argument `parent` if successful, and `NULL` otherwise.

Comments

After addition, the childs modeling and joint (articulation) transformations will be relative to those of its new parent.

See Also

RwGetClumpNumChildren()
RwGetClumpParent()
RwGetFirstChildClump()
RwGetNextClump()
RwRemoveChildFromClump()

```
RwScene *
RwAddClumpToScene(RwScene *scene, RwClump *clump);
```

Description

Adds the clump (and all its descendants) to the scene.

Arguments

`scene`	Pointer to the scene.
`clump`	Pointer to the clump.

Return Value

The argument `scene` if successful, and `NULL` otherwise.

Comments

Note that the clump being added must not have a parent (i.e., it must be a root clump).

See Also

RwAddLightToScene()
RwDestroyClump()
RwDestroyScene()
RwForAllClumpsInScene()
RwGetClumpOwner()
RwGetSceneNumClumps()
RwRemoveClumpFromScene()

`RwBool`
`RwAddHint``(RwClumpHints hints);`

Description

Adds a hint (or set of hints) to the current clump under construction. A clumps hints enable RenderWare to render a scene containing that clump more efficiently.

Arguments

`hints`	A bitfield representing a hint (or bitwise or of hints).

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

Currently, the following hints are supported:

`rwCONTAINER`	The clump spatially contains other clumps.
`rwHS`	Action should be taken to prevent hidden surfaces from being visible when the clump is rendered.
`rwEDITABLE`	The clumps geometry is editable (its vertices can be moved and new vertices and polygons added).

This function can only be called within the context of an **`RwClumpBegin`**`()` … **`RwClumpEnd`**`()` block.

See Also

`RwAddHintToClump``()`
`RwClumpBegin``()`
`RwClumpEnd``()`
`RwRemoveHint``()`
`RwSetHints``()`

```
RwClump *
RwAddHintToClump(RwClump *clump, RwClumpHints hint);
```

Description

Adds a hint (or set of hints) to the clump. A clumps hints enable RenderWare to render a scene containing that clump more efficiently.

Arguments

`clump`	Pointer to the clump.
`hint`	A bitfield representing a hint (or bitwise or of hints).

Return Value

The argument `clump` if successful, and `NULL` otherwise.

Comments

Currently, the following hints are supported:

`rwCONTAINER`	The clump spatially contains other clumps.
`rwHS`	Action should be taken to prevent hidden surfaces from being visible when the clump is rendered.
`rwEDITABLE`	The clumps geometry is editable (its vertices can be moved and new vertices and polygons added).

See Also

RwAddHint()
RwGetClumpHints()
RwRemoveHintFromClump()
RwSetClumpHints()

```
RwScene *
RwAddLightToScene(RwScene *scene, RwLight *light);
```

Description

Adds the light to the scene.

Arguments

`scene`	Pointer to the scene.
`light`	Pointer to the light.

Return Value

The argument `scene` if successful, and `NULL` otherwise.

See Also

RwDestroyLight()
RwDestroyScene()
RwForAllLightsInScene()
RwGetLightOwner()
RwGetSceneNumLights()
RwRemoveLightFromScene()

`RwClump *`
`RwAddPolygonsToClump``(RwClump *dest, RwClump *source);`

Description

Makes copies of all the polygons (and their associated vertices and materials) in the source clump and adds them to the destination clump.

Arguments

`dest`	Pointer to the destination clump.
`source`	Pointer to the source clump.

Return Value

The argument `dest` if successful, and `NULL` otherwise.

Comments

As this function modifies the geometry of the destination clump, the destination clump is made editable (the `rwEDITABLE` hint is set).

See Also

`RwAddHintToClump``()`
`RwAddPolygonToClump``()`
`RwAddVertexToClump``()`
`RwForAllPolygonsInClump``()`
`RwGetClumpNumPolygons``()`
`RwGetPolygonOwner``()`

RwPolygon3d *
RwAddPolygonToClump(RwClump *clump, RwInt32 sides, RwInt32 *vlist);

Description

Creates a polygon and adds it to the clump. The current material is applied to the new polygon.

Arguments

`clump`	Pointer to the clump.
`sides`	Number of sides of the polygon.
`vlist`	Pointer to an array of `RwInt32`s containing the vertex indices of the new polygon.

Return Value

A pointer to the new polygon if successful, and `NULL` otherwise.

Comments

As this function modifies the geometry of the destination clump, the destination clump is made editable (the `rwEDITABLE` hint is set).

For 16-bit applications accessing the RenderWare DLL the vertex index list pointed to by `vlist` must be declared as an array of `RwInt32`s and not `int`s.

See Also

RwAddHintToClump()
RwAddPolygonsToClump()
RwAddVertexToClump()
RwForAllPolygonsInClump()
RwGetClumpNumPolygons()
RwGetPolygonOwner()

```
RwMaterial *
RwAddTextureModeToMaterial(RwMaterial *material,
        RwTextureModes mode);
```

Description

Adds the given texture mode (or modes) to the material. Texture modes permit fine grain control over the rendering of textures.

Arguments

`material`	Pointer to the material.
`mode`	A bitfield representing a texture mode (or bitwise or of modes).

Return Value

The argument `material` if successful, and `NULL` otherwise.

Comments

The following texture modes are supported:

`rwLIT`	The texture will be lit according to the current light sampling type of the material (`rwFACET` or `rwVERTEX`).
`rwFORESHORTEN`	The texture will be foreshortened in a perspectively correct manner.
`rwFILTER`	A filter will be applied to the texture to reduce the effect of pixelation due to aliasing.

For further information see the Texture Modes section in Chapter 2: Data Types.

See Also

RwAddTextureModeToPolygon()
RwAddTextureModeToSurface()
RwGetMaterialTextureModes()
RwSetMaterialLightSampling()
RwSetMaterialTexture()
RwSetMaterialTextureModes()
RwRemoveTextureModeFromMaterial()

```
RwPolygon3d *
RwAddTextureModeToPolygon(RwPolygon3d *polygon,
        RwTextureModes mode);
```

Description

Adds the given texture mode (or modes) to the polygons material. Texture modes permit fine grain control over the rendering of textures.

Arguments

`polygon`	Pointer to the polygon.
`Mode`	A bitfield representing a texture mode (or bitwise or of modes).

Return Value

The argument `polygon` if successful, and `NULL` otherwise.

Comments

The following texture modes are supported:

`rwLIT`	The texture will be lit according to the current light sampling type of the material (`rwFACET` or `rwVERTEX)`.
`rwFORESHORTEN`	The texture will be foreshortened in a perspectively correct manner.
`rwFILTER`	A filter will be applied to the texture to reduce the effect of pixelation due to aliasing.

For further information see the Texture Modes section in Chapter 2: Data Types.

See Also

RwAddTextureModeToMaterial()
RwAddTextureModeToSurface()
RwGetPolygonTextureModes()
RwSetPolygonLightSampling()
RwSetPolygonTexture()
RwSetPolygonTextureModes()
RwRemoveTextureModeFromPolygon()

`RwBool`
`RwAddTextureModeToSurface``(RwTextureModes mode);`

Description

Adds the given texture mode (or modes) to the current material. Texture modes permit fine grain control over the rendering of textures.

Arguments

`mode`	A bitfield representing a texture mode (or bitwise or of modes).

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

The following texture modes are supported:

`rwLIT`	The texture will be lit according to the current light sampling type of the material (`rwFACET` or `rwVERTEX`).
`rwFORESHORTEN`	The texture will be foreshortened in a perspectively correct manner.
`rwFILTER`	A filter will be applied to the texture to reduce the effect of pixelation due to aliasing.

For further information see the Texture Modes section in Chapter 2: Data Types.

This function can only be called within the context of an **`RwModelBegin`**`()` … **`RwModelEnd`**`()` block.

See Also

`RwAddTextureModeToMaterial``()`
`RwAddTextureModeToPolygon``()`
`RwModelBegin``()`
`RwModelEnd``()`
`RwSetSurfaceTexture``()`
`RwSetSurfaceTextureModes``()`
`RwRemoveTextureModeFromSurface``()`

```
RwClump *
RwAddUserDrawToClump(RwClump *clump, RwUserDraw *userdraw);
```

Description

Adds the user-draw to the clump. If the user-draw is already owned by another clump, it is first removed from that clump.

Arguments

`clump`	Pointer to the clump.
`userdraw`	Pointer to the user-draw.

Return Value

The argument `clump` if successful, and `NULL` otherwise.

See Also

RwCreateUserDraw()
RwDestroyClump()
RwDestroyUserDraw()
RwDuplicateUserDraw()
RwForAllUserDrawsInClump()
RwGetClumpNumUserDraws()
RwGetUserDrawOwner()
RwRemoveUserDrawFromClump()

```
RwV3d *
RwAddVector(RwV3d *a, RwV3d *b, RwV3d *c);
```

Description

Adds two vectors.

Arguments

`a`	Pointer to the first vector.
`b`	Pointer to the second vector.
`c`	Pointer to the vector that will receive the result.

Return Value

The argument `c` if successful, and `NULL` otherwise.

See Also

RwCrossProduct()
RwDotProduct()
RwNormalize()
RwScaleVector()
RwSubtractVector()
RwTransformVector()

RwInt32
RwAddVertexToClump(RwClump *clump, RwReal x, RwReal y, RwReal z);

Description

Adds a vertex to the clump.

Arguments

`clump`	Pointer to the clump.
`x`	X co-ordinate of the vertex (in object space co-ordinates).
`y`	Y co-ordinate of the vertex (in object space co-ordinates).
`z`	Z co-ordinate of the vertex (in object space co-ordinates).

Return Value

A positive integer representing the index of the vertex within the clump if successful, and `0` otherwise.

Comments

As this function modifies the geometry of the clump, the clump is made editable by this function (the `rwEDITABLE` hint is set).

The initial texture co-ordinates of the vertex are [`CREAL(0.5)`, `CREAL(0.5)`] and the initial unit shading normal is computed by RenderWare.

See Also

RwAddPolygonsToClump()
RwAddPolygonToClump()
RwGetClumpNumVertices()
RwSetClumpVertex()
RwSetClumpVertexUV()
RwSetClumpVertexNormal()
RwSetClumpVertices()

`RwCamera *`
`RwBeginCameraUpdate``(RwCamera *camera, void *param);`

Description

Makes `camera` the current camera (the camera used in subsequent rendering operations).

Arguments

`camera`	Pointer to the camera.
`param`	Device dependent parameter.

Return Value

The argument `camera` if successful, and `NULL` otherwise.

Comments

For a description of the device dependent parameter, `param`, see Appendix B.

`RwClearCameraViewport``()`, **`RwRenderClump`**`()`, and **`RwRenderScene`**`()` should only be called from within an **`RwBeginCameraUpdate`**`()` … **`RwEndCameraUpdate`**`()`.

See Also

`RwClearCameraViewport``()`
`RwEndCameraUpdate``()`
`RwRenderClump``()`
`RwRenderScene``()`
`RwShowCameraImage``()`

RwRaster *
RwBitmapRaster(void *bitmap, RwRasterOptions options);

Description

Converts a platform specific bitmap to a raster. The bitmap will be processed according to the specified options.

Arguments

`bitmap`	The source bitmap (device dependent parameter).
`options`	A bitfield representing a raster processing option (or bitwise or of options).

Return Value

A pointer to the new raster if successful, and `NULL` otherwise.

Comments

This function is useful for generating texture maps at run-time. An application can convert platform specific, 2D rendering into a raster with **RwBitmapRaster**(). The resultant raster can then be selected into a texture map with **RwSetTextureRaster**().

The supported raster options are as follows:

`rwAUTODITHERRASTER`	Dither the raster only if the source bitmap is to be resized (`rwFITRASTER` has been specified) or if the bitmap is a different depth from the current RenderWare render depth.
`rwDITHERRASTER`	Dither the raster.
`rwFITRASTER`	Resize the raster to texture map dimensions, i.e., 128 x `n` * 128 (where `n` is the number of frames in a multi-frame texture).
`rwGAMMARASTER`	Gamma correct the raster.

See Also

RwCreateRaster()
RwCreateTexture()
RwDestroyRaster()
RwDuplicateRaster()
RwGetCameraViewportRaster()
RwReadRaster()
RwReadMaskRaster()
RwSetTextureRaster()

`RwBool`
`RwBlock``(RwReal width, RwReal height, RwReal depth);`

Description

Adds a block, centered about the origin, to the current clump under construction. The block is transformed by the CTM, and the current material is applied to its polygons.

Arguments

`width`	Block width.
`height`	Block height.
`depth`	Block depth.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

It is an error if any of the blocks dimensions are degenerate, i.e., `CREAL(0.0)`.

This function can only be called within the context of an **`RwClumpBegin()`** … **`RwClumpEnd()`** or **`RwProtoBegin()`** … **`RwProtoEnd()`** block.

See Also

`RwClumpBegin()`
`RwClumpEnd()`
`RwCone()`
`RwCylinder()`
`RwDisc()`
`RwHemisphere()`
`RwProtoBegin()`
`RwProtoEnd()`
`RwSphere()`

`RwClump *`
`RwCalculateClumpVertexNormal``(RwClump *clump, RwInt32 index);`

Description

Activates automatic calculation of the unit shading normal at the vertex which belongs to clump and has the vertex index `index`. This ensures that the unit shading normal of the vertex will be recalculated every time the vertex is moved with **`RwSetClumpVertex`**`()` or the set of polygons sharing the vertex is modified.

Arguments

`clump`	Pointer to the clump.
`index`	The vertex index.

Return Value

The argument `clump` if successful, and `NULL` otherwise.

Comments

Automatic calculation is deactivated after a call to **`RwSetClumpVertexNormal`**`()` or if the vertex is created by calling **`RwVertexExt`**`()` with a non-`NULL` normal.

Unit shading normals are automatically recalculated by default.

See Also

`RwGetClumpVertexNormal``()`
`RwSetClumpVertex``()`
`RwSetClumpVertexNormal``()`
`RwVertexExt``()`

```
RwCamera *
RwClearCameraViewport(RwCamera *camera);
```

Description

Clears the cameras image buffer. If the camera does not have a backdrop raster the viewport will be cleared to the cameras background color. If the camera has a backdrop raster the cameras backdrop viewport rectangle will be filled with the backdrop raster. The remainder of the viewport will be cleared to the cameras background color.

Arguments

`camera` Pointer to the camera.

Return Value

The argument `camera` if successful, and `NULL` otherwise.

Comment

This function can only be called within the context of an **`RwBeginCameraUpdate()`** … **`RwEndCameraUpdate()`** block.

See Also

`RwBeginCameraUpdate()`
`RwDamageCameraViewport()`
`RwEndCameraUpdate()`
`RwInvalidateCameraViewport()`
`RwSetCameraBackColor()`
`RwSetCameraBackdrop()`
`RwSetCameraBackdropViewportRect()`
`RwUndamageCameraViewport()`

```
void
RwClose(void);
```

Description

Closes the RenderWare library.

Arguments

None.

Return Value

None.

Comments

This function must be called before the program exits.

RwClose() frees the following resources:

- the default scene and any clumps and lights contained in that scene.
- the transformation matrix stack.
- the matrix stack.
- all cameras.
- all named textures (and their rasters) and texture dictionaries.

RwClose() does not free the following resources:

- scenes other than the default scene.
- materials created with **RwCreateMaterial**() or **RwDuplicateMaterial**().
- matrices created with **RwCreateMatrix**() or **RwDuplicateMatrix**().
- unnamed textures and rasters (this includes camera backdrop rasters which must be destroyed explicitly).
- splines.
- user-draws not owned by clumps in the default scene.

See Also

RwCreateMaterial()
RwCreateMatrix()
RwDuplicateMaterial()
RwDuplicateMatrix()
RwOpen()
RwOpenExt()

```
void
RwCloseDebugStream(void);
```

Description

Closes the current debug stream.

Arguments

None.

Return Value

None.

Comments

No more debugging messages will be issued until a debugging stream is specified with **RwSetDebugStream**() or opened using **RwOpenDebugStream**().

See Also

RwOpenDebugStream()
RwSetDebugStream()

RwBool
RwClumpBegin(void);

Description

Identifies the beginning of a clump definition. The modeling matrix for the clump is set to the CTM at this time. The joint (articulation) matrix for the clump is set to the current joint transformation matrix.

The current transformation matrix, the current joint transformation matrix, and the current material are pushed onto the main transformation stack, the joint transformation stack, and the material stack respectively.

Arguments

None.

Return Value

TRUE if successful, and FALSE otherwise.

Comments

This function can only be called within the context of an **RwModelBegin**() … **RwModelEnd**() block. A call to **RwClumpBegin**() may be nested, to any depth, within an **RwProtoBegin**() … **RwProtoEnd**() or **RwClumpBegin**() … **RwClumpEnd**() block.

When nested within an **RwProtoBegin**() … **RwProtoEnd**() or **RwClumpBegin**() … **RwClumpEnd**() block, an **RwClumpBegin**() … **RwClumpEnd**() block creates a child clump.

If the nested **RwClumpBegin**() … **RwClumpEnd**() block is within an **RwProtoBegin**() … **RwProtoEnd**() block, then no clump is actually created, instead a child definition is added to the prototype under construction.

See Also

RwClumpEnd()
RwModelBegin()
RwModelEnd()
RwProtoBegin()
RwProtoEnd()

```
RwReal
RwClumpDistance(RwClump *clump, RwV3d *point);
```

Description

Calculates the distance from the origin of the clump to the point (in world space units).

Arguments

`clump`	Pointer to the clump.
`point`	Pointer to the point.

Return Value

The distance between the origin of the clump and the point in world space units if successful. Errors can be checked for using **`RwGetError`**`()`.

See Also

`RwGetClumpOrigin``()`

`RwClump *`
`RwClumpEnd``(RwClump **pointer);`

Description

Marks the end of the construction of clump and returns the newly created clump. The main transformation stack, the joint transformation stack and the material stack are restored to their state at the time of the matching **`RwClumpBegin`**`()`.

Arguments

`pointer` Pointer to the clump pointer that will receive the clump.

Return Value

A pointer to the new clump if successful, and `NULL` otherwise.

Comments

The function returns a pointer to the newly created clump. If a non-`NULL` argument is passed then pointer will also be set to point to the new clump.

When there are nested **`RwClumpEnd`**`()` calls, i.e., a hierarchical model is being built, `NULL` should be used as the argument in all **`RwClumpEnd`**`()` calls except the top-level one. Do not rely on the clump pointers returned by this function when creating child clumps.

See Also

`RwClumpBegin``()`
`RwModelBegin``()`
`RwModelEnd``()`
`RwProtoBegin``()`
`RwProtoEnd``()`

RwBool
RwCone(RwReal height, RwReal radius, RwInt32 nsides);

Description

Adds a cone to the current clump under construction. The cone is transformed by the CTM, and the current material is assigned to its polygons. The base of the cone lies on the X-Z plane, extending up the Y axis. The base is not closed.

Arguments

`height`	Cone height.
`radius`	Radius of the cone base.
`nsides`	Number of sides.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

It is an error if the cones radius is degenerate, i.e. `CREAL(0.0)`.

If a negative radius is specified, the polygons forming the cone will face inward.

This function can only be called within the context of an **RwClumpBegin**() … **RwClumpEnd**() or **RwProtoBegin**() … **RwProtoEnd**() block.

See Also

RwBlock()
RwClumpBegin()
RwClumpEnd()
RwCylinder()
RwDisc()
RwHemisphere()
RwProtoBegin()
RwProtoEnd()
RwSphere()

```
RwMaterial *
RwCopyMaterial(RwMaterial *source, RwMaterial *dest);
```

Description

Copies material `source` to material `dest`.

Arguments

`source`	Pointer to the source material.
`dest`	Pointer to the destination material.

Return Value

The argument `dest` if successful, and `NULL` otherwise.

See Also

RwCreateMaterial()
RwDestroyMaterial()
RwDuplicateMaterial()
RwPushCurrentMaterial()

```
RwMatrix4d *
RwCopyMatrix(RwMatrix4d *source, RwMatrix4d *dest);
```

Description

Copies matrix `source` to matrix `dest`.

Arguments

`source`	Pointer to the source matrix.
`dest`	Pointer to the destination matrix.

Return Value

The argument `dest` if successful, and `NULL` otherwise.

See Also

RwCreateMatrix()
RwDestroyMatrix()
RwDuplicateMatrix()
RwPushScratchMatrix()

```
RwCamera *
RwCreateCamera(RwInt32 maxwidth, RwInt32 maxheight, void *param);
```

Description

Creates a new camera.

Arguments

`maxwidth`	Maximum width of the camera viewport (in device space units).
`maxheight`	Maximum height of the camera viewport (in device space units).
`param`	Device dependent parameter.

Return Value

A pointer to the new camera if successful, and `NULL` otherwise.

Comments

For a description of the device dependent parameter, `param`, see Appendix B.

[illegible]

- The backdrop offset is (`0`, `0`).
- The backdrop viewport rectangle has a position of (`0`, `0`) and a size of `0` by `0`.

See Also

RwDestroyCamera()
RwDuplicateCamera()
RwResetCamera()

RwClump *
RwCreateClump(RwInt32 vcount, RwInt32 pcount);

Description

Creates a new, empty clump. The clump is added to the default scene.

Arguments

`vcount`	Initial number of vertices.
`pcount`	Initial number of polygons.

Return Value

A pointer to the new clump if successful, and `NULL` otherwise.

Comments

The arguments `vcount` and `pcount` are initial guidelines only, the actual number of polygons and vertices in a clump is not constrained by these initial values.

See Also

RwAddPolygonsToClump()
RwAddPolygonToClump()
RwAddVertexToClump()
RwClumpBegin()
RwClumpEnd()
RwCreateSprite()
RwDestroyClump()
RwDuplicateClump()
RwReadShape()

```
RwLight *
RwCreateLight(RwLightType type, RwReal x, RwReal y, RwReal z,
      RwReal intensity);
```

Description

Creates a new light. The light is added to the default scene.

Arguments

`type`	Type of light.
`x`	X component of the lights position in world space co-ordinates (for point and conical lights), or vector (for directional lights).
`y`	Y component of the lights position in world space co-ordinates (for point and conical lights), or vector (for directional lights).
`z`	Z component of the lights position in world space co-ordinates (for point and conical lights), or vector (for directional lights).
`intensity`	Intensity of the light in the range `CREAL(0.0)` to `CREAL(1.0)`.

Return Value

A pointer to the new light if successful, and `NULL` otherwise.

Comments

The default light state is `rwON`.

For conical lights, the default direction vector is down the negative Y axis, and the default cone angle is `CREAL(30.0)` degrees.

See Also

RwAddLightToScene()
RwDestroyLight()
RwDuplicateLight()

RwMaterial *
RwCreateMaterial(void);

Description

Creates a new material with default values for its attributes.

Arguments

None.

Return Value

A pointer to the newly created material if successful, and NULL otherwise.

Comments

By default:

- The materials coefficients of ambient, diffuse, and specular reflection are [illegible]
- [illegible] is rwSOLID.
- The materials only texture mode is rwLIT.
- The materials opacity is CREAL(1.0).

See Also

RwCurrentMaterial()
RwDestroyMaterial()
RwDuplicateMaterial()
RwPushCurrentMaterial()

`RwMatrix4d *`
`RwCreateMatrix``(void);`

Description

Creates a new transformation matrix.

Arguments

None.

Return Value

A pointer to the new matrix if successful, and `NULL` otherwise.

Comments

The new matrix is initialized to the identity matrix.

See Also

`RwScratchMatrix``()`
`RwDestroyMatrix``()`
`RwDuplicateMatrix``()`
`RwPushScratchMatrix``()`

```
RwRaster *
RwCreateRaster(RwInt32 width, RwInt32 height);
```

Description

Creates a new raster.

Arguments

`width`	Width of the raster (in pixels).
`height`	Height of the raster (in pixels).

Return Value

A pointer to the new raster if successful, and `NULL` otherwise.

Comments

The depth of the raster created is the same as the current render depth (the current render depth can be retrieved by **`RwGetDeviceInfo`**`()`).

The rasters pixels are not initialized by **`RwCreateRaster`**`()`. The initial pixel values are undefined.

See Also

`RwBitmapRaster``()`
`RwCreateTexture``()`
`RwDestroyRaster``()`
`RwDuplicateRaster``()`
`RwGetCameraViewportRaster``()`
`RwGetDeviceInfo``()`
`RwGetTextureRaster``()`
`RwReadRaster``()`
`RwReadMaskRaster``()`
`RwSetCameraBackdrop``()`
`RwSetTextureRaster``()`

RwScene *
RwCreateScene(void);

Description

Creates a new, empty scene.

Arguments

None.

Return Value

A pointer to the new scene if successful, and NULL otherwise.

See Also

RwAddClumpToScene()
RwAddLightToScene()
RwDefaultScene()
RwDestroyScene()

RwSpline *
RwCreateSpline(RwInt32 npoints, RwSplineType type, RwV3d *points);

Description

Creates a new spline.

Arguments

npoints	Number of control points (greater than or equal to 4).
type	Type of the spline.
points	Array of control points.

Return Value

A pointer to the new spline if successful, and NULL otherwise.

Comments

A minimum of 4 control points must be specified.

See Also

RwDestroySpline()
RwDuplicateSpline()

```
RwClump *
RwCreateSprite(RwTexture *texture);
```

Description

Creates a sprite. A sprite is a specialized form of clump which is used to display unlit textures constrained to be co-planar with the viewplane of a camera.

Arguments

`texture` Pointer to the texture.

Return Value

A pointer to the newly created sprite if successful, and `NULL` otherwise.

Comments

This function is a simplification function which creates a clump with a single, rectangular polygon, one unit in width and one unit in height, centered about the origin and lying in the X-Y plane. The given texture is made the current texture of the polygons material and the materials texture mode is set to `0`, i.e., the sprite is unlit, not foreshortened and unfiltered. The clumps axis alignment parameter is set to `rwALIGNAXISXYZ`.

The resulting clump may be manipulated in exactly the same way and using exactly the same API calls as clumps created by **RwReadShape()**, **RwClumpBegin()** … **RwClumpEnd()**, and **RwCreateClump()**. In particular, **RwDestroyClump()** should be used to destroy the clump created by **RwCreateSprite()** when it is no longer required.

See Also

RwAddPolygonToClump()
RwAddVertexToClump()
RwClumpBegin()
RwClumpEnd()
RwCreateClump()
RwDestroyClump()
RwFindNamedTexture()
RwGetNamedTexture()
RwReadNamedTexture()
RwReadShape()
RwReadTexture()
RwSetClumpAxisAlignment()
RwSetPolygonTexture()
RwSetPolygonTextureModes()

`RwTexture *`
`RwCreateTexture``(RwRaster *raster);`

Description

Creates a new texture and sets its raster to `raster`.

Arguments

`raster` Pointer to the raster.

Return Value

A pointer to the newly created texture if successful, and `NULL` otherwise.

Comments

The specified raster must have a width of 128 pixels and a height of 128 pixels (or `n` * 128 pixels for multi-frame textures where `n` is the number of frames). The raster will not be resized if it not already of the correct size.

Rasters cannot be shared between textures. It is an error to specify a raster already selected into a texture.

See Also

`RwBitmapRaster``()`
`RwCreateRaster``()`
`RwDestroyTexture``()`
`RwFindNamedTexture``()`
`RwGetCameraViewportRaster``()`
`RwGetNamedTexture``()`
`RwMaskTexture``()`
`RwReadMaskRaster``()`
`RwReadNamedTexture``()`
`RwReadRaster``()`
`RwReadTexture``()`
`RwSetTextureRaster``()`

```
RwUserDraw *
RwCreateUserDraw(RwUserDrawType type,
        RwUserDrawAlignmentTypes alignment,
        RwInt32 x, RwInt32 y, RwInt32 width, RwInt32 height,
        void (*callback)(RwUserDraw *userdraw,
        void *camimage, RwRect *rect, void *data));
```

Description

Creates a user-draw.

Arguments

`type`	Type of user-draw to create.
`alignment`	A bitfield representing an alignment type (or bitwise or of alignment types).
`x`	X offset of the user-draw from the alignment point (in viewport space units).
`y`	Y offset of the user-draw from the alignment point (in viewport space units).
`width`	Width of the user-draw (in viewport space units).
`height`	Height of the user-draw (in viewport space units).
`callback`	Pointer to the call-back function that will render the user-draw.

Return Value

A pointer to the new user-draw if successful, and `NULL` otherwise.

Comments

The type of the user-draw determines whether it is aligned with a clumps origin (`rwCLUMPALIGN`), with a clumps vertex (`rwVERTEXALIGN`), with a clumps bounding box in viewport space (`rwBBOXALIGN`), or a cameras viewport (`rwVPALIGN`).

The following alignment flags are supported: `rwALIGNTOP`, `rwALIGNBOTTOM`, `rwALIGNLEFT` and `rwALIGNRIGHT`. For convenience, two common combinations of these flags, `rwALIGNTOPLEFT` and `rwALIGNBOTTOMRIGHT` are also defined.

Assuming that the type of the user-draw is `rwVERTEXALIGN`, then the interpretations of the different valid values for alignment are as follows:

`0`	The center of the user-draw is aligned with the vertex.
`rwALIGNTOP`	The midpoint of the top edge of the user-draw rectangle is aligned with the vertex.
`rwALIGNBOTTOM`	The midpoint of the bottom edge of the user-draw rectangle is aligned with the vertex.
`rwALIGNLEFT`	The midpoint of the left edge of the user-draw rectangle is aligned with the vertex.
`rwALIGNRIGHT`	The midpoint of the right edge of the user-draw rectangle is aligned with the vertex.
`RwALIGNTOP` | `rwALIGNLEFT`	The top left corner of the user-draw rectangle is aligned with the vertex.

`RwALIGNTOP | rwALIGNRIGHT`
: The top right corner of the user-draw rectangle is aligned with the vertex.

`RwALIGNBOTTOM | rwALIGNLEFT`
: The bottom left corner of the user-draw rectangle is aligned with the vertex.

`RwALIGNBOTTOM | rwALIGNRIGHT`
: The bottom right corner of the user-draw rectangle is aligned with the vertex.

A user-draw is positioned at an offset `(x, y)` from the point of alignment and its size is specified by `width` and `height`.

User-draw call-backs should be declared as follows:

```
void callback(RwUserDraw *userdraw, void *camimage,
              RwRect *rect, void *data);
```

Where the call-backs arguments are as follows:

`userdraw` Pointer to the user-draw to be rendered.

`camimage` The cameras image buffer as returned by **RwGetCameraImage()** for the current camera. camimage is device dependent. For more information, see Appendix B.

`rect` Pointer to a rectangle defining the area of the cameras image buffer into which the call-back may render. This rectangle is specified in viewport space co-ordinates, i.e., (0, 0) is the origin of the viewport.

`data` Pointer to the user data of the user-draw being drawn. This value can be obtained by calling **RwGetUserDrawData()** with userdraw as an argument. data is passed directly to the call-back function for the convenience of the application developer.

Note that the call-back function is always called after all clumps in the scene have been rendered, i.e., when **RwEndCameraUpdate()** is called. Therefore user-draw rendering always appear in front of clump rendering. In the case of overlapping user-draws, the order of rendering is not defined.

See Also

RwAddUserDrawToClump()
RwDestroyClump()
RwDestroyUserDraw()
RwDuplicateUserDraw()
RwEndCameraUpdate()
RwGetCameraImage()
RwGetUserDrawData()

`RwV3d *`
`RwCrossProduct``(RwV3d *a, RwV3d *b, RwV3d *c);`

Description

Calculates the cross product of two vectors.

Arguments

`a`	Pointer to the left vector.
`b`	Pointer to the right vector.
`c`	Pointer to the vector that will receive the result.

Return Value

The argument `c` if successful, and `NULL` otherwise.

Comments

`c` must not point to the same vector as either of the other arguments.

See Also

`RwAddVector``()`
`RwDotProduct``()`
`RwNormalize``()`
`RwScaleVector``()`
`RwSubtractVector``()`
`RwTransformVector``()`

`RwClump *`
`RwCubicTexturizeClump``(RwClump *clump);`

Description

Sets the texture co-ordinates for every polygon belonging to the clump using the cubic projection method.

A cubic mapping results in the construction of a nominal cube which has the texture applied to each of the cubes six facets. The resulting cube (with a copy of the texture applied to each face) is then mapped to the clump by shrink wrapping the clump with the cube.

Arguments

`clump` Pointer to the clump.

Return Value

The argument `clump` if successful, and `NULL` otherwise.

Comments

This function need only be called once, the first time a clump is textured, and not each time the clump is rendered.

Note that this function does not set the textures associated with the clumps polygons; this must be accomplished separately. The following code fragment illustrates this procedure:

```
RwForAllPolygonsInClumpPointer(clump,  (RwPolygon3d*(*)
())RwSetPolygonTexture, texture);
```

See Also

`RwEnvMapClump``()`
`RwForAllPolygonsInClump``()`
`RwSetClumpVertexUV``()`
`RwSetPolygonTexture``()`
`RwSetPolygonUV``()`
`RwSphericalTexturizeClump``()`
`RwVertexExt``()`

RwMaterial *

RwCurrentMaterial(void);

Description

Retrieves the current material.

Arguments

None.

Return Value

A pointer to the current material if successful, and NULL otherwise.

Comments

The material returned by **RwCurrentMaterial**() must not be destroyed with **RwDestroyMaterial**(). The material stack is destroyed by RenderWare when **RwClose**() is called.

See Also

RwClose()
RwCreateMaterial()
RwDestroyMaterial()
RwPopCurrentMaterial()
RwPushCurrentMaterial()

```
RwBool
RwCylinder(RwReal height, RwReal baserad, RwReal toprad,
    RwInt32 nsides);
```

Description

Adds a cylinder to the current clump under construction. The cylinder is transformed by the CTM, and the current material is assigned to its polygons. The base of the cylinder lies on the X-Z plane, extending up the Y axis.

Arguments

`height`	Cylinder height.
`baserad`	Radius of the cylinder base.
`toprad`	Radius of the cylinder top.
`nsides`	Number of sides.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

It is an error if the cylinders radius is degenerate, i.e., `CREAL(0.0)`.

Note that if both `baserad` and `toprad` are negative the polygons forming the cylinder will face inward. It is an error if one of the radii is negative and the other is positive.

This function can only be called within the context of an **RwClumpBegin**`()` … **RwClumpEnd**`()` or **RwProtoBegin**`()` … **RwProtoEnd**`()` block.

See Also

RwBlock`()`
RwClumpBegin`()`
RwClumpEnd`()`
RwCone`()`
RwDisc`()`
RwHemisphere`()`
RwSphere`()`
RwProtoBegin`()`
RwProtoEnd`()`

```
RwCamera *
RwDamageCameraViewport(RwCamera *camera, RwInt32 x, RwInt32 y,
       RwInt32 width, RwInt32 height);
```

Description

Damages a rectangular area of the cameras viewport. The rectangle is added to the area to be updated by **RwShowCameraImage**() and cleared by **RwClearCameraViewport**().

Arguments

`camera`	Pointer to the camera.
`x`	X co-ordinate of the rectangles top left corner (in viewport space co-ordinates).
`y`	Y co-ordinate of the rectangles top left corner (in viewport space co-ordinates).
`width`	Width of the rectangle (in viewport space units).
`height`	Height of the rectangle (in viewport space units).

Return Value

The argument `camera` if successful, and `NULL` otherwise.

See Also

RwBeginCameraUpdate()
RwClearCameraViewport()
RwEndCameraUpdate()
RwInvalidateCameraViewport()
RwShowCameraImage()
RwUndamageCameraViewport()

RwScene *
RwDefaultScene(void);

Description

Retrieves the default scene.

Arguments

None.

Return Value

A pointer to the default scene.

Comments

The scene returned by **RwDefaultScene**() must not be destroyed with **RwDestroyScene**(). The default scene is destroyed by RenderWare when **RwClose**() is called.

See Also

RwClose()
RwClumpEnd()
RwCreateClump()
RwCreateLight()
RwCreateScene()
RwCreateSprite()
RwDestroyScene()
RwReadShape()
RwRemoveClumpFromScene()
RwRemoveLightFromScene()

RwBool
RwDestroyCamera(RwCamera *camera);

Description

Destroys the camera.

Arguments

camera	Pointer to the camera.

Return Value

TRUE if successful, and FALSE otherwise.

Comments

All cameras not explicitly destroyed are automatically destroyed by **RwClose**().

This function does not destroy the device dependent object specified in the call to **RwCreateCamera**(). Furthermore, the cameras backdrop raster (if any) is not destroyed.

See Also

RwClose()
RwCreateCamera()
RwDuplicateCamera()
RwSetCameraBackdrop()

RwBool
RwDestroyClump(RwClump *clump);

Description

Destroys the clump.

Arguments

clump	Pointer to the clump.

Return Value

TRUE if successful, and FALSE otherwise.

Comments

Note that this function is recursive - it destroys the clump and all its descendants (if any). Furthermore, any user-draw objects added to this clump will also be destroyed by **RwDestroyClump**().

See Also

RwAddUserDrawToClump()
RwClumpEnd()
RwCreateClump()
RwCreateSprite()
RwDestroyScene()
RwDuplicateClump()
RwReadShape()

```
RwBool
RwDestroyLight(RwLight *light);
```

Description

Destroys the light.

Arguments

`light`	Pointer to the light.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

See Also

`RwCreateLight()`
`RwDestroyScene()`
`RwDuplicateLight()`
`RwRemoveLightFromScene()`

`RwBool`
`RwDestroyMaterial``(RwMaterial *material);`

Description

Destroys the material.

Arguments

`material` Pointer to the material.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

This function must not be used to destroy a polygons material (as obtained by a call to **`RwGetPolygonMaterial`**`()`), or a material from the material stack (as obtained by **`RwCurrentMaterial`**`()`, **`RwPopCurrentMaterial`**`()` or **`RwPushCurrentMaterial`**`()`).

See Also

`RwCreateMaterial``()`
`RwDuplicateMaterial``()`
`RwGetPolygonMaterial``()`
`RwPopCurrentMaterial``()`
`RwPushCurrentMaterial``()`

`RwBool`
`RwDestroyMatrix``(RwMatrix4d *matrix);`

Description

Destroys the matrix.

Arguments

`matrix`	Pointer to the matrix.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

This function must not be used to destroy a matrix from the scratch matrix stack (as obtained by **`RwScratchMatrix`**`()`, **`RwPopScratchMatrix`**`()` or **`RwPushScratchMatrix`**`()`).

See Also

`RwCreateMatrix``()`
`RwScratchMatrix``()`
`RwDuplicateMatrix``()`
`RwPopScratchMatrix``()`
`RwPushScratchMatrix``()`

`RwBool`
`RwDestroyPolygon``(RwPolygon3d *polygon);`

Description

Destroys the polygon.

Arguments

`polygon` Pointer to the polygon.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

See Also

RwAddPolygonsToClump()
RwAddPolygonToClump()
RwDestroyClump()

```
RwBool
RwDestroyRaster(RwRaster *raster);
```

Description

Destroys the raster.

Arguments

`raster` Pointer to the raster.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

It is an error to attempt to destroy a textures raster. **RwDestroyTexture()** destroys both a texture and its raster. This also applies to rasters selected into texture by **RwCreateTexture()** or **RwSetTextureRaster()**.

RwDestroyCamera() does not destroy a cameras backdrop raster. The backdrop raster should be destroyed by **RwDestroyRaster()**.

See Also

RwBitmapRaster()
RwCreateRaster()
RwCreateTexture()
RwDestroyCamera()
RwDestroyTexture()
RwDuplicateRaster()
RwGetCameraViewportRaster()
RwGetTextureRaster()
RwReadRaster()
RwReadMaskRaster()
RwSetCameraBackdrop()
RwSetTextureRaster()

```
RwBool
RwDestroyScene(RwScene *scene);
```

Description

Destroys the scene and all its clumps and lights.

Arguments

`scene` Pointer to the scene.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

To prevent a clump or light from being destroyed, use **RwRemoveClumpFromScene**() or **RwRemoveLightFromScene**() before calling **RwDestroyScene**().

The default scene cannot be destroyed.

See Also

RwCreateScene()
RwDefaultScene()
RwDestroyClump()
RwDestroyLight()
RwRemoveClumpFromScene()
RwRemoveLightFromScene()

```
RwBool
RwDestroySpline(RwSpline *spline);
```

Description

Destroys the spline.

Arguments

`spline`	Pointer to the spline.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

See Also

RwCreateSpline()
RwDuplicateSpline()

```
RwBool
RwDestroyTexture(RwTexture *texture);
```

Description

Destroys the texture (and its raster).

Arguments

`texture` Pointer to the texture.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

Textures which are still referenced by materials must not be destroyed. Remove the references to a texture with **`RwSetMaterialTexture`**`()` or **`RwSetPolygonTexture`**`()` before destroying the texture.

If the texture is defined in a dictionary, this function removes it from that dictionary.

See Also

`RwCreateTexture``()`
`RwFindNamedTexture``()`
`RwGetNamedTexture``()`
`RwReadNamedTexture``()`
`RwReadShape``()`
`RwReadTexture``()`
`RwSetMaterialTexture``()`
`RwSetPolygonTexture``()`
`RwSetTextureRaster``()`
`RwTextureDictEnd``()`

`RwBool`
`RwDestroyUserDraw``(RwUserDraw *userdraw);`

Description

Destroys the user-draw.

Arguments

`userdraw` Pointer to the user-draw.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

If `userdraw` is owned by a clump it will be removed from the clump prior to being destroyed.

Note that **`RwDestroyClump`**`()` destroys any user-draw objects that belong to the clump being destroyed.

See Also

`RwAddUserDrawToClump``()`
`RwCreateUserDraw``()`
`RwDestroyClump``()`
`RwDuplicateUserDraw``()`
`RwRemoveUserDrawFromClump``()`

```
RwInt32
RwDeviceControl(RwDeviceAction action, RwInt32 param1,
        void *param2, RwInt32 size);
```

Description

Performs low-level, device dependent actions.

Arguments

`action`	Device dependent action.
`param1`	First action specific parameter.
`param2`	Second action specific parameter.
`size`	Size in bytes of the buffer (if any) pointed to by `param2`.

Return Value

The return value is dependent on the device dependent action being performed.

Comments

The `size` parameter is new with RenderWare V1.4. `size` gives the size in bytes of the buffer pointed to by the second action specific parameter, `param2`. For example, to control the stretching of rendering under Microsoft Windows the following device control would be used:

```
RwWinOutputSize winOutputSize;
winOutputSize.width = 640;
winOutputSize.height = 480;
winOutputSize.camera = Camera;
RwDeviceControl(rwWINSETOUTPUTSIZE, 0L, &winOutputSize,
                    sizeof(winOutputSize));
```

If `param2` is `NULL` `size` is ignored.

The supported actions and their associated parameter values are device dependent. See Appendix B for details.

```
RwBool
RwDisc(RwReal height, RwReal radius, RwInt32 nsides);
```

Description

Adds a disc to the current clump under construction. The disc is transformed by the CTM, and the current material is assigned to its polygons. The disc lies on the Y = `height` plane, centered about the Y axis. This function is primarily used to cap cones and cylinders.

Arguments

`height`	Disc plane.
`radius`	Radius of the disc.
`nsides`	Number of sides.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

It is an error if the discs radius is degenerate, i.e., `CREAL(0.0)`.

Note that it is possible for the argument radius to have a negative value. In which case, the polygons forming the disc will be reversed.

This function can only be called within the context of an **`RwClumpBegin`**`()` … **`RwClumpEnd`**`()` or **`RwProtoBegin`**`()` … **`RwProtoEnd`**`()` block.

See Also

`RwBlock``()`
`RwClumpBegin``()`
`RwClumpEnd``()`
`RwCone``()`
`RwCylinder``()`
`RwHemisphere``()`
`RwProtoBegin``()`
`RwProtoEnd``()`
`RwSphere``()`

```
RwReal
RwDotProduct(RwV3d *a, RwV3d *b);
```

Description

Calculates the dot product of two vectors.

Arguments

`a`	Pointer to the left vector.
`b`	Pointer to the right vector.

Return Value

The dot product. Errors can be checked for using **`RwGetError`**`()`.

Comments

The answer is effectively meaningful only when the vectors are normalized (unit length).

See Also

`RwAddVector``()`
`RwCrossProduct``()`
`RwGetError``()`
`RwNormalize``()`
`RwScaleVector``()`
`RwSubtractVector``()`
`RwTransformVector``()`

```
RwCamera *
RwDuplicateCamera(RwCamera *camera, void *param);
```

Description

Creates a new camera with the same attributes as `camera`.

Arguments

`camera`	Pointer to the camera.
`param`	Device dependent parameter.

Return Value

A pointer to the new camera if successful, and `NULL` otherwise.

Comments

For a description of the device dependent parameter, `param`, see Appendix B.

If `camera` has a backdrop raster the raster will not be duplicated. The new camera will share the raster with `camera`.

See Also

RwCreateCamera()
RwDestroyCamera()

```
RwClump *
RwDuplicateClump(RwClump *clump);
```

Description

Creates a new clump with the same attributes as `clump`. The new clump is added to the same scene as `clump`.

Arguments

`clump` Pointer to the clump.

Return Value

A pointer to the new clump if successful, and `NULL` otherwise.

Comments

Note that this function is recursive - it copies the clump and all its descendants (if any).

See Also

RwAddClumpToScene()
RwClumpBegin()
RwClumpEnd()
RwCreateClump()
RwCreateSprite()
RwDestroyClump()
RwReadShape()

```
RwLight *
RwDuplicateLight(RwLight *light);
```

Description

Creates a new light with the same attributes as `light`. The new light is added to the same scene as `light`.

Arguments

`light` Pointer to the light.

Return Value

A pointer to the new light if successful, and `NULL` otherwise.

See Also

RwAddLightToScene()
RwCreateLight()
RwDestroyLight()

```
RwMaterial *
RwDuplicateMaterial(RwMaterial *material);
```

Description

Creates a new material with the same attributes as `material`.

Arguments

`material` Pointer to the material.

Return Value

A pointer to the new material if successful, and `NULL` otherwise.

See Also

RwCreateMaterial()
RwDestroyMaterial()

```
RwMatrix4d *
RwDuplicateMatrix(RwMatrix4d *matrix);
```

Description

Creates a new matrix with the same elements as `matrix`.

Arguments

`matrix` Pointer to the matrix to duplicate.

Return Value

Pointer to the new matrix if successful, and `NULL` otherwise.

See Also

RwCreateMatrix()
RwDestroyMatrix()

```
RwRaster *
RwDuplicateRaster(RwRaster *raster);
```

Description

Creates a new raster with the same attributes as `raster`. The pixels of `raster` are copied to the new raster.

Arguments

`raster` Pointer to the raster.

Return Value

Pointer to the new raster if successful, and `NULL` otherwise.

See Also

RwBitmapRaster()
RwCreateRaster()
RwDestroyRaster()
RwGetCameraViewportRaster()
RwReadRaster()
RwReadMaskRaster()

```
RwSpline *
RwDuplicateSpline(RwSpline *spline);
```

Description

Creates a new spline with the same attributes as `spline`.

Arguments

`spline`	Pointer to the spline.

Return Value

A pointer to the new spline if successful, and `NULL` otherwise.

See Also

RwCreateSpline()
RwDestroySpline()

`RwUserDraw *`
`RwDuplicateUserDraw``(RwUserDraw *userdraw);`

Description

Creates a new user-draw with the same attributes as `userdraw`.

Arguments

`userdraw` Pointer to the user-draw to be duplicated.

Return Value

A pointer to the new user-draw if successful, and `NULL` otherwise.

Comments

The new user-draw is owned by the same clump as `userdraw`, if `userdraw` has an owning clump, otherwise it will not be owned by a clump and should be added to a clump with **`RwAddUserDrawToClump`**`()`.

See Also

`RwAddUserDrawToClump``()`
`RwCreateUserDraw``()`
`RwDestroyUserDraw``()`

```
RwCamera *
RwEndCameraUpdate(RwCamera *camera);
```

Description

Performs all necessary housekeeping activities after rendering into the cameras image buffer is complete.

Arguments

`camera` Pointer to the camera.

Return Value

The argument `camera` if successful, and `NULL` otherwise.

Comments

`RwClearCameraViewport``()`, **`RwRenderClump`**`()` and **`RwRenderScene`**`()` should only be called from within an **`RwBeginCameraUpdate`**`()` … **`RwEndCameraUpdate`**`()`. Upon exit from this block, rendering to the specified cameras image buffer is complete. **`RwShowCameraImage`**`()` can then be used to update the hosts display.

See Also

`RwBeginCameraUpdate``()`
`RwClearCameraViewport``()`
`RwRenderClump``()`
`RwRenderScene``()`
`RwShowCameraImage``()`

```
RwClump *
RwEnvMapClump(RwClump *clump);
```

Description

Performs a view dependent projection of an environment map onto a clump.

Arguments

`clump`	Pointer to the clump.

Return Value

The argument `clump` if successful, and `NULL` otherwise.

Comments

To ensure the accuracy of the map, call this function each time the clump is transformed or the viewing camera is moved.

The environment map must have been previously assigned to that clump using;

```
RwForAllPolygonsInClumpPointer(clump, (RwPolygon3d*(*)
())RwSetPolygonTexture, texture);
```

See Also

RwCubicTexturizeClump()
RwForAllPolygonsInClump()
RwSetClumpVertexUV()
RwSetPolygonTexture()
RwSetPolygonUV()
RwSphericalTexturizeClump()

```
RwClump *
RwFindClump(RwClump *root, RwInt32 (*func)(RwClump *clump));
```

```
RwClump *
RwFindClumpInt(RwClump *root,
     RwBool (*func)(RwClump *clump, RwInt32 arg), RwInt32 arg);
```

```
RwClump *
RwFindClumpLong(RwClump *root,
     RwBool (*func)(RwClump *clump, RwInt32 arg), RwInt32 arg);
```

```
RwClump *
RwFindClumpReal(RwClump *root,
     RwBool (*func)(RwClump *clump, RwReal arg), RwReal arg);
```

```
RwClump *
RwFindClumpPointer(RwClump *root,
    RwBool (*func)(RwClump *clump, void *arg), void *arg);
```

Description

Finds a particular clump in a hierarchy by applying a boolean call-back function to each clump in the hierarchy in turn. If any invocation of the call-back function returns `TRUE`, iteration is terminated and the clump passed as the argument to the call-back function is returned.

The call-back function can either be a RenderWare API function or user-defined. It is important to note that a return value of `TRUE` indicates success (i.e., the clump being sought was found) and stops iteration, while a return value of `FALSE` indicates that the search should continue. If the search fails (i.e., no predicate returns `TRUE`), `NULL` is returned.

The difference between **RwFindClump()** and its variations listed above is that for **RwFindClump()** the call-back function takes only one argument (a clump pointer), whereas in the case of its variations, the call-back function takes an additional, user-supplied argument (`arg`) that can be of type `RwInt32`, `RwReal` or `void *` respectively.

Arguments

`root`	Pointer to the root clump.
`func`	Pointer to the call-back function.
`arg`	A user supplied argument to be passed to the call-back function.

Return Value

A pointer to the clump found if the search was successful, and `NULL` if the search failed or if any errors occurred. Errors can be checked for using **RwGetError()**.

Comments

The traversal of the clump hierarchy is done in a depth-first manner.

Note: **RwFindClumpLong()** now has identical functionality to **RwFindClumpInt()**. It is retained in this release for backward compatibility but will be removed from a future release of RenderWare. New applications should use **RwFindClumpInt()**.

See Also

RwAddChildToClump()
RwFindTaggedClump()
RwForAllClumpsInHierarchy()
RwGetError()
RwGetFirstChildClump()
RwGetNextClump()
RwRemoveChildFromClump()

RwTexture *
RwFindNamedTexture(char *name);

Description

Searches for a texture with the name `name`. If the current search mode is `rwLOCAL`, only the current dictionary is searched. If the current search mode is `rwGLOBAL`, the entire texture dictionary stack is searched. In the latter case, the search starts with the current texture dictionary, i.e., the top element of the stack, and proceeds downwards until a texture with the specified name is found or there are no more dictionaries to be examined.

Arguments

`name`	The texture name.

Return Value

A pointer to a texture. If there is an error or if no texture with that name is found, `NULL` is returned. Errors can be checked for using **RwGetError**().

Comments

This function only searches the current dictionary or dictionary stack. It will not attempt to load a texture from disk.

See Also

RwCreateTexture()
RwDestroyTexture()
RwForAllNamedTextures()
RwGetError()
RwGetNamedTexture()
RwReadNamedTexture()
RwReadTexture()
RwSetMaterialTexture()
RwSetPolygonTexture()
RwSetTextureDictSearchMode()
RwTextureDictBegin()
RwTextureDictEnd()

```
RwClump *
RwFindTaggedClump(RwClump *clump, RwInt32 tag);
```

Description

Looks for the clump with the specified tag in the hierarchy rooted at `clump`.

Arguments

`clump`	Pointer to the clump.
`tag`	Integer tag to find.

Return Value

A pointer to the clump found if the search was successful, and `NULL` if the search failed or if any errors occurred. Errors can be checked for using **RwGetError**().

See Also

RwAddChildToClump()
RwFindClump()
RwFindTaggedPolygon()
RwForAllClumpsInHierarchy()
RwGetClumpTag()
RwGetFirstChildClump()
RwGetNextClump()
RwRemoveChildFromClump()
RwSetClumpTag()
RwSetTag()

`RwPolygon3d *`
`RwFindTaggedPolygon``(RwClump *clump, RwInt32 tag);`

Description

Looks for the polygon with the specified tag in the polygon list of `clump`.

Arguments

`clump`	Pointer to the clump.
`tag`	Integer tag to find (only the 16 least significant bits are valid).

Return Value

A pointer to the polygon found if the search was successful, and `NULL` if the search failed or if any errors occurred. Errors can be checked for using **`RwGetError`**`()`.

See Also

`RwForAllPolygonsInClump``()`
`RwFindTaggedClump``()`
`RwGetError``()`
`RwGetPolygonTag``()`
`RwPolygonExt``()`
`RwQuadExt``()`
`RwSetPolygonTag``()`
`RwTriangleExt``()`

```
RwClump *
RwForAllClumpsInHierarchy(RwClump *root,
     RwClump *(*func) (RwClump *clump));
```

```
RwClump *
RwForAllClumpsInHierarchyInt(RwClump *root,
     RwClump *(*func)(RwClump *clump, RwInt32 arg), RwInt32 arg);
```

```
RwClump *
RwForAllClumpsInHierarchyLong(RwClump *root,
    RwClump *(*func)(RwClump *clump, RwInt32 arg), RwInt32 arg);
```

```
RwClump *
RwForAllClumpsInHierarchyReal(RwClump *root,
     RwClump *(*func)(RwClump *clump, RwReal arg), RwReal arg);
```

`RwClump *`
`RwForAllClumpsInHierarchyPointer``(RwClump *root,`
`RwClump *(*func)(RwClump *clump, void *arg), void *arg);`

Description

Applies a call-back function to all clumps in the hierarchy whose root is pointed to by `root`. If any invocation of the call-back function sets RenderWares error status, iteration is terminated. The call-back function can either be a RenderWare API function or a user-defined function. In the latter case, the call-back function should call **`RwSetUserError`**`()` if it fails for any reason.

The difference between **`RwForAllClumpsInHierarchy`**`()` and its variations listed above is that for **`RwForAllClumpsInHierarchy`**`()` the call-back function takes only one argument (a clump pointer), whereas in the case of its variations, the call-back function takes an additional, user-supplied argument (`arg`) that can be of type `RwInt32`, `RwReal` or `void *` respectively.

Arguments

`root`	Pointer to the root clump.
`func`	Pointer to the call-back function.
`arg`	A user-supplied argument to be passed to the call-back function.

Return Value

The argument `clump` if successful, and `NULL` otherwise.

Comments

If the return type of the call-back function is not `RwClump` *, then the pointer to the call-back function should be cast to the expected type, i.e., a pointer to a function whose return type is `RwClump` *. For example, in the case of a call-back function named `foo` whose return type is int, the following C expression should be used:

```
(RwClump*(*)())foo
```

The traversal of the clump hierarchy is done in a depth-first manner.

Note: **`RwForAllClumpsInHierarchyLong`**`()` now has identical functionality to **`RwForAllClumpsInHierarchyInt`**`()`. It is retained in this release for backward compatibility but will be removed from a future release of RenderWare. New applications should use **`RwForAllClumpsInHierarchyInt`**`()`.

See Also

`RwAddChildToClump``()`
`RwFindClump``()`
`RwFindTaggedClump``()`
`RwGetFirstChildClump``()`
`RwGetNextClump``()`
`RwRemoveChildFromClump``()`
`RwSetUserError``()`

```
RwScene *
RwForAllClumpsInScene(RwScene *scene,
     RwClump *(*func) (RwClump *clump));
```

```
RwScene *
RwForAllClumpsInSceneInt(RwScene *scene,
    RwClump *(*func) (RwClump *clump, RwInt32 arg), RwInt32 arg);
```

```
RwScene *
RwForAllClumpsInSceneLong(RwScene *scene,
     RwClump *(*func) (RwClump *clump, RwInt32 arg), RwInt32 arg);
```

```
RwScene *
RwForAllClumpsInSceneReal(RwScene *scene,
     RwClump *(*func) (RwClump *clump, RwReal arg), RwReal arg);
```

```
RwScene *
RwForAllClumpsInScenePointer(RwScene *scene,
    RwClump *(*func) (RwClump *clump, void *arg), void *arg);
```

Description

Applies a call-back function to all clumps in the scene. If any invocation of the call-back function sets RenderWares error status, iteration is terminated. The call-back function can either be a RenderWare API function or a user-defined function. In the latter case, the call-back function should call **RwSetUserError()** if it fails for any reason.

The difference between **RwForAllClumpsInScene()** and its variations listed above is that for **RwForAllClumpsInScene()** the call-back function takes only one argument (a clump pointer), whereas in the case of its variations, the call-back function takes an additional, user-supplied argument (`arg`) that can be of type `RwInt32`, `RwReal` or `void *` respectively.

Arguments

`scene`	Pointer to the scene.
`func`	Pointer to the call-back function.
`arg`	A user-supplied argument to be passed to the call-back function

Return Value

The argument `scene` if successful, and `NULL` otherwise.

Comments

If the return type of the call-back function is not `RwClump` *, then the pointer to the call-back function should be cast to the expected type, i.e., a pointer to a function whose return type is `RwClump` *. For example, in the case of a call-back function named foo whose return type is int, the following C expression should be used:

```
(RwClump*(*)())foo
```

Note: **RwForAllClumpsInSceneLong()** now has identical functionality to **RwForAllClumpsInSceneInt()**. It is retained in this release for backward compatibility but will be removed from a future release of RenderWare. New applications should use **RwForAllClumpsInSceneInt()**.

See Also

RwAddClumpToScene()
RwClumpBegin()
RwClumpEnd()
RwCreateClump()
RwForAllClumpsInHierarchy()
RwForAllLightsInScene()
RwGetSceneNumClumps()
RwReadShape()
RwRemoveClumpFromScene()
RwSetUserError()

```
RwScene *
RwForAllLightsInScene(RwScene *scene,
     RwLight *(*func) (RwLight *light));
```

```
RwScene *
RwForAllLightsInSceneInt(RwScene *scene,
     RwLight *(*func) (RwLight *light, RwInt32 arg), RwInt32 arg);
```

```
RwScene *
RwForAllLightsInSceneLong(RwScene *scene,
     RwLight *(*func) (RwLight *light, RwInt32 arg), RwInt32 arg);
```

```
RwScene *
RwForAllLightsInSceneReal(RwScene *scene,
     RwLight *(*func) (RwLight *light, RwReal arg), RwReal arg);
```

```
RwScene *
RwForAllLightsInScenePointer(RwScene *scene,
    RwLight *(*func)(RwLight *light, void *arg), void *arg);
```

Description

Applies a call-back function to all lights in the scene. If any invocation of the call-back function sets RenderWares error status, iteration is terminated. The call-back function can either be a RenderWare API function or a user-defined function. In the latter case, the call-back function should call **RwSetUserError**() if it fails for any reason.

The difference between **RwForAllLightsInScene**() and its variations listed above is that for **RwForAllLightsInScene**() the call-back function takes only one argument (a light pointer), whereas in the case of its variations, the call-back function takes an additional, user-supplied argument (`arg`) that can be of type `RwInt32`, `RwReal` or `void *` respectively.

Arguments

`scene`	Pointer to the scene.
`func`	Pointer to the call-back function.
`arg`	A user-supplied argument to be passed to the call-back function.

Return Value

The argument `scene` if successful, and `NULL` otherwise.

Comments

If the return type of the call-back function is not `RwLight` *, then the pointer to the call-back function should be cast to the expected type, i.e., a pointer to a function whose return type is `RwLight` *. For example, in the case of a call-back function named foo whose return type is int, the following C expression should be used:

```
(RwLight*(*)())foo
```

Note: **RwForAllLightsInSceneLong**() now has identical functionality to **RwForAllLightsInSceneInt**(). It is retained in this release for backward compatibility but will be removed from a future release of RenderWare. New applications should use **RwForAllLightsInSceneInt**().

See Also

RwAddLightToScene()
RwCreateLight()
RwDestroyLight()
RwForAllClumpsInScene()
RwGetSceneNumLights()
RwRemoveLightFromScene()

```
RwBool
RwForAllNamedTextures(RwTexture *(*func)(RwTexture *texture));
```

```
RwBool
RwForAllNamedTexturesInt(RwTexture *(*func)(RwTexture *texture, RwInt32 arg),
     RwInt32 arg);
```

```
RwBool
RwForAllNamedTexturesLong(RwTexture *(*func)(RwTexture *texture, RwInt32 arg),
     RwInt32 arg);
```

```
RwBool
RwForAllNamedTexturesReal(RwTexture *(*func)(RwTexture *texture, RwReal arg),
    RwReal arg);
```

RwBool
RwForAllNamedTexturesPointer(RwTexture *(*func)(RwTexture *texture, void *arg), void *arg);

Description

Applies a call-back function to all named textures. Depending on the current search mode, the scope is either the current texture dictionary (`rwLOCAL`) or the entire texture dictionary stack (`rwGLOBAL`). The call-back function can either be a RenderWare API function or a user-defined function. In the latter case, the call-back function should call **RwSetUserError**() if it fails for any reason.

The difference between **RwForAllNamedTextures**() and its variations listed above is that for **RwForAllNamedTextures**() the call-back function takes only one argument (a texture pointer), whereas in the case of its variations, the call-back function takes an additional, user-supplied argument (`arg`) that can be of type `RwInt32`, `RwReal` or `void *` respectively.

Arguments

`func`	Pointer to the call-back function.
`arg`	A user-supplied argument to be passed to the call-back function.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

If the return type of the call-back function is not `RwTexture` *, then the pointer to the call-back function should be cast to the expected type, i.e., a pointer to a function whose return type is `RwTexture` *. For example, in the case of a call-back function named `foo` whose return type is `int`, the following C expression should be used:

```
RwTexture*(*)())foo
```

Note: **RwForAllNamedTexturesLong**() now has identical functionality to **RwForAllNamedTexturesInt**(). It is retained in this release for backward compatibility but will be removed from a future release of RenderWare. New applications should use **RwForAllNamedTexturesInt**().

See Also

RwFindNamedTexture()
RwDestroyTexture()
RwGetNamedTexture()
RwReadNamedTexture()
RwSetTextureDictSearchMode()
RwSetUserError()
RwTextureDictBegin()
RwTextureDictEnd()

```
RwClump *
RwForAllPolygonsInClump(RwClump *clump,
     RwPolygon3d *(*func)(RwPolygon3d *polygon));
```

```
RwClump *
RwForAllPolygonsInClumpInt(RwClump *clump,
     RwPolygon3d *(*func)(RwPolygon3d *polygon, RwInt32 arg),
     RwInt32 arg);
```

```
RwClump *
RwForAllPolygonsInClumpLong(RwClump *clump,
     RwPolygon3d *(*func)(RwPolygon3d *polygon, RwInt32 arg),
     RwInt32 arg);
```

```
RwClump *
RwForAllPolygonsInClumpReal(RwClump *clump,
     RwPolygon3d *(*func)(RwPolygon3d *polygon, RwReal arg),
     RwReal arg);
```

```
RwClump *
RwForAllPolygonsInClumpPointer(RwClump *clump,
    RwPolygon3d *(*func)(RwPolygon3d *polygon, void *arg),
    void *arg);
```

Description

Applies a call-back function to all polygons belonging to a given clump. If any invocation of the call-back function sets RenderWares error status, iteration is terminated. The call-back function can either be a RenderWare API function or a user-defined function. In the latter case, the call-back function should call **RwSetUserError**() if it fails for any reason.

The difference between **RwForAllPolygonsInClump**() and its variations listed above is that for **RwForAllPolygonsInClump**() the call-back function takes only one argument (a polygon pointer), whereas in the case of its variations, the call-back function takes an additional, user-supplied argument (`arg`) that can be of type `RwInt32`, `RwReal` or `void *` respectively.

Arguments

`clump`	Pointer to the clump.
`func`	Pointer to the call-back function.
`arg`	A user-supplied argument to be passed to the call-back function.

Return Value

The argument `clump` if successful, and `NULL` otherwise.

Comments

If the return type of the call-back function is not `RwPolygon3d` *, then the pointer to the call-back function should be cast to the expected type, i.e., a pointer to a function whose return type is `RwPolygon3d` *. For example, in the case of a call-back function named foo whose return type is int, the following C expression should be used:

```
(RwPolygon3d*(*)()) foo
```

Note: **RwForAllPolygonsInClumpLong**() now has identical functionality to **RwForAllPolygonsInClumpInt**(). It is retained in this release for backward compatibility but will be removed from a future release of RenderWare. New applications should use **RwForAllPolygonsInClumpInt**().

See Also

RwAddPolygonToClump()
RwAddPolygonsToClump()
RwFindTaggedPolygon()
RwGetClumpNumPolygons()
RwPolygon()
RwPolygonExt()
RwQuad()
RwQuadExt()
RwSetUserError()
RwTriangle()
RwTriangleExt()

```
RwClump *
RwForAllUserDrawsInClump(RwClump *clump,
     RwUserDraw *(*func)(RwUserDraw *userdraw));
```

```
RwClump *
RwForAllUserDrawsInClumpInt(RwClump *clump,
     RwUserDraw *(*func)(RwUserDraw *userdraw, RwInt32 arg),
     RwInt32 arg);
```

```
RwClump *
RwForAllUserDrawsInClumpLong(RwClump *clump,
     RwUserDraw *(*func)(RwUserDraw *userdraw, RwInt32 arg),
     RwInt32 arg);
```

```
RwClump *
RwForAllUserDrawsInClumpReal(RwClump *clump,
     RwUserDraw *(*func)(RwUserDraw *userdraw, RwReal arg),
     RwReal arg);
```

```
RwClump *
RwForAllUserDrawsInClumpPointer(RwClump *clump,
    RwUserDraw *(*func)(RwUserDraw *userdraw, void *arg),
    void *arg);
```

Description

Applies a call-back function to all user-draws belonging to a given clump. If any invocation of the call-back function sets RenderWares error status, iteration is terminated. The call-back function can either be a RenderWare API function or a user-defined function. In the latter case, the call-back function should call **`RwSetUserError()`** if it fails for any reason.

The difference between **`RwForAllUserDrawsInClump()`** and its variations listed above is that for **`RwForAllUserDrawsInClump()`** the call-back function takes only one argument (a user-draw pointer), whereas in the case of its variations, the call-back function takes an additional, user-supplied argument (`arg`) that can be of type `RwInt32`, `RwReal` or `void *` respectively.

Arguments

`clump`	Pointer to the clump.
`func`	Pointer to the call-back function.
`arg`	A user-supplied argument to be passed to the call-back function.

Return Value

The argument `clump` if successful, and `NULL` otherwise.

Comments

If the return type of the call-back function is not `RwUserDraw` *, then the pointer to the call-back function should be cast to the expected type, i.e., a pointer to a function whose return type is `RwUserDraw` *. For example, in the case of a call-back function named foo whose return type is int, the following C expression should be used:

```
(RwUserDraw*(*)())foo
```

Note: **`RwForAllUserDrawsInClumpLong()`** now has identical functionality to **`RwForAllUserDrawsInClumpInt()`**. It is retained in this release for backward compatibility but will be removed from a future release of RenderWare. New applications should use **`RwForAllUserDrawsInClumpInt()`**.

See Also

`RwAddUserDrawToClump()`
`RwDuplicateUserDraw()`
`RwGetClumpNumUserDraws()`
`RwRemoveUserDrawFromClump()`
`RwSetUserError()`

RwRGBColor *
RwGetCameraBackColor(RwCamera *camera, RwRGBColor *color);

Description

Retrieves the cameras background fill color.

Arguments

camera	Pointer to the camera.
color	Pointer to the RwRGBColor that will receive the cameras color.

Return Value

The argument color if successful, and NULL otherwise.

See Also

RwClearCameraViewport()
RwGetCameraBackdrop()
RwGetCameraBackdropViewportRect()
RwSetCameraBackColor()
RwSetCameraBackColorStruct()

```
RwRaster *
RwGetCameraBackdrop(RwCamera *camera);
```

Description

Retrieves the cameras backdrop raster.

Arguments

`camera` Pointer to the camera.

Return Value

Pointer to the cameras backdrop raster if one has been set, and `NULL` if there is an error or if no backdrop raster is associated with the camera. Errors can be checked for using **`RwGetError`**`()`.

See Also

`RwDestroyRaster``()`
`RwGetCameraBackColor``()`
`RwGetCameraBackdropOffset``()`
`RwGetCameraBackdropViewportRect``()`
`RwSetCameraBackdrop``()`

`RwCamera *`
`RwGetCameraBackdropOffset``(RwCamera *camera, RwInt32 *x, RwInt32 *y);`

Description

Retrieves the offset (from the origin of the cameras backdrop viewport rectangle) of the cameras backdrop.

Arguments

`camera`	Pointer to the camera.
`x`	Pointer to integer to receive the horizontal offset (in pixels).
`y`	Pointer to integer to receive the vertical offset (in pixels).

Return Value

The argument `camera` if successful, and `NULL` otherwise.

Comments

The X and Y offset (modulo the width and height of the backdrop) specify the pixel in the backdrop which will be mapped to the origin of the backdrop viewport rectangle. Therefore, the effect of increasing the X offset will be to scroll the backdrop to the left and increasing the Y offset will scroll the backdrop up.

For 16-bit applications accessing the RenderWare DLL the variables pointed to by `x` and `y` must be declared as `RwInt32`s and not `int`s.

See Also

`RwGetCameraBackdrop``()`
`RwGetCameraBackdropViewportRect``()`
`RwSetCameraBackdrop``()`
`RwSetCameraBackdropOffset``()`
`RwSetCameraBackdropViewportRect``()`

`RwCamera *`
`RwGetCameraBackdropViewportRect``(RwCamera *camera, RwInt32 *x, RwInt32 *y, RwInt32 *width, RwInt32 *height);`

Description

Retrieves the rectangular area of the viewport into which the cameras backdrop raster is rendered.

Arguments

`camera`	Pointer to the camera.
`x`	Pointer to integer to receive the X co-ordinate of rectangle (in viewport space co-ordinates).
`y`	Pointer to integer to receive the Y co-ordinate of rectangle (in viewport space co-ordinates).
`width`	Pointer to integer to receive the width of the rectangle (in viewport space units).
`height`	Pointer to integer to receive the height of the rectangle (in viewport space units).

Return Value

The argument `camera` if successful, and `NULL` otherwise.

Comments

If the backdrop viewport rectangle is larger than the backdrop raster, the raster will be tiled to fill the rectangle. If the backdrop viewport rectangle is smaller than the backdrop raster, the raster will be cropped to the rectangle.

Areas of the viewport not covered by the backdrop will be filled with the cameras background color.

The backdrop viewport rectangle is not automatically changed when the cameras viewport is modified. **`RwSetCameraBackdropViewportRect`**`()` should be used to modify the backdrop viewport rectangle appropriately when the cameras viewport is modified.

For 16-bit applications accessing the RenderWare DLL the variables pointed to by `x`, `y`, `width` and `height` must be declared as `RwInt32`s and not `int`s.

See Also

`RwGetCameraBackColor``()`
`RwGetCameraBackdrop``()`
`RwGetCameraBackdropOffset``()`
`RwSetCameraBackColor``()`
`RwSetCameraBackdrop``()`
`RwSetCameraBackdropOffset``()`
`RwSetCameraBackdropViewportRect``()`
`RwSetCameraViewport``()`

`void *`
`RwGetCameraData``(RwCamera *camera);`

Description

Retrieves the cameras user data pointer.

Arguments

`camera` Pointer to the camera.

Return Value

The user data pointer. `NULL` is returned if there is an error or if the user data pointer is `NULL`. Errors can be checked for using **`RwGetError`**`()`.

See Also

`RwGetError``()`
`RwSetCameraData``()`

RwReal
RwGetCameraFarClipping(RwCamera *camera);

Description

Retrieves the distance from the camera's position to the back clipping plane.

Arguments

`camera` Pointer to the camera.

Return Value

The distance from the camera to the far clipping plane if successful, and `CREAL(-1.0)` otherwise.

See Also

RwCreateCamera()
RwGetCameraNearClipping()
RwSetCameraFarClipping()
RwSetCameraNearClipping()

`void *`
`RwGetCameraImage``(RwCamera *camera);`

Description

Retrieves a pointer to the cameras image buffer.

Arguments

`camera` Pointer to the camera.

Return Value

A pointer to the image buffer if successful, and `NULL` otherwise.

Comments

The image buffer format is device dependent. For more information, see Appendix B.

See Also

`RwCreateCamera``()`
`RwCreateUserDraw``()`
`RwDuplicateCamera``()`
`RwDestroyCamera``()`

```
RwV3d *
RwGetCameraLookAt(RwCamera *camera, RwV3d *vector);
```

Description

Retrieves the cameras Look At vector (the direction in which the camera points).

Arguments

`camera`	Pointer to the camera.
`vector`	Pointer to the vector that will receive the Look At vector.

Return Value

The argument `vector` if successful, and `NULL` otherwise.

See Also

RwGetCameraLookRight()
RwGetCameraLookUp()
RwPanCamera()
RwPointCamera()
RwResetCamera()
RwSetCameraLookAt()
RwTiltCamera()
RwTransformCameraOrientation()

```
RwV3d *
RwGetCameraLookRight(RwCamera *camera, RwV3d *vector);
```

Description

Retrieves the cameras Look Right (or U) vector.

Arguments

`camera`	Pointer to the camera.
`vector`	Pointer to the vector that will receive the Look Right vector.

Return Value

The argument `vector` if successful, and `NULL` otherwise.

See Also

RwGetCameraLookAt()
RwGetCameraLookUp()
RwPanCamera()
RwPointCamera()
RwResetCamera()
RwRevolveCamera()
RwSetCameraLookAt()
RwSetCameraLookUp()
RwTransformCameraOrientation()

RwV3d *
RwGetCameraLookUp(RwCamera *camera, RwV3d *vector);

Description

Retrieves the cameras Look Up (or V) vector.

Arguments

camera	Pointer to the camera.
vector	Pointer to the vector that will receive the Look Up vector.

Return Value

The argument vector if successful, and NULL otherwise.

See Also

RwGetCameraLookAt()
RwGetCameraLookRight()
RwPointCamera()
RwRevolveCamera()
RwResetCamera()
RwSetCameraLookUp()
RwTiltCamera()
RwTransformCameraOrientation()

`RwMatrix4d *`
`RwGetCameraLTM(``RwCamera *camera, RwMatrix4d *matrix`**`)`**

Description

Retrieves the cameras Local Transformation Matrix (LTM) which maps object space to world space.

Arguments

`camera`	Pointer to the camera.
`matrix`	Pointer to the matrix that will receive the LTM.

Return Value

The argument `matrix` if successful and `NULL` otherwise.

Comments

The matrix returned by this function may be used to position a light or a clump at the camera. The following code fragment demonstrates this.

```
RwGetCameraLTM(Camera, RwScratchMatrix());
RwTransformLight(Light, RwScratchMatrix(), rwREPLACE);
```

See Also

`RwCreateCamera``()`
`RwGetCameraLookAt``()`
`RwGetCameraLookRight``()`
`RwGetCameraLookUp``()`
`RwGetCameraPosition``()`
`RwGetClumpLTM``()`
`RwGetLightLTM``()`
`RwResetCamera``()`
`RwSetCameraLookAt``()`
`RwSetCameraLookUp``()`
`RwSetCameraPosition``()`
`RwTransformCamera``()`
`RwTransformClump``()`
`RwTransformLight``()`

RwReal
RwGetCameraNearClipping(RwCamera *camera);

Description

Retrieves the distance from the cameras position to the near clipping plane.

Arguments

`camera` Pointer to the camera.

Return Value

The distance from the camera to the near clipping plane if successful, and `CREAL(–1.0)` otherwise.

See Also

RwCreateCamera()
RwGetCameraFarClipping()
RwSetCameraFarClipping()
RwSetCameraNearClipping()

```
RwV3d *
RwGetCameraPosition(RwCamera *camera, RwV3d *position);
```

Description

Retrieves the cameras position in world space.

Arguments

`camera`	Pointer to the camera.
`position`	Pointer to the point that will receive the cameras position (in world space co-ordinates).

Return Value

The argument `position` if successful, and `NULL` otherwise.

See Also

RwCreateCamera()
RwDuplicateCamera()
RwResetCamera()
RwSetCameraPosition()
RwTransformCamera()
RwVCMoveCamera()
RwWCMoveCamera()

`RwCameraProjection`
`RwGetCameraProjection``(RwCamera *camera);`

Description

Retrieves the cameras projection type.

Arguments

`camera`	Pointer to the camera.

Return Value

The cameras projection type if successful, and `rwNACAMERAPROJECTION` otherwise.

Comments

The projection types are:

`rwPARALLEL`	Parallel projection.
`rwPERSPECTIVE`	Perspective projection.

See Also

`RwCreateCamera``()`
`RwSetCameraProjection``()`

```
RwV3d *
RwGetCameraViewOffset(RwCamera *camera, RwV3d *offset);
```

Description

Retrieves the view offset of the camera.

Arguments

`camera`	Pointer to the camera.
`offset`	Pointer to the vector to receive the view offset.

Return Value

The argument `offset` if successful, and `NULL` otherwise.

Comments

The X field of offset will be set to the offset in the direction of the cameras "Look Right" vector, the Y field will be set to the offset in the direction "Look Up" vector, whilst the Z field will be set to `CREAL(0.0)`.

See Also

RwResetCamera()
RwSetCameraViewOffset()

```
RwCamera *
RwGetCameraViewport(RwCamera *camera, RwInt32 *x, RwInt32 *y,
    RwInt32 *width, RwInt32 *height);
```

Description

Retrieves the cameras viewport in device space co-ordinates.

Arguments

`camera`	Pointer to the camera.
`x`	Pointer to the integer that will receive the X co-ordinate of the top left corner of the viewport (in device space co-ordinates).
`y`	Pointer to the integer that will receive the Y co-ordinate of the top left corner of the viewport (in device space co-ordinates).
`width`	Pointer to the integer that will receive the width of the viewport (in device space units).
`height`	Pointer to the integer that will receive the height of the viewport (in device space units).

Return Value

The argument `camera` if successful, and `NULL` otherwise.

Comments

The viewport origin is the top left of the viewport.

For 16-bit applications accessing the RenderWare DLL the variables pointed to by `x`, `y`, `width` and `height` must be declared as `RwInt32`s and not `int`s.

See Also

RwGetCameraViewwindow()
RwResetCamera()
RwSetCameraViewport()
RwSetCameraViewwindow()

RwRaster *
RwGetCameraViewportRaster(RwCamera *camera, RwRaster *raster);

Description

Copies the cameras viewport to the specified raster.

Arguments

`camera`	Pointer to the source camera.
`raster`	Pointer to the destination raster.

Return Value

The argument `raster` if successful, and `NULL` otherwise.

Comments

RwGetCameraViewportRaster() performs a straight copy. No conversion, filtering or color matching is performed. `raster` must have been previously created by **RwCreateRaster**() and must be large enough to hold the cameras viewport.

If the raster is to be subsequently used as a texture map the width and height of the cameras viewport must be 128.

See Also

RwBitmapRaster()
RwCreateRaster()
RwCreateTexture()
RwDestroyRaster()
RwDuplicateRaster()
RwGetCameraViewport()
RwGetTextureRaster()
RwSetCameraViewport()
RwSetTextureRaster()

```
RwCamera *
RwGetCameraViewwindow(RwCamera *camera, RwReal *width,
    RwReal *height);
```

Description

Retrieves the width and height of the cameras view window in world space units.

Arguments

`camera`	Pointer to the camera.
`width`	Pointer to the `RwReal` that will receive the width of the view window (in world space units).
`height`	Pointer to the `RwReal` that will receive the height of the view window (in world space units).

Return Value

The argument `camera` if successful, and `NULL` otherwise.

See Also

RwCreateCamera()
RwGetCameraViewport()
RwResetCamera()
RwSetCameraViewport()
RwSetCameraViewwindow()

RwAxisAlignment
RwGetClumpAxisAlignment(RwClump *clump);

Description

Retrieves the axis alignment type of the clump.

Arguments

clump	Pointer to the clump.

Return Value

The axis alignment type of the clump if successful, and rwNAAXISALIGNMENT otherwise.

Comments

The following axis alignment types are supported:

rwNOAXISALIGNMENT	The clump is not axis aligned, it is unconstrained.
RwALIGNAXISZORIENTX	The clumps local Z axis is aligned with the Look At vector of the camera, but the orientation of the 2D projection of the clumps local X axis is preserved.
RwALIGNAXISZORIENTY	The clumps local Z axis is aligned with the Look At vector of the camera, but the orientation of the 2D projection of the clumps local Y axis is preserved.
RwALIGNAXISXYZ	The local X, Y and Z axes of the clump are aligned with the cameras Look Right, Look Up and Look At vectors respectively.

A clump that is axis aligned will be aligned with the view planes of all cameras used to view that clump.

See Also

RwCreateSprite()
RwSetAxisAlignment()
RwSetClumpAxisAlignment()

```
RwClump *
RwGetClumpBBox(RwClump *clump, RwV3d *bll, RwV3d *fur);
```

Description

Retrieves the bounding box of the clump in world space co-ordinates.

Arguments

`clump`	Pointer to the clump.
`bll`	Pointer to the point that will receive the back, lower, left co-ordinates of the bounding box (in world space co-ordinates).
`fur`	Pointer to the point that will receive the front, upper, right co-ordinates of the bounding box (in world space co-ordinates).

Return Value

The argument `clump` if successful, and `NULL` otherwise.

Comments

This function is closely related to **`RwGetClumpLocalBBox()`**. However, this function returns a bounding box which is in world coordinate space. The bounding box returned by this function is aligned with the X, Y and Z axes of world space.

Note that this function is not recursive; it returns the bounding box of the specified clump only and not its descendants.

See Also

`RwGetClumpLocalBBox()`
`RwGetClumpViewportRect()`

```
void *
RwGetClumpData(RwClump *clump);
```

Description

Retrieves the clumps user data pointer.

Arguments

`clump`	Pointer to the clump.

Return Value

The user data pointer. `NULL` is returned if there is an error or if the user data pointer in `NULL`. Errors can be checked for using **`RwGetError()`**.

See Also

`RwGetError()`
`RwSetClumpData()`

`RwClumpHints`
`RwGetClumpHints``(RwClump *clump);`

Description

Retrieves the hints associated with the clump.

Arguments

`clump`	Pointer to the clump.

Return Value

A bitfield representing a hint (or bitwise or of hints) associated with the clump. If there is an error or if no hints have been added to the clump, `0` is returned. Errors can be checked for using **`RwGetError`**`()`.

Comments

The clump hints are:

`rwCONTAINER`	The clump spatially contains other clumps.
`rwHS`	Action should be taken to prevent hidden surfaces from being visible when the clump is rendered.
`rwEDITABLE`	The clumps geometry is editable (its vertices can be moved and new vertices and polygons added).

See Also

`RwAddHintToClump``()`
`RwGetError``()`
`RwRemoveHintFromClump``()`
`RwSetClumpHints``()`

RwMatrix4d *
RwGetClumpJointMatrix(RwClump *clump, RwMatrix4d *matrix);

Description

Retrieves the clumps joint (articulation) matrix.

Arguments

`clump`	Pointer to the clump.
`matrix`	Pointer to the matrix that will receive the clumps joint (articulation) matrix.

Return Value

The argument `matrix` if successful, and `NULL` otherwise.

See Also

RwGetClumpLTM()
RwGetClumpMatrix()
RwNormalizeClump()
RwTransformClumpJoint()

`RwClump *`
`RwGetClumpLocalBBox``(RwClump *clump, RwV3d *bll, RwV3d *fur);`

Description

Retrieves the bounding box of the clump in local space co-ordinates.

Arguments

`clump`	Pointer to the clump.
`bll`	Pointer to the point that will receive the back, lower, left co-ordinates of the bounding box (in local space co-ordinates).
`fur`	Pointer to the point that will receive the front, upper, right co-ordinates of the bounding box (in local space co-ordinates).

Return Value

The argument `clump` if successful, and `NULL` otherwise.

Comments

This function is closely related to **`RwGetClumpBBox`**`()`. However, this function returns a bounding box which is in local coordinate space (i.e., has not been transformed by the clumps LTM).

Note that this function is not recursive; it returns the bounding box of the specified clump only and not its descendants.

See Also

`RwGetClumpBBox``()`
`RwGetClumpViewportRect``()`

```
RwMatrix4d *
RwGetClumpLTM(RwClump *clump, RwMatrix4d *matrix);
```

Description

Retrieves the clumps Local Transformation Matrix (LTM) which maps object space to world space.

Arguments

`clump`	Pointer to the clump.
`matrix`	Pointer to the matrix that will receive the clumps LTM.

Return Value

The argument `matrix` if successful, and `NULL` otherwise.

Comments

The clumps LTM is the result of the concatenation of all modeling and joint (articulation) matrices from this clump to the root of the hierarchy.

See Also

RwAddChildToClump()
RwGetClumpJointMatrix()
RwGetClumpMatrix()
RwGetClumpParent()
RwGetClumpRoot()
RwGetFirstChildClump()
RwGetNextClump()
RwNormalizeClump()
RwTransformClump()
RwTransformClumpJoint()

```
RwMatrix4d *
RwGetClumpMatrix(RwClump *clump, RwMatrix4d *matrix);
```

Description

Retrieves the clumps modeling matrix.

Arguments

`clump`	Pointer to the clump.
`matrix`	Pointer to the matrix that will receive the clumps modeling matrix.

Return Value

The argument `matrix` if successful, and `NULL` otherwise.

See Also

RwGetClumpJointMatrix()
RwGetClumpLTM()
RwNormalizeClump()
RwTransformClump()

`RwInt32`
`RwGetClumpNumChildren``(RwClump *clump);`

Description

Retrieves the number of children of the clump.

Arguments

`clump` Pointer to the clump.

Return Value

The number of children of the clump if successful, and `–1` otherwise.

Comments

This function returns the number of direct children of the clump and not the number of descendants of the clump in the hierarchy.

See Also

`RwAddChildToClump``()`
`RwGetFirstChildClump``()`
`RwGetNextClump``()`
`RwRemoveChildFromClump``()`

```
RwInt32
RwGetClumpNumPolygons(RwClump *clump);
```

Description

Retrieves the number of polygons in the clump.

Arguments

`clump` Pointer to the clump.

Return Value

The number of polygons in the clump if successful, and `-1` otherwise.

See Also

RwAddPolygonToClump()
RwAddPolygonsToClump()
RwDestroyPolygon()
RwForAllPolygonsInClump()
RwGetClumpNumVertices()
RwPolygon()
RwPolygonExt()
RwQuad()
RwQuadExt()
RwTriangle()
RwTriangleExt()

```
RwInt32
RwGetClumpNumUserDraws(RwClump *clump);
```

Description

Retrieves the number of user-draws owned by the clump.

Arguments

`clump` Pointer to the clump.

Return Value

The number of user-draws owned by clump if successful, and `–1` otherwise.

See Also

RwAddUserDrawToClump()
RwDestroyUserDraw()
RwDuplicateUserDraw()
RwForAllUserDrawsInClump()
RwRemoveUserDrawFromClump()

```
RwInt32
RwGetClumpNumVertices(RwClump *clump);
```

Description

Retrieves the number of vertices in the clump.

Arguments

`clump` Pointer to the clump.

Return Value

The number of vertices in the clump if successful, and `-1` otherwise.

See Also

`RwAddVertexToClump``()`
`RwGetClumpNumPolygons``()`
`RwVertex``()`
`RwVertexExt``()`

```
RwV3d *
RwGetClumpOrigin(RwClump *clump, RwV3d *origin);
```

Description

Retrieves the origin of the clumps local co-ordinate (object) space in world space co-ordinates.

Arguments

`clump`	Pointer to the clump.
`origin`	Pointer to the point that will receive the clumps origin (in world space co-ordinates).

Return Value

The argument `origin` if successful, and `NULL` otherwise.

See Also

RwClumpDistance()
RwGetClumpLTM()

RwScene *
RwGetClumpOwner(RwClump *clump);

Description

Retrieves the scene that owns the clump.

Arguments

`clump` Pointer to the clump.

Return Value

The scene that owns the clump if successful, and `NULL` otherwise.

See Also

RwAddClumpToScene()
RwClumpBegin()
RwClumpEnd()
RwCreateClump()
RwCreateSprite()
RwDuplicateClump()
RwReadShape()
RwRemoveClumpFromScene()

`RwClump *`
`RwGetClumpParent``(RwClump *clump);`

Description

Retrieves the clumps parent.

Arguments

`clump` Pointer to the clump.

Return Value

A pointer to the parent clump. `NULL` is returned if the clump is the root of a hierarchy or if an error occurred. Errors can be checked for using **`RwGetError`**`()`.

See Also

`RwAddChildToClump``()`
`RwGetClumpRoot``()`
`RwGetError``()`
`RwGetFirstChildClump``()`
`RwRemoveChildFromClump``()`

`RwClump *`
`RwGetClumpRoot``(RwClump *clump);`

Description

Retrieves the root of the clump hierarchy containing the clump.

Arguments

`clump`	Pointer to the clump.

Return Value

The root clump if successful, and `NULL` otherwise.

See Also

`RwAddChildToClump``()`
`RwGetClumpParent``()`
`RwGetFirstChildClump``()`
`RwGetNextClump``()`
`RwRemoveChildFromClump``()`

`RwState`
`RwGetClumpState``(RwClump *clump);`

Description

Retrieves the clumps on/off state.

Arguments

`clump`	Pointer to the clump.

Return Value

The clumps state if successful, and `rwNASTATE` otherwise.

Comments

The states are:

`rwON`	The clump is to be a candidate for rendering and picking.
`rwOFF`	The clump is not to be a candidate for rendering and picking.

A state of `rwON` should be interpreted as making a clump a candidate for rendering and picking. Such a clump will not be rendered if it lies outside the view volume and it will not be picked unless one of its polygons is the foremost under the pick position.

The state affects only the clump to which it is applied and not to that clumps children. Thus, to prevent a single clump in a hierarchy from being rendered it is preferable to modify the clumps state rather than to remove it from a scene with **`RwRemoveClumpFromScene`**`()`.

See Also

`RwAddClumpToScene``()`
`RwDestroyClump``()`
`RwRemoveClumpFromScene``()`
`RwSetClumpState``()`

RwInt32
RwGetClumpTag(RwClump *clump);

Description

Retrieves the integer tag associated with the clump.

Arguments

clump Pointer to the clump.

Return Value

The clumps tag. Errors can be checked for using **RwGetError**().

See Also

RwFindTaggedClump()
RwGetError()
RwGetPolygonTag()
RwSetClumpTag()
RwSetTag()

```
RwV3d *
RwGetClumpVertex(RwClump *clump, RwInt32 index, RwV3d *coords);
```

Description

Retrieves the object space co-ordinates of the vertex which belongs to clump and has the vertex index `index`.

Arguments

`clump`	Pointer to the clump.
`index`	The vertex index.
`coords`	Pointer to the point that will receive the vertexs position (in object space co-ordinates).

Return Value

The argument `coords` if successful, and `NULL` otherwise.

Comments

The vertex index must be an integer greater than `0` and less than or equal to the number of vertices that belong to the clump.

See Also

RwGetClumpVertexNormal()
RwGetClumpVertexUV()
RwSetClumpVertex()
RwSetClumpVertices()

RwV3d *
RwGetClumpVertexNormal(RwClump *clump, RwInt32 index, RwV3d *normal);

Description

Returns the unit shading normal of the vertex which belongs to clump and has vertex index `index`.

Arguments

`clump`	Pointer to the clump.
`index`	The vertex index.
`normal`	Pointer to the vector that will receive the unit shading normal.

Return Value

The argument `normal` if successful, and `NULL` otherwise.

Comments

The vertex index must be an integer greater than `0` and less than or equal to the number of vertices that belong to the clump.

See Also

RwCalculateClumpVertexNormal()
RwGetClumpVertex()
RwGetClumpVertexUV()
RwGetPolygonNormal()
RwSetClumpVertexNormal()
RwVertexExt()

`RwUV *`
`RwGetClumpVertexUV``(RwClump *clump, RwInt32 index, RwUV *uv);`

Description

Retrieves the texture (U, V) co-ordinates of the vertex which belongs to clump and has vertex index `index`.

Arguments

`clump`	Pointer to the clump.
`index`	The vertex index.
`uv`	Pointer to the `RwUV` structure that will receive the texture co-ordinates of the vertex.

Return Value

The argument `uv` if successful, and `NULL` otherwise.

Comments

The vertex index must be an integer greater than `0` and less than or equal to the number of vertices that belong to the clump.

See Also

`RwCubicTexturizeClump``()`
`RwEnvMapClump``()`
`RwGetClumpVertex``()`
`RwGetClumpVertexNormal``()`
`RwSetClumpVertexUV``()`
`RwSetPolygonUV``()`
`RwSphericalTexturizeClump``()`
`RwVertexExt``()`

`RwBool`
`RwGetClumpVertexViewportPosition``(RwClump *clump, RwInt32 index, RwCamera *camera, RwInt32 *x, RwInt32 *y, RwBool *visible);`

Description

Retrieves the viewport space co-ordinates of the vertex belonging to clump with the vertex index `index` in the viewport of camera. `visible` indicates whether the vertex has been clipped from the view volume.

Arguments

`clump`	Pointer to the clump.
`index`	The vertex index.
`camera`	Pointer to the camera.
`x`	Pointer to integer to receive the X co-ordinate of the vertex (in viewport space co-ordinates).
`y`	Pointer to integer to receive the Y co-ordinate of the vertex (in viewport space co-ordinates).
`visible`	Pointer to integer to receive `TRUE` if the vertex is visible, and `FALSE` if it has been clipped from the view volume.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

If visible is `FALSE` the integer values pointed to by `x` and `y` are undefined.

For 16-bit applications accessing the RenderWare DLL the variables pointed to by `x` and `y` must be declared as `RwInt32`s and not `int`s. Furthermore, the variable pointed to by `visible` must be declared as an `RwBool`.

See Also

`RwGetClumpVertex``()`
`RwGetClumpViewportRect``()`
`RwSetClumpVertex``()`

`RwClump *`
`RwGetClumpViewportRect``(RwClump *clump, RwCamera *camera, RwInt32 *x, RwInt32 *y, RwInt32 *width, RwInt32 *height);`

Description

Retrieves the 2D rectangle that encloses the projection of the clump onto the specified cameras viewport.

Arguments

`clump`	Pointer to the clump.
`camera`	Pointer to the camera.
`x`	Pointer to the integer that will receive the X co-ordinate of the top left corner of the rectangle (in viewport space co-ordinates).
`y`	Pointer to the integer that will receive the Y co-ordinate of the top left corner of the rectangle (in viewport space co-ordinates).
`width`	Pointer to the integer that will receive the width of the rectangle (in viewport space units).
`height`	Pointer to the integer that will receive the height of the rectangle (in viewport space units).

Return Value

A pointer to the argument `clump`, and `NULL` otherwise.

For 16-bit applications accessing the RenderWare DLL the variables pointed to by `x`, `y`, `width` and `height` must be declared as `RwInt32`s and not `int`s.

See Also

`RwDamageCameraViewport``()`
`RwGetClumpBBox``()`
`RwGetClumpLocalBBox()`
`RwGetClumpVertexViewportPosition``()`
`RwUndamageCameraViewport``()`

RwState
RwGetDebugAssertionState(void);

Description

Gets the current state of assertion failure messages.

Arguments

None.

Return Value

The current state of assertion failure messages.

Comments

The assertion message states are:

rwON	Assertion messages are enabled.
rwOFF	Assertion messages are disabled.

See Also

RwGetDebugMessageState()
RwGetDebugScriptState()
RwGetDebugTraceState()
RwSetDebugAssertionState()
RwSetDebugOutputState()

RwState
RwGetDebugMessageState(void);

Description

Gets the current state of miscellaneous messages.

Arguments

None.

Return Value

The current state of miscellaneous messages.

Comments

The miscellaneous message states are:

rwON	Miscellaneous messages are enabled.
rwOFF	Miscellaneous messages are disabled.

See Also

RwGetDebugAssertionState()
RwGetDebugScriptState()
RwGetDebugTraceState()
RwSetDebugMessageState()
RwSetDebugOutputState()

RwState
RwGetDebugScriptState(void);

Description

Gets the current state of scripting trace messages.

Arguments

None.

Return Value

The current state of scripting trace messages.

Comments

The script trace message states are:

rwON	Script trace messages are enabled.
rwOFF	Script trace messages are disabled.

See Also

RwGetDebugAssertionState()
RwGetDebugMessageState()
RwGetDebugTraceState()
RwSetDebugOutputState()
RwSetDebugScriptState()

`RwDebugSeverity`
`RwGetDebugSeverity``(void);`

Description

Gets the current minimum severity level for the reporting of debugging messages.

Arguments

None.

Return Value

The current debug severity level.

Comments

The debug message severity levels are:

`rwINFORM`	Control flow annotations, non-fatal exceptions and fatal exceptions are all enabled.
`rwWARNING`	Non-fatal exceptions and fatal exceptions are enabled.
`rwERROR`	Fatal exceptions are enabled.

See Also

`RwSetDebugSeverity``()`

`RwState`
`RwGetDebugTraceState``(void);`

Description

Gets the current state of API function trace messages.

Arguments

None.

Return Value

The current state of API function trace messages.

Comments

The API function trace message states are:

`rwON`	API function trace messages are enabled.
`rwOFF`	API function trace messages are disabled.

See Also

`RwGetDebugAssertionState``()`
`RwGetDebugMessageState``()`
`RwGetDebugScriptState()`
`RwSetDebugOutputState``()`
`RwSetDebugTraceState``()`

RwBool
RwGetDeviceInfo(RwDeviceInfo info, void *value, RwInt32 size);

Description

Retrieves information about an aspect of the current RenderWare device driver. The specific information to query is given by `info`.

Arguments

`info`	Aspect of current device driver to query.
`value`	Pointer to a buffer to receive the result of the query. The actual data type of value is dependent on the value of info.
`size`	Size in bytes of the buffer pointed to by `param2`.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

The value parameter of each device information type is as follows:

`rwRENDERDEPTH`	A pointer to an RwInt32 to receive the current depth (in bits) of rendering.
`rwINDEXEDRENDERING`	A pointer to an RwBool which will be nonzero if rendering is indexed color based and zero if direct color based.
`rwPALETTEBASED`	A pointer to an RwBool which will be nonzero if the output device has a palette that RenderWare will attempt to modify, and zero if the output device uses direct color.

The following options apply when the output device is palette based:

`rwPALETTE`	A pointer to a device specific value which will receive a device dependent RenderWare palette object. See Appendix B for a description of this parameter.
`rwPALETTESIZE`	A pointer to an RwInt32 to receive the number of entries in the entire palette (this will normally be 256 for 8-bit devices).
`rwFIRSTPALETTEENTRY`	A pointer to an RwInt32 to receive the palette index of the first palette entry available for use by an application.
`rwLASTPALETTEENTRY`	A pointer to an RwInt32 to receive the palette index of the last palette entry available for use by an application.

Further information types may be supported by specific device drivers. See Appendix B for more information.

The `size` parameter is new with RenderWare V1.4. `size` gives the size in bytes of the buffer pointed to by value. For example to determine RenderWares current render depth the following would be used:

```
RwInt32 depth;
RwGetDeviceInfo(rwRENDERDEPTH, &depth, sizeof(depth);
```

See Also

RwDeviceControl()

RwGetSystemInfo()
RwSetPaletteEntries()
RwOpenExt()

```
RwErrorCode
RwGetError(void);
```

Description

Gets the value of RenderWares global error status (as set by the first function that generated an error since the last call to **RwGetError**()) and then clears the error status.

Arguments

None.

Return Value

An error code indicating the type of error that has occurred if the error status has been set. If no error has been set then `E_RW_NOERROR` is returned.

See Also

RwGetInternalError()
RwSetUserError()

`RwClump *`
`RwGetFirstChildClump``(RwClump *clump);`

Description

Retrieves the first child of the clump.

Arguments

`clump` Pointer to the clump.

Return Value

A pointer to the child clump. `NULL` is returned if the clump has no children or an error occurred. Errors can be checked for using **`RwGetError`**`()`.

Comments

In the absence of any deletions, the first child of a clump is the one that was first added to the clump using **`RwAddChildToClump`**`()`.

See Also

`RwAddChildToClump``()`
`RwGetClumpNumChildren``()`
`RwGetClumpParent``()`
`RwGetError``()`
`RwGetNextClump``()`
`RwRemoveChildFromClump``()`

```
RwInt32
RwGetInternalError(void);
```

Description

Retrieves a code representing the type of internal error that has occurred.

Arguments

None.

Return Value

The internal error code.

Comments

In the unlikely event that **RwGetError**() returns `E_RW_INTERNAL`, this function should be called in order to retrieve the internal error code. The number returned can then be reported to the RenderWare technical support department.

See Also

RwGetError()
RwSetUserError()

```
RwReal
RwGetLightBrightness(RwLight *light);
```

Description

Retrieves the lights brightness.

Arguments

`light` Pointer to the light.

Return Value

The brightness of the light if successful. Errors can be checked for using **`RwGetError`**`()`.

Comments

If the lights color has been previously set with a call to **`RwSetLightColor`**`()` then the value returned by **`RwGetLightBrightness`**`()` will be the average intensity of the red, green and blue channels of the lights color.

See Also

`RwCreateLight``()`
`RwGetError``()`
`RwSetLightBrightness``()`

```
RwRGBColor *
RwGetLightColor(RwLight *light, RwRGBColor *color);
```

Description

Retrieves the color of a light.

Arguments

`light` Pointer to the light.

`color` Pointer to the `RwRGBColor` that will receive the lights color.

Return Value

The argument `color` if successful, and `NULL` otherwise.

Comments

If a lights brightness has been previously set with **`RwSetLightBrightness`**`()` the red, green and blues channels of the color returned by **`RwGetLightColor`**`()` will be equal to the specified brightness.

In RenderWare V1.4, colored light sources are only available when performing 16-bit rendering. Under 8-bit rendering **`RwGetLightColor`**`()`, **`RwSetLightColor`**`()` and **`RwSetLightColorStruct`**`()` are still available to the API, however, light sources will always be white.

See Also

`RwSetLightColor``()`
`RwSetLightColorStruct``()`
`RwSetLightBrightness``()`

`RwReal`
`RwGetLightConeAngle``(RwLight *light);`

Description

Retrieves the cone angle of a conical or point light.

Arguments

`light` Pointer to the light.

Return Value

The cone angle of the light if successful. Errors can be checked for using **`RwGetError()`**.

Comments

For a point light source, `CREAL(180.0)` is returned.

See Also

`RwCreateLight()`
`RwGetError()`
`RwSetLightConeAngle()`

```
void *
RwGetLightData(RwLight *light);
```

Description

Retrieves the lights user data pointer.

Arguments

`light` Pointer to the light.

Return Value

The user data pointer. `NULL` is returned if there is an error or if the user data pointer is `NULL`. Errors can be checked for using **`RwGetError`**`()`.

See Also

`RwGetError``()`
`RwSetLightData``()`

`RwMatrix4d *`
RwGetLightLTM(`RwLight *light, RwMatrix4d *matrix`**)**

Description

Retrieves the lights Local Transformation Matrix (LTM) which maps object space to world space.

Arguments

`light`	Pointer to the light.
`matrix`	Pointer to the matrix that will receive the LTM.

Return Value

The argument `matrix` if successful and `NULL` otherwise.

Comments

The matrix returned by this function may be used to position a camera or a clump at the light. The following code fragment demonstrates this.

```
RwGetLightLTM(Light, RwScratchMatrix());
RwTransformCamera(Camera, RwScratchMatrix(), rwREPLACE);
```

See Also

RwCreateLight()
RwGetClumpLTM()
RwGetCameraLTM()
RwGetLightPosition()
RwGetLightVector()
RwSetLightPosition()
RwSetLightVector()
RwTransformCamera()
RwTransformClump()
RwTransformLight()

```
RwScene *
RwGetLightOwner(RwLight *light);
```

Description

Retrieves the scene that owns the light.

Arguments

`light`	Pointer to the light.

Return Value

The scene that owns the light if successful, and `NULL` otherwise.

See Also

RwAddLightToScene()
RwCreateLight()
RwRemoveLightFromScene()

```
RwV3d *
RwGetLightPosition(RwLight *light, RwV3d *position);
```

Description

Retrieves the position of a point or conical light in world space co-ordinates.

Arguments

`light`	Pointer to the light.
`position`	Pointer to the point that will receive the lights position (in world space co-ordinates).

Return Value

The argument `position` if successful, and `NULL` otherwise.

See Also

`RwCreateLight()`
`RwSetLightPosition()`
`RwTransformLight()`

`RwState`
`RwGetLightState``(RwLight *light);`

Description

Retrieves the lights on/off state.

Arguments

`light`	Pointer to the light.

Return Value

The lights state if successful, and `rwNASTATE` otherwise.

Comments

The states are:

`rwON`	The light is on.
`rwOFF`	The light is off.

See Also

`RwCreateLight``()`
`RwGetClumpState``()`
`RwSetLightState``()`

`RwLightType`
`RwGetLightType``(RwLight *light);`

Description

Retrieves the type of the light.

Arguments

`light`	Pointer to the light.

Return Value

The light type if successful, and `rwNALIGHTTYPE` otherwise.

Comments

The light types are:

`rwDIRECTIONAL`	A directional light source.
`rwPOINT`	A point light source.
`rwCONICAL`	A conical (or spot) light source.

See Also

`RwCreateLight``()`

```
RwV3d *
RwGetLightVector(RwLight *light, RwV3d *vector);
```

Description

Retrieves the illumination vector of a directional or conical light.

Arguments

`light`	Pointer to the light.
`vector`	Pointer to a vector that will receive the lights vector.

Return Value

The argument `vector` if successful, and `NULL` otherwise.

See Also

RwCreateLight()
RwSetLightVector()
RwTransformLight()

```
RwReal
RwGetMaterialAmbient(RwMaterial *material);
```

Description

Retrieves the materials ambient reflection coefficient.

Arguments

`material` Pointer to the material.

Return Value

The ambient reflection coefficient if successful. Errors can be checked for using **`RwGetError`**`()`.

See Also

`RwGetError``()`
`RwGetMaterialDiffuse``()`
`RwGetMaterialSpecular``()`
`RwGetPolygonAmbient``()`
`RwSetMaterialAmbient``()`
`RwSetMaterialSurface``()`

```
RwRGBColor *
RwGetMaterialColor(RwMaterial *material, RwRGBColor *color);
```

Description

Retrieves the materials color.

Arguments

`material`	Pointer to the material.
`color`	Pointer to the `RwRGBColor` that will receive the materials color.

Return Value

The argument `color` if successful, and `NULL` otherwise.

See Also

`RwGetPolygonColor()`
`RwSetMaterialColor()`
`RwSetMaterialColorStruct()`

RwReal
RwGetMaterialDiffuse(RwMaterial *material);

Description

Retrieves the materials diffuse reflection coefficient.

Arguments

`material` Pointer to the material.

Return Value

The diffuse reflection coefficient if successful. Errors can be checked for using **RwGetError**().

See Also

RwGetError()
RwGetMaterialAmbient()
RwGetMaterialSpecular()
RwGetPolygonDiffuse()
RwSetMaterialDiffuse()
RwSetMaterialSurface()

```
RwGeometrySampling
RwGetMaterialGeometrySampling(RwMaterial *material);
```

Description

Retrieves the materials geometry sampling type.

Arguments

`material` Pointer to the material.

Return Value

The materials geometry sampling type if successful, and `rwNAGEOMETRYSAMPLING` otherwise.

Comments

The geometry sampling types are:

`rwPOINTCLOUD`	Render geometry as a cloud of points.
`rwWIREFRAME`	Render geometry as a wireframe of polygon edges.
`rwSOLID`	Render geometry as a solid bounded by filled polygons.

See Also

RwGetMaterialLightSampling()
RwGetPolygonGeometrySampling()
RwSetMaterialGeometrySampling()

`RwLightSampling`
`RwGetMaterialLightSampling``(RwMaterial *material);`

Description

Retrieves the materials light sampling type.

Arguments

`material` Pointer to the material.

Return Value

The materials light sampling type if successful, and `rwNALIGHTSAMPLING` otherwise.

Comments

The light sampling types are:

`rwFACET`	Flat shading.
`rwVERTEX`	Smooth shading.

See Also

`RwGetMaterialGeometrySampling``()`
`RwGetPolygonLightSampling``()`
`RwSetMaterialLightSampling``()`

`RwReal`
`RwGetMaterialOpacity``(RwMaterial *material);`

Description

Retrieves the materials opacity

Arguments

`material` Pointer to the material.

Return Value

The opacity if successful. Errors can be checked for using **`RwGetError`**`()`.

Comments

An opacity of `CREAL(1.0)` yields an entirely opaque material. An opacity of `CREAL(0.0)` yields an entirely transparent material.

See Also

`RwGetError``()`
`RwGetPolygonOpacity``()`
`RwSetMaterialOpacity``()`

`RwReal`
`RwGetMaterialSpecular``(RwMaterial *material);`

Description

Retrieves the materials specular reflection coefficient.

Arguments

`material` Pointer to the material.

Return Value

The specular reflection coefficient if successful. Errors can be checked for using **`RwGetError`**`()`.

See Also

`RwGetError``()`
`RwGetMaterialAmbient``()`
`RwGetMaterialDiffuse``()`
`RwGetPolygonSpecular``()`
`RwSetMaterialSpecular``()`
`RwSetMaterialSurface``()`

```
RwTexture *
RwGetMaterialTexture(RwMaterial *material);
```

Description

Retrieves the materials texture.

Arguments

`material` Pointer to the material.

Return Value

A pointer to the materials texture if successful, and `NULL` if there is no texture associated with the material or if there is an error. Errors can be checked for using **RwGetError()**.

See Also

RwGetError()
RwGetMaterialTextureModes()
RwGetPolygonTexture()
RwSetMaterialTexture()

`RwTextureModes`
`RwGetMaterialTextureModes``(RwMaterial *material);`

Description

Retrieves the materials texture mode (or modes).

Arguments

`material` Pointer to the material.

Return Value

The materials texture modes if successful. Errors can be checked for using **`RwGetError`**`()`.

Comments

The texture modes are:

`rwLIT`	The texture will be lit according to the current light sampling type of the material (`rwFACET` or `rwVERTEX`).
`rwFORESHORTEN`	The texture will be foreshortened in a perspectively correct manner.
`rwFILTER`	A filter will be applied to the texture to reduce the effect of pixelation due to aliasing.

For further information see the Texture Modes section in Chapter 2: Data Types.

See Also

`RwAddTextureModeToMaterial``()`
`RwGetError``()`
`RwGetMaterialTexture``()`
`RwGetPolygonTextureModes``()`
`RwRemoveTextureModeFromMaterial``()`
`RwSetMaterialTextureModes``()`

RwReal
RwGetMatrixElement(RwMatrix4d *matrix, RwInt32 row, RwInt32 column);

Description

Retrieves the value of an individual element of the matrix.

Arguments

matrix	Pointer to the matrix.
row	Row index in the range 0 to 3.
column	Column index in the range 0 to 3.

Return Value

The matrix element if successful. Errors can be checked for using **RwGetError**().

See Also

RwGetError()
RwGetMatrixElements()
RwSetMatrixElement()
RwSetMatrixElements()

```
RwReal *
RwGetMatrixElements(RwMatrix4d *matrix, RwReal elements[4][4]);
```

Description

Retrieves the elements of a matrix into a four by four array of `RwReal`s. The top row of the matrix is copied into the first four array entries.

Arguments

`matrix`	Pointer to the matrix.
`elements`	Pointer to a four by four array of `RwReal`s to receive the elements of the matrix.

Return Value

The argument `elements` if successful, and `NULL` otherwise.

Comments

By convention a matrix is taken to transform a row vector by post multiplication.

See Also

RwGetMatrixElement()
RwSetMatrixElement()
RwSetMatrixElements()

`RwTexture *`
`RwGetNamedTexture``(char *name);`

Description

Searches for the named texture. If the current search mode is `rwLOCAL`, the function searches only the current dictionary. If the current search mode is `rwGLOBAL`, the function searches the whole of the texture dictionary stack. If the search fails, an attempt is made to read the named texture from disk. If the named texture is found, it is stored into the current texture dictionary.

Arguments

`name` Name of a texture.

Return Value

A pointer to the named texture if successful, and `NULL` otherwise.

Comments

The string supplied as the texture name should form the leaf part (i.e., without path or extension) of the filename of the texture file. Furthermore, for the sake of portability of texture files across the different platforms supported by RenderWare, it is best to choose texture file names that are a maximum of eight characters long and which are acceptable to MS-DOS as file names.

If the function cannot find the named texture in the texture dictionary stack, it will look for a texture file in the directories whose names appear in the shape path. Furthermore, if the specified name does not have a file extension, then this function will also search for the specified name followed by the extensions `.ras`, `.env`, `.tex`, `.bmp` and `.rle`.

An example of a valid texture is `marble`, which will match with file names `marble`, `marble.tex`, `marble.ras`, `marble.env`, `marble.bmp` and `marble.rle`.

See Also

`RwCreateTexture``()`
`RwDestroyTexture``()`
`RwFindNamedTexture``()`
`RwForAllNamedTextures``()`
`RwGetShapePath``()`
`RwReadNamedTexture``()`
`RwReadTexture``()`
`RwSetShapePath``()`
`RwSetTextureDictSearchMode``()`
`RwTextureDictBegin``()`
`RwTextureDictEnd``()`

```
RwClump *
RwGetNextClump(RwClump *clump);
```

Description

Retrieves the next sibling of the clump.

Arguments

`clump`	Pointer to the clump.

Return Value

A pointer to the sibling clump. `NULL` is returned if the clump has no next sibling or an error occurred. Errors can be checked for using **RwGetError**().

See Also

RwAddChildToClump()
RwGetClumpNumChildren()
RwGetClumpParent()
RwGetError()
RwGetFirstChildClump()
RwRemoveChildFromClump()

RwInt32

RwGetNumNamedTextures(void);

Description

Retrieves the number of named textures in either the current texture dictionary or in all dictionaries in the texture dictionary stack.

Arguments

None.

Return Value

The number of named textures if successful, and -1 otherwise.

Comments

If the texture dictionary search mode is rwLOCAL, the number of textures in the current texture dictionary is returned. If the search mode is rwGLOBAL the number of textures in all dictionaries on the texture dictionary stack is returned.

See Also

RwDestroyTexture()
RwFindNamedTexture()
RwGetNamedTexture()
RwReadNamedTexture()
RwSetTextureDictSearchMode()
RwTextureDictBegin()
RwTextureDictEnd()

```
RwPaletteEntry *
RwGetPaletteEntries(RwInt32 start, RwInt32 length,
    RwPaletteEntry *palette);
```

Description

Reads `length` entries from the current RenderWare palette starting at entry `start`.

Arguments

`start`	First palette entry to read.
`length`	Number of entries to read
`palette`	Pointer to an array of `RwPaletteEntry`s to hold retrieved values.

Return Value

The argument `palette` if successful, and `NULL` otherwise.

See Also

RwGetDeviceInfo()
RwSetPaletteEntries()

```
RwReal
RwGetPolygonAmbient(RwPolygon3d *polygon);
```

Description

Retrieves the ambient reflection coefficient of the polygons material.

Arguments

`polygon` Pointer to the polygon.

Return Value

The ambient reflection coefficient if successful. Errors can be checked for using **RwGetError()**.

See Also

RwGetError()
RwGetMaterialAmbient()
RwGetPolygonDiffuse()
RwGetPolygonSpecular()
RwSetPolygonAmbient()
RwSetPolygonSurface()

```
RwV3d *
RwGetPolygonCenter(RwPolygon3d *polygon, RwV3d *center);
```

Description

Retrieves the center of the polygon in object space co-ordinates.

Arguments

`polygon`	Pointer to the polygon.
`center`	Pointer to point that will receive the polygon center (in object space co-ordinates).

Return Value

The argument `center` if successful, and `NULL` otherwise.

See Also

`RwGetPolygonNormal``()`

`RwRGBColor *`
`RwGetPolygonColor``(RwPolygon3d *polygon, RwRGBColor *color);`

Description

Retrieves the color of the polygons material.

Arguments

`polygon` Pointer to the polygon.

`color` Pointer to the `RwRGBColor` that will receive the materials color.

Return Value

The argument `color` if successful, and `NULL` otherwise.

See Also

`RwGetMaterialColor``()`
`RwSetPolygonColor``()`
`RwSetPolygonColorStruct``()`

```
void *
RwGetPolygonData(RwPolygon3d *polygon);
```

Description

Retrieves the polygons user data pointer.

Arguments

`polygon` Pointer to the polygon.

Return Value

The user data pointer. `NULL` is returned if there is an error or if the user data pointer is `NULL`. Errors can be checked for using **RwGetError**().

See Also

RwGetError()
RwSetPolygonData()

```
RwReal
RwGetPolygonDiffuse(RwPolygon3d *polygon);
```

Description

Retrieves the diffuse reflection coefficient of the polygons material.

Arguments

`polygon` Pointer to the polygon.

Return Value

The diffuse reflection coefficient if successful. Errors can be checked for using **`RwGetError`**`()`.

See Also

`RwGetError``()`
`RwGetMaterialDiffuse``()`
`RwGetPolygonAmbient``()`
`RwGetPolygonSpecular``()`
`RwSetPolygonDiffuse``()`
`RwSetPolygonSurface``()`

```
RwGeometrySampling
RwGetPolygonGeometrySampling(RwPolygon3d *polygon);
```

Description

Retrieves the geometry sampling type of the polygons material.

Arguments

`polygon`	Pointer to the polygon.

Return Value

The geometry sampling type of the polygons material if successful, and `rwNAGEOMETRYSAMPLING` otherwise.

Comments

The geometry sampling types are:

`rwPOINTCLOUD`	Render geometry as a cloud of points.
`rwWIREFRAME`	Render geometry as a wireframe of polygon edges.
`rwSOLID`	Render geometry as a solid bounded by filled polygons.

See Also

RwGetMaterialGeometrySampling()
RwGetPolygonLightSampling()
RwSetPolygonGeometrySampling()

`RwLightSampling`
`RwGetPolygonLightSampling``(RwPolygon3d *polygon);`

Description

Retrieves the light sampling type of the polygons material.

Arguments

`polygon` Pointer to the polygon.

Return Value

The light sampling type of the polygons material if successful, and `rwNALIGHTSAMPLING` otherwise.

Comments

The light sampling types are:

`rwFACET`	Flat shading.
`rwVERTEX`	Smooth shading.

See Also

`RwGetMaterialLightSampling``()`
`RwGetPolygonGeometrySampling``()`
`RwSetPolygonLightSampling``()`

```
RwMaterial *
RwGetPolygonMaterial(RwPolygon3d *polygon);
```

Description

Retrieves the polygons material.

Arguments

`polygon` Pointer to the polygon.

Return Value

Pointer to the polygons material if successful, and `NULL` otherwise.

Comments

Do not attempt to destroy the material returned by this function.

See Also

RwDestroyMaterial()
RwGetPolygonAmbient()
RwGetPolygonColor()
RwGetPolygonDiffuse()
RwGetPolygonGeometrySampling()
RwGetPolygonLightSampling()
RwGetPolygonOpacity()
RwGetPolygonSpecular()
RwGetPolygonTexture()
RwGetPolygonTextureModes()
RwSetPolygonMaterial()

`RwV3d *`
`RwGetPolygonNormal``(RwPolygon3d *polygon, RwV3d *normal);`

Description

Retrieves the polygons surface normal vector.

Arguments

`polygon`	Pointer to the polygon.
`normal`	Pointer to the vector that will receive the polygon normal.

Return Value

The argument `normal` if successful, and `NULL` otherwise.

See Also

`RwGetClumpVertexNormal``()`
`RwGetPolygonCenter``()`

```
RwInt32
RwGetPolygonNumSides(RwPolygon3d *polygon);
```

Description

Retrieves the number of sides of the polygon.

Arguments

`polygon`	Pointer to the polygon.

Return Value

The number of sides of the polygon if successful, and `0` otherwise.

See Also

`RwGetPolygonVertices``()`

```
RwReal
RwGetPolygonOpacity(RwPolygon3d *polygon);
```

Description

Retrieves the opacity of the polygons material.

Arguments

`polygon` Pointer to the polygon.

Return Value

The opacity if successful. Errors can be checked for using **`RwGetError`**`()`.

Comments

An opacity of `CREAL(1.0)` yields an entirely opaque polygon. An opacity of `CREAL(0.0)` yields an entirely transparent polygon.

See Also

`RwGetError``()`
`RwGetMaterialOpacity``()`
`RwSetPolygonOpacity``()`

```
RwClump *
RwGetPolygonOwner(RwPolygon3d *polygon);
```

Description

Retrieves the clump that owns the polygon.

Arguments

`polygon` Pointer to the polygon.

Return Value

A pointer to the clump that owns the polygon if successful, and `NULL` otherwise.

See Also

RwFindTaggedPolygon()
RwForAllPolygonsInClump()

```
RwReal
RwGetPolygonSpecular(RwPolygon3d *polygon);
```

Description

Retrieves the specular reflection coefficient of the polygons material.

Arguments

`polygon` Pointer to the polygon.

Return Value

The specular reflection coefficient if successful. Errors can be checked for using **RwGetError**().

See Also

RwGetError()
RwGetMaterialSpecular()
RwGetPolygonAmbient()
RwGetPolygonDiffuse()
RwSetPolygonSpecular()

```
RwInt32
RwGetPolygonTag(RwPolygon3d *polygon);
```

Description

Retrieves the polygons tag.

Arguments

`polygon` Pointer to the polygon.

Return Value

The polygons tag if successful. Errors can be checked for using **`RwGetError`**`()`.

Note: Only the least significant 16 bits of the tag are valid. The most significant 16-bits will be set to zero.

See Also

`RwFindTaggedPolygon``()`
`RwGetError``()`
`RwGetClumpTag``()`
`RwPolygonExt``()`
`RwQuadExt``()`
`RwSetPolygonTag``()`
`RwVertexExt``()`

```
RwTexture *
RwGetPolygonTexture(RwPolygon3d *polygon);
```

Description

Retrieves the texture of the polygons material.

Arguments

`polygon` Pointer to the polygon.

Return Value

A pointer to the texture of the polygons material. `NULL` is returned if there is no texture associated with the polygons material or if there is an error. Errors can be checked for using **RwGetError**().

See Also

RwGetError()
RwGetMaterialTexture()
RwSetPolygonTexture()

`RwTextureModes`
`RwGetPolygonTextureModes``(RwPolygon3d *polygon);`

Description

Retrieves the texture mode (or modes) of the polygons material.

Arguments

`polygon`	Pointer to the polygon.

Return Value

The polygons materials texture modes if successful. Errors can be checked for using **`RwGetError`**`()`.

Comments

The texture modes are:

`rwLIT`	The texture will be lit according to the current light sampling type of the material (`rwFACET` or `rwVERTEX`).
`rwFORESHORTEN`	The texture will be foreshortened in a perspectively correct manner.
`rwFILTER`	A filter will be applied to the texture to reduce the effect of pixelation due to aliasing.

For further information see the Texture Modes section in Chapter 2: Data Types.

See Also

`RwAddTextureModeToPolygon``()`
`RwGetError``()`
`RwGetMaterialTextureModes``()`
`RwRemoveTextureModeFromPolygon``()`
`RwSetPolygonTexture``()`
`RwSetPolygonTextureModes``()`

```
RwUV *
RwGetPolygonUV(RwPolygon3d *polygon, RwUV *uvarray);
```

Description

Retrieves the texture (U, V) co-ordinates of the vertices of the polygon.

Arguments

`polygon`	Pointer to the polygon.
`uvarray`	Pointer to an array of `RwUV` structures that will receive the texture co-ordinates of the polygons vertices.

Return Value

The argument `uvarray` if successful, and `NULL` otherwise.

Comments

The size of the array `uvarray` must match the number of the vertices of the polygon.

See Also

RwCubicTexturizeClump()
RwEnvMapClump()
RwGetClumpVertexUV()
RwGetPolygonNumSides()
RwSetClumpVertexUV()
RwSetPolygonUV()
RwSphericalTexturizeClump()
RwVertexExt()

```
RwInt32
RwGetPolygonVertices(RwPolygon3d *polygon, RwInt32 *vlist);
```

Description

Retrieves the polygons vertex indices.

Arguments

`polygon`	Pointer to the polygon.
`vlist`	Pointer to an array of `RwInt32`s that will receive the vertex indices.

Return Value

The number of vertices if successful, and `0` otherwise.

Comments

For 16-bit applications accessing the RenderWare DLL the vertex index list pointed to by `vlist` must be declared as an array of `RwInt32`s and not `int`s.

See Also

RwAddPolygonToClump()
RwGetPolygonNumSides()
RwGetPolygonUV()

```
void *
RwGetRasterData(RwRaster *raster);
```

Description

Retrieves the rasters user data pointer.

Arguments

`raster` Pointer to the raster.

Return Value

The user data pointer. `NULL` is returned if there is an error or if the user data pointer is `NULL`. Errors can be checked for using **`RwGetError`**`()`.

See Also

`RwGetError``()`
`RwSetRasterData``()`

`RwInt32`
`RwGetRasterDepth``(RwRaster *raster);`

Description

Retrieves the depth (in bits) of the raster.

Arguments

`raster` Pointer to the raster.

Return Value

The depth (in bits) of `raster` if successful, and `–1` otherwise.

Comments

The raster depth is always equal to RenderWares current render depth.

See Also

`RwBitmapRaster``()`
`RwCreateRaster``()`
`RwGetDeviceInfo``()`
`RwGetRasterHeight``()`
`RwGetRasterPixels``()`
`RwGetRasterStride``()`
`RwGetRasterWidth``()`
`RwReadMaskRaster``()`
`RwReadRaster``()`

`RwInt32`
`RwGetRasterHeight``(RwRaster *raster);`

Description

Retrieves the height (in pixels) of the raster.

Arguments

`raster` Pointer to the raster.

Return Value

The height (in pixels) of `raster` if successful, and `-1` otherwise.

See Also

`RwBitmapRaster``()`
`RwCreateRaster``()`
`RwGetRasterDepth``()`
`RwGetRasterPixels``()`
`RwGetRasterStride``()`
`RwGetRasterWidth``()`
`RwReadMaskRaster``()`
`RwReadRaster``()`

`unsigned char *`
`RwGetRasterPixels``(RwRaster *raster);`

Description

Retrieves a pointer to the pixels of the raster.

Arguments

`raster` Pointer to the raster.

Return Value

A pointer to the pixels of `raster` if successful, and `NULL` otherwise.

Comments

The memory used to store the pixels of a raster may be stored in the memory of a peripheral device or may move in main memory. In order that an application can read and write to this memory it must be locked. **`RwGetRasterPixels`**`()` performs this locking. The pointer returned by this function must be released (and the associated memory unlocked) after use by a call to **`RwReleaseRasterPixels`**`()`. Following **`RwReleaseRasterPixels`**`()` the pointer is no longer valid and it must not be cached for later use. To prevent performance degradation it is essential that the pointer is released as soon as possible.

The pointer returned by this function points to an array of bytes organized into **`RwGetRasterHeight`**`()` scan lines. Each scan line is **`RwGetRasterStride`**`()` bytes in width.

The pixel format is dependent on the rasters depth. For an 8 bit raster each pixel occupies a single byte. This byte is an index into the RenderWare color palette. For a 16 bit raster each pixel occupies two bytes which are interpreted as a short (16 bit) integer. This integer represents a direct, RGB color specification. The least significant five bits (bits 0 - 4) are the blue channel, the next six bits (bits 5 - 10) are the green channel and the most significant five bits (bits 11 - 15) are the red channel.

Under Windows 3.1x, the type of the pointer returned by **`RwGetRasterPixels`**`()` can vary with the development environment being used. See Appendix B for more information.

See Also

`RwBitmapRaster``()`
`RwCreateRaster``()`
`RwGetRasterDepth``()`
`RwGetRasterHeight``()`
`RwGetRasterStride``()`
`RwGetRasterWidth``()`
`RwReadMaskRaster``()`
`RwReadRaster``()`
`RwReleaseRasterPixels``()`

```
RwInt32
RwGetRasterStride(RwRaster *raster);
```

Description

Retrieves the stride (width in bytes) of the raster.

Arguments

`raster` Pointer to the raster.

Return Value

The stride (width in bytes) of `raster` if successful, and `-1` otherwise.

See Also

RwBitmapRaster()
RwCreateRaster()
RwGetRasterDepth()
RwGetRasterHeight()
RwGetRasterPixels()
RwGetRasterWidth()
RwReadMaskRaster()
RwReadRaster()

```
RwInt32
RwGetRasterWidth(RwRaster *raster);
```

Description

Retrieves the width (in pixels) of the raster.

Arguments

`raster`   Pointer to the raster.

Return Value

The width (in pixels) of `raster` if successful, and `–1` otherwise.

See Also

RwBitmapRaster()
RwCreateRaster()
RwGetRasterDepth()
RwGetRasterHeight()
RwGetRasterPixels()
RwGetRasterStride()
RwReadMaskRaster()
RwReadRaster()

`void *`
`RwGetSceneData``(RwScene *scene);`

Description

Retrieves the scenes user data pointer.

Arguments

`scene` Pointer to the scene.

Return Value

The user data pointer. `NULL` is returned if there is an error or if the user data pointer is `NULL`. Errors can be checked for using **`RwGetError`**`()`.

See Also

`RwGetError``()`
`RwSetSceneData``()`

RwInt32
RwGetSceneNumClumps(RwScene *scene);

Description

Retrieves the number of clumps in the scene.

Arguments

scene	Pointer to the scene.

Return Value

The number of clumps in the scene if successful, and –1 otherwise.

See Also

RwAddClumpToScene()
RwClumpBegin()
RwClumpEnd()
RwCreateClump()
RwCreateSprite()
RwForAllClumpsInScene()
RwGetSceneNumLights()
RwReadShape()
RwRemoveClumpFromScene()

```
RwInt32
RwGetSceneNumLights(RwScene *scene);
```

Description

Retrieves the number of lights in the scene.

Arguments

`scene` Pointer to the scene.

Return Value

The number of lights in the scene if successful, and `-1` otherwise.

See Also

RwAddLightToScene()
RwCreateLight()
RwForAllLightsInScene()
RwGetSceneNumClumps()
RwRemoveLightFromScene()

```
char *
RwGetShapePath(char *path);
```

Description

Retrieves the current shape path.

Arguments

`path` Pointer to the string that will receive the path. The required size of path is defined by `RWMAXPATHLEN`.

Return Value

The argument `path` if successful, and `NULL` otherwise.

See Also

RwGetNamedTexture()
RwReadNamedTexture()
RwReadShape()
RwReadRaster()
RwReadMaskRaster()
RwReadTexture()
RwSetShapePath()
RwSetSurfaceTexture()
RwSetSurfaceTextureExt()

```
void *
RwGetSplineData(RwSpline *spline);
```

Description

Retrieves the splines user data pointer.

Arguments

`spline`	Pointer to the spline.

Return Value

The user data pointer. `NULL` is returned if there is an error or if the user data pointer is `NULL`. Errors can be checked for using **`RwGetError`**`()`.

See Also

`RwGetError``()`
`RwSetSplineData``()`

```
RwInt32
RwGetSplineNumPoints(RwSpline *spline);
```

Description

Retrieves the number of control points of the spline.

Arguments

`spline`	Pointer to the spline.

Return Value

Number of control points if successful, and `0` otherwise.

See Also

RwCreateSpline()
RwGetSplinePoint()
RwSetSplinePoint()

```
RwV3d *
RwGetSplinePoint(RwSpline *spline, RwInt32 index, RwV3d *point);
```

Description

Retrieves the specified control point of the spline.

Arguments

`spline`	Pointer to the spline.
`index`	Index of the control point to get, in the range `1` ≤ `index` ≤ total number of control points.
`point`	Pointer to the point that will receive the specified control point.

Return Value

The argument `point` if successful, and `NULL` otherwise.

Comments

Note that an `index` of `1` will retrieve the first control point.

See Also

RwCreateSpline()
RwGetSplineNumPoints()
RwSetSplinePoint()

RwBool
RwGetSystemInfo(RwSystemInfo info, void *value, RwInt32 size);

Description

Retrieves information about an aspect of the RenderWare system. The particular aspect of system configuration to query is given by `info`.

Arguments

`info`	Aspect of RenderWare system configuration to query.
`value`	Pointer to a buffer to receive the result of the query. The actual data type of value is dependent on the value of `info`.
`size`	Size in bytes of the buffer pointed to by `value`.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

The value parameter for each system information type is as follows:

`rwVERSIONSTRING`	A pointer to an array of characters which will receive a version string of the form `NN.nn rel`. Where `NN` is the major version number of the RenderWare library being used, `nn` is the minor version number and rel is a string identifying the release. For example, `1.4 FCS`.
`rwVERSIONMAJOR`	A pointer to an RwInt32 which will receive the major version number of the RenderWare library being used.
`rwVERSIONMINOR`	A pointer to an RwInt32 which will receive the minor version number of the RenderWare library being used.
`rwVERSIONRELEASE`	A pointer to an array of characters which will receive a string identify the particular release of the RenderWare library being used.
`rwFIXEDPOINTLIB`	A pointer to an RwBool which will be non-zero if the RenderWare library uses fixed point arithmetic, and zero if the library uses floating point arithmetic.
`rwDEBUGGINGLIB`	A pointer to an RwBool which will be non-zero if a debugging RenderWare library is being used, and zero if a retail library is being used.

The `size` parameter is new with RenderWare V1.4. `size` gives the size in bytes of the buffer pointed to by `value`. For example, to retrieve the RenderWare version string the following would be used:

```
char verStr[80];
RwGetSystemInfo(rwVERSIONSTRING, verStr, sizeof(verStr));
```

See Also

RwGetDeviceInfo()
RwOpenExt()

```
void *
RwGetTextureData(RwTexture *texture);
```

Description

Retrieves the textures user data pointer.

Arguments

`texture` Pointer to the texture.

Return Value

The user data pointer. `NULL` is returned if there is an error or if the user data pointer is `NULL`. Errors can be checked for using **`RwGetError`**`()`.

See Also

`RwGetError``()`
`RwSetTextureData``()`

RwSearchMode
RwGetTextureDictSearchMode(void);

Description

Retrieves the texture dictionary stacks current search mode.

Arguments

None.

Return Value

The current texture dictionary stack search mode.

Comments

The texture dictionary search modes are:

rwLOCAL	Search only the top most dictionary in the texture dictionary stack.
rwGLOBAL	Search all the dictionaries in the texture dictionary stack.

See Also

RwFindNamedTexture()
RwForAllNamedTextures()
RwGetNamedTexture()
RwSetTextureDictSearchMode()
RwTextureDictBegin()
RwTextureDictEnd()

`RwTextureDitherMode`
`RwGetTextureDithering``(void);`

Description

Retrieves the current global texture dithering mode applied to subsequently loaded textures.

Arguments

None.

Return Value

The current texture dithering mode.

Comments

The current texture dithering mode is a global parameter which controls whether textures read from disk are dithered to increase perceived color resolution.

The texture dithering modes are:

`rwDITHERON`	Activates dithering.
`rwDITHEROFF`	Deactivates dithering.
`rwAUTODITHER`	Adopts the auto-dithering mode of raster reading to decide whether to dither textures.

The default mode is `rwAUTODITHER`.

See Also

`RwGetTextureGammaCorrection``()`
`RwReadRaster``()`
`RwSetTextureDithering``()`
`RwSetTextureGammaCorrection``()`

`RwInt32`
`RwGetTextureFrame``(RwTexture *texture);`

Description

Gets the textures current frame index.

Arguments

`texture` Pointer to the texture.

Return Value

The current frame index (an integer greater than or equal to zero) if successful, and `-1` otherwise.

See Also

`RwGetTextureFrameStep``()`
`RwGetTextureNumFrames``()`
`RwSetTextureFrame``()`
`RwTextureNextFrame``()`

```
RwInt32
RwGetTextureFrameStep(RwTexture *texture);
```

Description

Retrieves the textures frame step size.

Arguments

`texture` Pointer to the texture.

Return Value

The current step size to be used by **`RwTextureNextFrame`**`()`. Errors can be checked for using **`RwGetError`**`()`.

See Also

`RwGetError``()`
`RwGetTextureFrame``()`
`RwGetTextureNumFrames``()`
`RwSetTextureFrameStep``()`
`RwTextureNextFrame``()`

RwState

RwGetTextureGammaCorrection(void);

Description

Retrieves the current global texture gamma correction mode applied to subsequently loaded textures.

Arguments

None.

Return Value

The current texture gamma correction mode.

Comments

The current texture gamma correction mode is a global parameter which controls whether textures read from disk are gamma corrected or not.

The texture gamma correction modes are:

rwON	Gamma correct.
rwOFF	Do not gamma correct.

The default mode is rwON.

See Also

RwGetTextureDithering()
RwReadRaster()
RwSetTextureDithering()
RwSetTextureGammaCorrection()

```
char *
RwGetTextureName(RwTexture *texture, char *name, RwInt32 size);
```

Description

Retrieves the textures name.

Arguments

`texture`	Pointer to the texture.
`name`	Pointer to the string that will receive the texture name.
`size`	Size of the string pointed to by `name`.

Return Value

The textures name. If there is an error or if the texture is not in a dictionary, `NULL` is returned. Errors can be checked for using **`RwGetError()`**.

Comments

This function has changed in RenderWare V1.4. Previously, this function returned a pointer to an internal texture name. It now copies the texture name into an application supplied string of the given size.

Only textures which are defined in dictionaries have names. Textures which are created using **`RwCreateTexture()`** or **`RwReadTexture()`** (rather than **`RwFindNamedTexture()`**, **`RwGetNamedTexture()`** or **`RwReadNamedTexture()`**) are not placed in dictionaries and hence have no name. **`RwGetNamedTexture()`** will return `NULL` for such textures and the contents of the string pointed to by `name` will be undefined.

See Also

`RwCreateTexture()`
`RwFindNamedTexture()`
`RwGetError()`
`RwGetNamedTexture()`
`RwReadNamedTexture()`
`RwReadTexture()`

```
RwInt32
RwGetTextureNumFrames(RwTexture *texture);
```

Description

Retrieves the number of frames in the texture.

Arguments

`texture` Pointer to the texture.

Return Value

Number of frames in the texture if successful, and `–1` otherwise.

See Also

RwGetTextureFrame()
RwGetTextureFrameStep()
RwSetTextureFrame()
RwSetTextureFrameStep()
RwTextureNextFrame()

```
RwRaster *
RwGetTextureRaster(RwTexture *texture);
```

Description

Retrieves a pointer to the textures raster.

Arguments

`texture` Pointer to the texture.

Return Value

The textures raster if successful, and `NULL` otherwise.

See Also

RwBitmapRaster()
RwCreateRaster()
RwCreateTexture()
RwDestroyRaster()
RwDuplicateRaster()
RwFindNamedTexture()
RwGetCameraViewportRaster()
RwGetNamedTexture()
RwReadNamedTexture()
RwReadRaster()
RwReadTexture()
RwSetTextureRaster()

`RwUserDrawAlignmentTypes`
`RwGetUserDrawAlignment``(RwUserDraw *userdraw);`

Description

Retrieves the alignment flag (or flags) of the user-draw. The alignment flags determine which part of the user-draws bounding box is used for alignment.

Arguments

`userdraw` Pointer to the user-draw.

Return Value

A bitfield representing the set of alignment flags associated with the user-draw if successful. Errors can be checked for using **`RwGetError`**`()`.

Comments

The alignment flags are:

`0`	Center the user-draw.
`rwALIGNTOP`	Align with the top edge of the user-draw.
`rwALIGNBOTTOM`	Align with the bottom edge of the user-draw.
`rwALIGNLEFT`	Align with the left edge of the user-draw.
`rwALIGNRIGHT`	Align with the right edge of the user-draw.

See Also

`RwCreateUserDraw``()`
`RwGetError``()`
`RwGetUserDrawParentAlignment``()`
`RwSetUserDrawAlignment``()`

`void`

(***RwGetUserDrawCallback**(RwUserDraw *userdraw)) (RwUserDraw *userdraw, void *camimage, RwRect *rect, void *data);

Description

Retrieves the call-back function used to render the user-draw.

Arguments

`userdraw` Pointer to the user-draw.

Return Value

A pointer to the user-draws call-back function if successful, and `NULL` otherwise.

Comments

User-draw call-backs should be declared as follows:

```
void callback(RwUserDraw *userdraw, void *camimage,
    RwRect *rect, void *data);
```

Where the call-backs arguments are:

`userdraw`	Pointer to the user-draw to be rendered.
`camimage`	The cameras image buffer as returned by **RwGetCameraImage**() for the current camera. `camimage` is device dependent. For more information, see Appendix B.
`rect`	Pointer to a rectangle defining the area of the cameras image buffer into which the call-back may render. This rectangle is specified in viewport space co-ordinates, i.e., `(0, 0)` is the origin of the viewport.
`data`	Pointer to the user data of the user-draw being drawn. This value can be obtained by calling **RwGetUserDrawData**() with `userdraw` as an argument. `data` is passed directly to the call-back function for the convenience of the application developer.

Note that the call-back function is always called after all clumps in the scene have been rendered, i.e., when **RwEndCameraUpdate**() is called. Therefore user-draw rendering always appear in front of clump rendering. In the case of overlapping user-draws, the order of rendering is not defined.

See Also

RwCreateUserDraw()
RwEndCameraUpdate()
RwGetCameraImage()
RwGetUserDrawData()
RwSetUserDrawCallback()

```
void *
RwGetUserDrawData(RwUserDraw *userdraw);
```

Description

Retrieves the user-draws user data pointer.

Arguments

`userdraw` Pointer to the user-draw.

Return Value

The user data pointer. `NULL` is returned if there is an error or if the user data pointer is `NULL`. Errors can be checked for using **RwGetError**().

See Also

RwGetError()
RwSetUserDrawData()

`RwUserDraw *`
`RwGetUserDrawOffset``(RwUserDraw *userdraw, RwInt32 *x, RwInt32 *y);`

Description

Retrieves the user-draws offset (in viewport space units) from the alignment point of the user-draw.

Arguments

`userdraw`	Pointer to the user-draw.
`x`	Pointer to the integer that will receive the horizontal offset from the alignment point of the user-draw (in viewport space units).
`y`	Pointer to the integer that will receive the vertical offset from the alignment point of the user-draw (in viewport space units).

Return Value

The argument `userdraw` if successful, and `NULL` otherwise.

See Also

`RwCreateUserDraw``()`
`RwSetUserDrawOffset``()`

`RwClump *`
`RwGetUserDrawOwner``(RwUserDraw *userdraw);`

Description

Retrieves the clump that owns the user-draw.

Arguments

`userdraw` Pointer to the user-draw.

Return Value

The clump that owns the user-draw. `NULL` is returned if the user-draw is not currently owned by a clump or if an error occurs. Errors can be checked for using **`RwGetError`**`()`.

See Also

`RwAddUserDrawToClump``()`
`RwCreateUserDraw``()`
`RwDuplicateUserDraw``()`
`RwForAllUserDrawsInClump``()`
`RwGetError``()`
`RwRemoveUserDrawFromClump``()`

`RwUserDrawAlignmentTypes`
`RwGetUserDrawParentAlignment``(RwUserDraw *userdraw);`

Description

Retrieves the alignment flag (or flags) of the user-draws parent. A user-draws parent is either the bounding box of the clump that owns the user-draw or the current cameras viewport.

The alignment flags of the user-draws parent determine which part of the user-draws parent rectangle is aligned with the user-draw. The actual point of alignment between a user-draw and its parent is determined by the user-draws alignment flags and the parents alignment flags.

Arguments

`userdraw` Pointer to the user-draw.

Return Value

A bitfield representing the set of alignment flags associated with the user-draws parent. Errors can be checked for using **`RwGetError`**`()`.

Comments

If the user-draws type is `rwBBOXALIGN` then the user-draws parent is the bounding box of the clump to which the user-draw is attached. If the type is `rwVPALIGN`, the user-draws parent is the viewport of the current camera when the user-draw is rendered. If the user-draws type is `rwVERTEXALIGN` or `rwCLUMPALIGN` then the user-draw has no parent and the parent alignment flags are ignored.

The alignment flags are:

`0`	Align with the center of the parent.
`rwALIGNTOP`	Align with the top edge of the parent.
`rwALIGNBOTTOM`	Align with the bottom edge of the parent.
`rwALIGNLEFT`	Align with the left edge of the parent.
`rwALIGNRIGHT`	Align with the right edge of the parent.

See Also

`RwCreateUserDraw``()`
`RwGetUserDrawAlignment``()`
`RwSetUserDrawAlignment``()`
`RwSetUserDrawParentAlignment``()`

```
RwUserDraw *
RwGetUserDrawSize(RwUserDraw *userdraw, RwInt32 *width, RwInt32 *height);
```

Description

Retrieves the width and height (in viewport space units) of the user-draw.

Arguments

`userdraw`	Pointer to the user-draw.
`width`	Pointer to the integer that will receive the width of the user-draw (in viewport space units).
`height`	Pointer to the integer that will receive the height of the user-draw (in viewport space units).

Return Value

The argument `userdraw` if successful, and `NULL` otherwise.

See Also

RwCreateUserDraw()
RwSetUserDrawSize()

```
RwUserDrawType
RwGetUserDrawType(RwUserDraw *userdraw);
```

Description

Retrieves the user-draws type.

Arguments

`userdraw` Pointer to the user-draw.

Return Value

The user-draws type if successful, and `rwNAUSERDRAWTYPE` otherwise.

Comments

The user-draw types are:

`rwCLUMPALIGN`	Align with the origin of the owning clump.
`rwVERTEXALIGN`	Align with a vertex of the owning clump.
`rwBBOXALIGN`	Align with the viewport bounding box of the owning clump.
`rwVPALIGN`	Align with the viewing cameras viewport.

See Also

RwCreateUserDraw()
RwGetClumpViewportRect()
RwSetUserDrawType()

`RwInt32`
`RwGetUserDrawVertexIndex``(RwUserDraw *userdraw);`

Description

Retrieves the index of the clump vertex with which the user-draw is aligned.

Arguments

`userdraw` Pointer to the user-draw.

Return Value

The index of the vertex the user-draw is aligned with if successful, and `0` otherwise.

Comments

The vertex index is only used if the user-draws type is `rwVERTEXALIGN`, for all other user-draw types it is ignored.

The vertex index is an index into the vertex list of the owning clump of the user-draw.

See Also

`RwCreateUserDraw``()`
`RwSetUserDrawVertexIndex``()`

RwBool
RwHemisphere(RwReal radius, RwInt32 density);

Description

Adds a hemisphere to the current clump. The hemisphere is transformed by the CTM, and the current material is assigned to its polygons. The base of the hemisphere lies on the X-Z plane.

Arguments

radius	Radius of the hemisphere.
density	Density of facets in the hemisphere. The number of facets increases exponentially with density.

Return Value

TRUE if successful, and FALSE otherwise.

Comments

This function can only be called within the context of an **RwClumpBegin**() … **RwClumpEnd**() or **RwProtoBegin**() … **RwProtoEnd**() block.

See Also

RwBlock()
RwClumpBegin()
RwClumpEnd()
RwCone()
RwCylinder()
RwDisc()
RwProtoBegin()
RwProtoEnd()
RwSphere()

RwBool
RwIdentityCTM(void);

Description

Sets the CTM to the identity matrix.

Arguments

None.

Return Value

TRUE if successful, and FALSE otherwise.

Comments

This function can only be called within the context of an **RwModelBegin**() … **RwModelEnd**() block.

See Also

RwIdentityJointTM()
RwModelBegin()
RwModelEnd()
RwRotateCTM()
RwScaleCTM()
RwTransformCTM()
RwTranslateCTM()

RwBool
RwIdentityJointTM(void);

Description

Sets the current joint transformation matrix to the identity matrix.

Arguments

None.

Return Value

TRUE if successful, and FALSE otherwise.

Comments

This function can only be called within the context of an **RwModelBegin**() … **RwModelEnd**() block.

See Also

RwIdentityCTM()
RwModelBegin()
RwModelEnd()
RwRotateJointTM()
RwTransformJointTM()

```
RwMatrix4d *
RwIdentityMatrix(RwMatrix4d *matrix);
```

Description

Sets the matrix to the identity matrix.

Arguments

`matrix` Pointer to the matrix.

Return Value

The argument `matrix` if successful, and `NULL` otherwise.

See Also

RwIdentityCTM()
RwIdentityJointTM()
RwInvertMatrix()
RwOrthoNormalizeMatrix()
RwRotateMatrix()
RwRotateMatrixCos()
RwScaleMatrix()
RwTransformMatrix()
RwTranslateMatrix()

```
RwBool
RwInclude(RwClump *clump);
```

Description

Inserts copies of the polygons and vertices of clump into the current clump under construction. The source polygons and vertices are transformed by the CTM before being added. The materials of the source polygons are copied to the new polygons.

Arguments

`clump`	Pointer to the clump.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

This function can only be called within the context of an **`RwClumpBegin`**`()` … **`RwClumpEnd`**`()` **or** **`RwProtoBegin`**`()` … **`RwProtoEnd`**`()` block.

See Also

`RwClumpBegin``()`
`RwClumpEnd``()`
`RwIncludeGeometry``()`
`RwProtoBegin``()`
`RwProtoEnd``()`
`RwReadShape``()`

RwBool
RwIncludeGeometry(RwClump *clump);

Description

Inserts copies of the polygons and vertices of the clump into the current clump under construction. The source polygons and vertices are transformed by the CTM before being added to the current clump. The current material is assigned to the new polygons (the materials of the source polygons are ignored).

Arguments

clump	Pointer to the clump.

Return Value

TRUE if successful, and FALSE otherwise.

Comments

This function can only be called within the context of an **RwClumpBegin**() … **RwClumpEnd**() **or** **RwProtoBegin**() … **RwProtoEnd**() block.

See Also

RwClumpBegin()
RwClumpEnd()
RwInclude()
RwProtoBegin()
RwProtoEnd()
RwReadShape()

`RwCamera *`
`RwInvalidateCameraViewport``(RwCamera *camera);`

Description

Marks the whole of the cameras viewport as damaged.

Arguments

`camera`	Pointer to the camera.

Return Value

The argument `camera` if successful, and `NULL` otherwise.

Comment

This function will cause the entire viewport to be copied to the display when the next call to **`RwShowCameraImage`**`()` is made. It will also cause the entire viewport to be cleared by the next call to **`RwClearCameraViewport`**`()`.

See Also

`RwClearCameraViewport``()`
`RwDamageCameraViewport``()`
`RwSetCameraViewport``()`
`RwShowCameraImage``()`
`RwUndamageCameraViewport``()`

```
RwMatrix4d *
RwInvertMatrix(RwMatrix4d *source, RwMatrix4d *dest);
```

Description

Inverts matrix `source` and stores the result in matrix `dest`.

Arguments

`source`	Pointer to the matrix to be inverted.
`dest`	Pointer to the matrix that will receive the result.

Return Value

The argument `dest` if successful, and `NULL` otherwise.

Comments

The source and destination arguments must not point to the same matrix.

See Also

RwIdentityMatrix()
RwOrthoNormalizeMatrix()
RwRotateMatrix()
RwRotateMatrixCos()
RwScaleMatrix()
RwTransformMatrix()
RwTranslateMatrix()

RwBool
RwJointTransformBegin(void);

Description

Pushes a copy of the current joint transformation matrix onto the joint transformation stack.

Arguments

None.

Return Value

TRUE if successful, and FALSE otherwise.

Comments

This function can only be called within the context of an **RwModelBegin**() … **RwModelEnd**() block.

See Also

RwClumpBegin()
RwClumpEnd()
RwIdentityJointTM()
RwJointTransformEnd()
RwModelBegin()
RwModelEnd()
RwProtoBegin()
RwProtoEnd()
RwRotateJointTM()
RwTransformBegin()
RwTransformJointTM()

RwBool
RwJointTransformEnd(void);

Description

Restores the previous value of the joint transformation matrix. Also has the effect of restoring the joint transformation stack to its state at the time of the last **RwJointTransformBegin**().

Arguments

None.

Return Value

TRUE if successful, and FALSE otherwise.

Comments

This function can only be called within the context of an **RwModelBegin**() … **RwModelEnd**() block.

See Also

RwClumpBegin()
RwClumpEnd()
RwIdentityJointTM()
RwJointTransformBegin()
RwModelBegin()
RwModelEnd()
RwProtoBegin()
RwProtoEnd()
RwRotateJointTM()
RwTransformJointTM()
RwTransformEnd()

RwTexture *
RwMaskTexture(RwTexture *texture, RwRaster *mask);

Description

Masks the texture with the mask raster mask.

Arguments

`texture`	Pointer to the texture.
`mask`	Pointer to the mask raster.

Return Value

The argument `texture` if successful, and `NULL` otherwise.

Comments

The mask raster must of exactly the same width and height as the texture. If the texture is a multi-frame texture, the masks height must be equal to `n` * 128, where `n` is the number of frames in the texture.

The mask raster must have been previously created with **RwReadMaskRaster**().

Masking a texture is a destructive operation. The masking cannot be undone and applies to all materials referencing the masked texture.

See Also

RwCreateRaster()
RwCreateTexture()
RwReadMaskRaster()
RwSetSurfaceTextureExt()
RwSetTextureRaster()

RwBool
RwMaterialBegin(void);

Description

Pushes a copy of the current material onto the material stack.

Arguments

None.

Return Value

TRUE if successful, and FALSE otherwise.

Comments

This function can only be called within the context of an **RwModelBegin**() … **RwModelEnd**() block.

See Also

RwClumpBegin()
RwClumpEnd()
RwMaterialEnd()
RwModelBegin()
RwModelEnd()
RwProtoBegin()
RwProtoEnd()
RwPushCurrentMaterial()

`RwBool`
`RwMaterialEnd``(void);`

Description

Restores the previous state of the current material. The material stack is restored to its state at the time of the last **`RwMaterialBegin`**`()`.

Arguments

None.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

This function can only be called within the context of an **`RwModelBegin`**`()` … **`RwModelEnd`**`()` block.

See Also

`RwClumpBegin``()`
`RwClumpEnd``()`
`RwMaterialBegin``()`
`RwModelBegin``()`
`RwModelEnd``()`
`RwPopCurrentMaterial``()`
`RwProtoBegin``()`
`RwProtoEnd``()`

RwBool
RwModelBegin(void);

Description

Sets up a modeling context for prototype declaration and clump creation. Prototype clumps declared within an **RwModelBegin**() … **RwModelEnd**() block may subsequently be instanced when building further prototypes or the desired model.

Arguments

None.

Return Value

TRUE if successful, and FALSE otherwise.

Comments

An **RwModelBegin**() … **RwModelEnd**() block may have any number of prototype clump declarations but must have one, and only one, top-level **RwClumpBegin**() … **RwClumpEnd**() block.

See Also

RwClumpBegin()
RwClumpEnd()
RwModelEnd()
RwProtoBegin()
RwProtoEnd()
RwProtoInstance()
RwProtoInstanceGeometry()

RwBool
RwModelEnd(void);

Description

Marks the end of a model definition.

Arguments

None.

Return Value

TRUE if successful, and FALSE otherwise.

Comments

An **RwModelBegin**() … **RwModelEnd**() block may have any number of prototype clump declarations but must have one, and only one, top-level **RwClumpBegin**() … **RwClumpEnd**().

RwModelEnd() will destroy all prototypes defined since the previous **RwModelBegin**().

See Also

RwClumpBegin()
RwClumpEnd()
RwModelBegin()
RwProtoBegin()
RwProtoEnd()

```
RwMatrix4d *
RwMultiplyMatrix(RwMatrix4d *a, RwMatrix4d *b, RwMatrix4d *c);
```

Description

Multiplies two matrices together.

Arguments

`a`	Pointer to the left matrix.
`b`	Pointer to the right matrix.
`c`	Pointer to the matrix that will receive the result.

Return Value

The argument `c` if successful, and `NULL` otherwise.

Comments

Note that the matrix used for the result (`c`) must be a different matrix from the matrices being multiplied together, (`a` and `b`).

See Also

`RwIdentityMatrix``()`
`RwInvertMatrix``()`
`RwOrthoNormalizeMatrix``()`
`RwRotateMatrix``()`
`RwRotateMatrixCos``()`
`RwScaleMatrix``()`
`RwTransformMatrix``()`
`RwTranslateMatrix``()`

```
RwV3d *
RwNormalize(RwV3d *vector);
```

Description

Normalizes a vector to unit length while retaining the ratio between its X, Y, and Z components.

Arguments

`vector` Pointer to the vector.

Return Value

The argument `vector` if successful, and `NULL` otherwise.

Comments

Note that it is an error to normalize a vector whose magnitude is zero.

See Also

RwAddVector()
RwCrossProduct()
RwDotProduct()
RwScaleVector()
RwSubtractVector()

```
RwClump *
RwNormalizeClump(RwClump *clump);
```

Description

Transforms the clump so that all the clumps vertices lie within unit space. The clumps modeling and joint (articulation) matrices are set to the identity.

Arguments

`clump` Pointer to the clump.

Return Value

The argument `clump` if successful, and `NULL` otherwise.

Comments

Note that this function is not recursive. The descendants of the specified clump (if any) are not normalized.

See Also

RwGetClumpBBox()
RwGetClumpLocalBBox()
RwGetClumpJointMatrix()
RwGetClumpLTM()
RwGetClumpMatrix()
RwTransformClump()
RwTransformClumpJoint()

`RwBool`
`RwOpen``(char *device, void *param);`

Description

Initializes the RenderWare library. This function (or its variant **`RwOpenExt`**`()`) must be called before any other RenderWare functions.

Arguments

`device`	A string whose value is either `NullDevice` or device-dependent.
`param`	A device dependent parameter.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

For a description of the device dependent parameters, `device` and `param`, see Appendix B.

See Also

`RwClose``()`
`RwOpenExt``()`

`RwBool`
`RwOpenDebugStream``(char *filename);`

Description

Opens the named file as the current debugging stream.

Arguments

`filename` Name of a file.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

Note that this function appends information to the specified file or device.

On certain platforms special filenames are recognized which, for example, allow the debugging stream to be redirected to a monochrome, debugging monitor. For more information, see Appendix B.

See Also

`RwCloseDebugStream``()`
`RwSetDebugStream``()`

```
RwBool
RwOpenExt(char *device, void *param, RwInt32 numargs,
    RwOpenArgument *args);
```

Description

Initializes the RenderWare library with a number of optional arguments. This function (or its variant **RwOpen()**) must be called before any other RenderWare functions.

Arguments

`device`	A string whose value is either `NullDevice` or device-dependent.
`param`	A device dependent parameter.
`numargs`	The number of optional arguments specified.
`args`	An array of optional open arguments.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

The array of arguments (`args`) is modified by the call to **RwOpenExt()**. After a call to **RwOpenExt()** the contents of the argument array are no longer defined and must be reinitialized if the argument array is to be used in another call to **RwOpenExt()**.

For a description of the device dependent parameters, `device` and `param` and the optional argument types supported, see Appendix B.

See Also

RwClose()
RwDeviceControl()
RwOpen()

`RwMatrix4d *`
`RwOrthoNormalizeMatrix``(RwMatrix4d *source, RwMatrix4d *dest);`

Description

Ortho-normalizes the source matrix and places the result in the destination matrix.

Arguments

`source`	Pointer to the matrix to orthonormalize.
`dest`	Pointer to the matrix to receive the result.

Return Value

The argument `dest` if successful, and `NULL` otherwise.

Comments

Whilst RenderWare supports arbitrary 4 x 4 homogenous matrices, many applications deal only in rigid body transformations comprising only rotation and translation without scaling or shearing.

The 4 x 4 homogeneous matrix representing such a transformation has a special form of upper-left 3 x 3 sub-matrix known as an orthonormal matrix. An orthonormal matrix is characterized by its inverse being equal to its transpose.

Mathematically, the upper-left 3 x 3 sub-matrix corresponding to a rigid body transformation remains orthonormal after that transformation is combined with other rigid body transformations. The upper 3 x 3 sub matrix corresponding to the inverse of a rigid body transformation should also be orthonormal.

Numerically however, after extended matrix composition, some scale or shear factors may begin to accumulate due to rounding. To prevent the significant build up of such factors, **`RwOrthoNormalizeMatrix`**`()` should be periodically applied to a clumps modeling or joint matrix as appropriate. This will filter out any such accumulated rounding factors from the upper-left 3 x 3 sub-matrix.

The minimal satisfactory frequency of orthonormalization will depend on the nature of the application and whether a fixed- or floating- point version of the RenderWare library is being used. Typically, an orthonormalization frequency of once every 128 frames is adequate.

See Also

`RwIdentityMatrix``()`
`RwInvertMatrix``()`
`RwMultiplyMatrix``()`
`RwOrthoNormalizeMatrix``()`
`RwRotateMatrix``()`
`RwRotateMatrixCos``()`
`RwScaleMatrix``()`
`RwTransformMatrix``()`
`RwTranslateMatrix``()`

`RwCamera *`
`RwPanCamera``(RwCamera *camera, RwReal angle);`

Description

Rotates the camera about its Y axis.

Arguments

`camera`	Pointer to the camera.
`angle`	Angle of rotation (in degrees).

Return Value

The argument `camera` if successful, and `NULL` otherwise.

Comments

A positive value for `angle` will pan the camera to the left.

See Also

`RwGetCameraLookAt``()`
`RwGetCameraLookRight``()`
`RwPointCamera``()`
`RwResetCamera``()`
`RwRevolveCamera``()`
`RwSetCameraLookAt``()`
`RwTiltCamera``()`
`RwTransformCameraOrientation``()`

RwPickRecord *
RwPickClump(RwClump *clump, RwInt32 vpx, RwInt32 vpy, RwCamera *camera, RwPickRecord *pick);

Description

Finds the frontmost polygon of the clump whose projection on the cameras viewport contains the specified point.

Arguments

`clump`	Pointer to the clump.
`vpx`	X co-ordinate (in viewport space co-ordinates).
`vpy`	Y co-ordinate (in viewport space co-ordinates).
`camera`	Pointer to the camera.
`pick`	Pointer to the pick record.

Return Value

A pointer to the argument `pick` if successful, and `NULL` otherwise.

Comments

`vpx` and `vpy` must be in viewport rather than device space co-ordinates. To convert from a point in device space co-ordinates (such as the position of the mouse) to viewport space co-ordinates simply subtract the X and Y co-ordinates of the cameras viewport from the X and Y co-ordinates of the point.

The pick record has a type field that will have either the value `rwNAPICKOBJECT` or `rwPICKCLUMP`. The former means that the clump was not picked.

If the pick records type is `rwPICKCLUMP`, then assuming that `pick` is a pick record structure whose address was passed as the last argument of the function, upon return from the function:

`pick.object.clump.clump`

is a pointer to the clump picked,

`pick.object.clump.polygon`

is a pointer to the polygon picked,

`pick.object.clump.vertex`

is an RwPickVertexData structure giving information about the picked vertex, and

`pick.object.clump.wcpoint`

is the world space co-ordinates of the actual point picked.

RwPickVertexData is defined as follows:

```
typedef struct
{
    RwInt32 vindex;
    RwInt32 d2;
} RwPickVertexData;
```

Fields `vindex` and `d2` specify respectively the index of the vertex picked and the square of its distance (in viewport space units) from the actual pick position.

See Also

RwPickScene()

```
RwPickRecord *
RwPickScene(RwScene *scene, RwInt32 vpx, RwInt32 vpy, RwCamera *camera,
    RwPickRecord *pick);
```

Description

Finds the frontmost clump of the scene whose projection on the cameras viewport contains the specified point.

Arguments

`scene`	Pointer to the scene.
`vpx`	X co-ordinate (in viewport space co-ordinates).
`vpy`	Y co-ordinate (in viewport space co-ordinates).
`camera`	Pointer to the camera.
`pick`	Pointer to the pick record.

Return Value

A pointer to the argument `pick` if successful, and `NULL` otherwise.

Comments

`vpx` and `vpy` must be in viewport rather than device space co-ordinates. To convert from a point in device space co-ordinates (such as the position of the mouse) to viewport space co-ordinates simply subtract the X and Y co-ordinates of the cameras viewport from the X and Y co-ordinates of the point.

The pick record has a type field that will have either the value `rwNAPICKOBJECT` or `rwPICKCLUMP`.

If the pick records type is `rwNAPICKOBJECT`, then no clumps were picked.

If the pick records type is `rwPICKCLUMP`, then assuming that `pick` is a pick record structure whose address was passed as the last argument of the function, upon return from the function:

```
pick.object.clump.clump
```

is a pointer to the clump picked,

```
pick.object.clump.polygon
```

is a pointer to the polygon picked,

```
pick.object.clump.vertex
```

is an RwPickVertexData structure giving information about the vertex picked, and

```
pick.object.clump.wcpoint
```

is the world space co-ordinates of the actual point picked.

RwPickVertexData is defined as follows:

```
typedef struct
{
    RwInt32 vindex;
    RwInt32 d2;
} RwPickVertexData;
```

Fields `vindex` and `d2` specify respectively the index of the vertex picked and the square of its distance (in viewport space units) from the actual pick position.

See Also

RwPickClump()

```
RwCamera *
RwPointCamera(RwCamera *camera, RwReal x, RwReal y, RwReal z);
```

Description

Re-orients the camera to point at the given point, while keeping its position constant.

Arguments

`camera`	Pointer to the camera.
`x`	X co-ordinate of the point to look at (in world space co-ordinates).
`y`	Y co-ordinate of the point to look at (in world space co-ordinates).
`z`	Z co-ordinate of the point to look at (in world space co-ordinates).

Return Value

The argument `camera` if successful, and `NULL` otherwise.

See Also

RwGetCameraLookAt()
RwGetCameraLookRight()
RwGetCameraLookUp()
RwPanCamera()
RwResetCamera()
RwRevolveCamera()
RwSetCameraLookAt()
RwSetCameraLookUp()
RwTiltCamera()
RwTransformCameraOrientation()

RwBool
RwPolygon(RwInt32 sides, RwInt32 *vlist);

Description

Adds a polygon to the current clump under construction. The current material is assigned to the polygon.

Arguments

`sides`	Number of sides of the polygon.
`vlist`	Pointer to an array of vertex indices.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

The size of the array `vlist` must be equal to `sides`.

This function can only be called within the context of an **RwClumpBegin**() … **RwClumpEnd**() or **RwProtoBegin**() … **RwProtoEnd**() block.

For 16-bit applications accessing the RenderWare DLL the vertex index list pointed to by `vlist` must be declared as an array of `RwInt32`s and not `int`s.

See Also

RwAddPolygonToClump()
RwClumpBegin()
RwClumpEnd()
RwPolygonExt()
RwProtoBegin()
RwProtoEnd()
RwQuad()
RwQuadExt()
RwTriangle()
RwTriangleExt()
RwVertex()
RwVertexExt()

`RwBool`
`RwPolygonExt``(RwInt32 sides, RwInt32 *vlist, RwInt32 tag);`

Description

Adds a polygon with the given integer tag to the current clump under construction. The current material is assigned to the polygon.

Arguments

`sides`	Number of sides of the polygon.
`vlist`	Array of vertex indices.
`tag`	Integer tag to set (only the least significant 16 bits are valid).

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

The size of the array `vlist` must be equal to `sides`.

This function can only be called within the context of an **`RwClumpBegin`**`()` … **`RwClumpEnd`**`()` or **`RwProtoBegin`**`()` … **`RwProtoEnd`**`()` block.

For 16-bit applications accessing the RenderWare DLL the vertex index list pointed to by `vlist` must be declared as an array of `RwInt32`s and not `int`s.

See Also

`RwAddPolygonToClump``()`
`RwClumpBegin``()`
`RwClumpEnd``()`
`RwFindTaggedPolygon``()`
`RwPolygon``()`
`RwProtoBegin``()`
`RwProtoEnd``()`
`RwQuad``()`
`RwQuadExt``()`
`RwSetPolygonTag``()`
`RwSetTag``()`
`RwTriangle``()`
`RwTriangleExt``()`
`RwVertex``()`
`RwVertexExt``()`

`RwMaterial *`
`RwPopCurrentMaterial``(void);`

Description

Pops the current material from the material stack, restoring the previously pushed material.

Arguments

None.

Return Value

A pointer to the new current material if successful, and `NULL` otherwise.

See Also

`RwCurrentMaterial``()`
`RwMaterialEnd``()`
`RwPushCurrentMaterial``()`

`RwMatrix4d *`
`RwPopScratchMatrix``(void);`

Description

Pops the current scratch matrix from the scratch matrix stack, restoring the previously pushed matrix.

Arguments

None.

Return Value

A pointer to the new scratch matrix if successful, and `NULL` otherwise.

Comments

The scratch matrix stack is a convenient source of temporary matrices for building transforms.

See Also

`RwScratchMatrix``()`
`RwPushScratchMatrix``()`

RwBool
RwProtoBegin(char *name);

Description

Identifies the beginning of a prototype clump declaration.

Arguments

name	Name of the prototype.

Return Value

TRUE if successful, and FALSE otherwise.

Comments

This function overrides any existing prototype of the same name.

This function can only be called within the context of an **RwModelBegin**() … **RwModelEnd**() block.

See Also

RwClumpBegin()
RwClumpEnd()
RwModelBegin()
RwModelEnd()
RwProtoEnd()
RwProtoInstance()
RwProtoInstanceGeometry()

RwBool
RwProtoEnd(void);

Description

Marks the end of a prototype clump declaration.

Arguments

None.

Return Value

TRUE if successful, and FALSE otherwise.

Comments

This function can only be called within the context of an **RwModelBegin**() … **RwModelEnd**() block.

See Also

RwClumpBegin()
RwClumpEnd()
RwModelBegin()
RwModelEnd()
RwProtoBegin()
RwProtoInstance()
RwProtoInstanceGeometry()

RwBool
RwProtoInstance(char *name);

Description

Creates an instance of the named prototype and copies its polygons and vertices to the clump under construction. The source polygons and vertices are transformed by the CTM before being added to the current clump. The materials of the source polygons are copied to the new polygons.

Arguments

name	Name of a prototype.

Return Value

TRUE if successful, and FALSE otherwise.

Comments

A prototype of the given name must have already been defined within the enclosing **RwModelBegin**() … **RwModelEnd**() block.

This function can only be called within the context of an **RwClumpBegin**() … **RwClumpEnd**() or **RwProtoBegin**() … **RwProtoEnd**() block.

See Also

RwClumpBegin()
RwClumpEnd()
RwModelBegin()
RwModelEnd()
RwProtoBegin()
RwProtoEnd()
RwProtoInstanceGeometry()

`RwBool`
`RwProtoInstanceGeometry``(char *name);`

Description

Creates an instance of the named prototype and copies its polygons and vertices to the clump under construction. The source polygons and vertices are transformed by the CTM before being added to the current clump. The current material is assigned to the new polygons. The materials of the source polygons are ignored.

Arguments

`name`	Name of a prototype.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

A prototype of the given name must have already been defined within the enclosing **`RwModelBegin`**`()` … **`RwModelEnd`**`()` block.

This function can only be called within the context of an **`RwClumpBegin`**`()` … **`RwClumpEnd`**`()` or **`RwProtoBegin`**`()` … **`RwProtoEnd`**`()` block.

See Also

`RwClumpBegin``()`
`RwClumpEnd``()`
`RwModelBegin``()`
`RwModelEnd``()`
`RwProtoBegin``()`
`RwProtoEnd``()`
`RwProtoInstance``()`

```
RwMaterial *
RwPushCurrentMaterial(void);
```

Description

Pushes a copy of the current material onto the material stack.

Arguments

None.

Return Value

A pointer to the new current material if successful, and `NULL` otherwise.

See Also

RwCurrentMaterial()
RwMaterialBegin()
RwPopCurrentMaterial()

`RwMatrix4d *`
`RwPushScratchMatrix``(void);`

Description

Pushes a copy of the scratch matrix onto the scratch matrix stack.

Arguments

None.

Return Value

A pointer to the new scratch matrix if successful, and `NULL` otherwise.

Comments

The scratch matrix stack is a convenient source of temporary matrices for building transforms.

See Also

`RwScratchMatrix``()`
`RwPopScratchMatrix``()`

`RwBool`
`RwQuad``(RwInt32 v1, RwInt32 v2, RwInt32 v3, RwInt32 v4);`

Description

Adds a quadrilateral to the current clump under construction. The current material is assigned to the polygon.

Arguments

`v1`	Index of the first vertex of the polygon.
`v2`	Index of the second vertex of the polygon.
`v3`	Index of the third vertex of the polygon.
`v4`	Index of the fourth vertex of the polygon.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

This function is exactly equivalent to calling **`RwPolygon`**`()` with an array of four vertex indices.

This function can only be called within the context of an **`RwClumpBegin`**`()` … **`RwClumpEnd`**`()` **or** **`RwProtoBegin`**`()` … **`RwProtoEnd`**`()` block.

See Also

`RwAddPolygonToClump``()`
`RwPolygon``()`
`RwPolygonExt``()`
`RwQuadExt``()`
`RwTriangle``()`
`RwTriangleExt``()`
`RwVertex``()`
`RwVertexExt``()`

RwBool
RwQuadExt(RwInt32 v1, RwInt32 v2, RwInt32 v3, RwInt32 v4,
RwInt32 tag);

Description

Adds a quadrilateral with the given integer tag to the current clump under construction. The current material is assigned to the polygon.

Arguments

`v1`	Index of the first vertex of the polygon.
`v2`	Index of the second vertex of the polygon.
`v3`	Index of the third vertex of the polygon.
`v4`	Index of the fourth vertex of the polygon.
`tag`	Integer tag to set (only the least significant 16 bits of the tag are valid).

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

This function is exactly equivalent to calling **RwPolygonExt**() with an array of four vertex indices.

This function can only be called within the context of an **RwClumpBegin**() … **RwClumpEnd**() or **RwProtoBegin**() … **RwProtoEnd**() block.

See Also

RwAddPolygonToClump()
RwFindTaggedPolygon()
RwGetPolygonTag()
RwPolygon()
RwPolygonExt()
RwQuad()
RwSetPolygonTag()
RwSetTag()
RwTriangle()
RwTriangleExt()
RwVertex()
RwVertexExt()

```
RwMatrix4d *
RwQueryRotateMatrix(RwMatrix4d *matrix, RwV3d *axis,
    RwReal *degrees);
```

Description

Retrieves the rotation component from a matrix comprising only rotations and translations. The rotation is returned as a unit direction vector along the axis of rotation through the origin and an angle in degrees. This function is the inverse of **`RwRotateMatrix()`** with the `RwCombineOperation` `rwREPLACE`.

Arguments

`matrix`	Pointer to the rotation/translation matrix.
`axis`	Pointer to the vector to receive the unit direction vector along the axis of rotation through the origin.
`degrees`	Pointer to the `RwReal` to receive the angle of rotation in degrees.

Return Value

The argument `matrix` if successful, and `NULL` otherwise.

Comments

A rotation has two possible descriptions in this axis/angle form, since a rotation about a given axis through the origin by a given angle *theta* is the same as a rotation about an axis in the reverse direction by the angle *360-theta*. The angle returned is always in the range `CREAL(0.0)` to `CREAL(180.0)`. The direction of the axis of rotation returned is chosen to ensure that the angle lies in this range.

Notice that only matrices known to be composed solely of rotations and translations should be queried with this function. The results of querying other matrices incorporating transforms such as scales are unlikely to be of practical use. **`RwOrthoNormalizeMatrix()`** may be applied to extract the rotation/translation component matrix from a more general transformation matrix.

See Also

`RwGetMatrixElements()`
`RwGetMatrixElement()`
`RwOrthoNormalizeMatrix()`
`RwRotateMatrix()`
`RwRotateMatrixCos()`

`RwInt32`
`RwRandom(``void`**`)`**

Description

Generates a pseudo random number

Arguments

None.

Return Value

A pseudo random `RwInt32` number. There is no error return.

Comments

`RwRandom()` uses a non-linear additive feedback random number generator employing a default table of size 31 long integers to return successive pseudo-random numbers in the range from `0` to (2**31)-1. The period of this random number generator is very large, approximately 16*((2**31)-1).

`RwRandom()` and **`RwSRandom()`** have (almost) the same calling sequence and initialization properties as `rand()` and `srand()`. The difference is that `rand()` produces a much less random sequence - in fact, the low dozen bits generated by rand go through a cyclic pattern. All the bits generated by **`RwRandom()`** are usable. For example,

```
RwRandom() & 01
```

will produce a random binary value.

See Also

`RwSRandom()`

RwRaster *
RwReadMaskRaster(char *filename);

Description

Reads a mask raster from the specified file. The resulting raster can be applied to a texture as a mask.

Arguments

`filename` Name of the mask raster file.

Return Value

A pointer to the new mask raster if successful, and `NULL` otherwise.

Comments

The raster read by **RwReadMaskRaster**() is used to mask a texture. Those pixels which are masked out by the raster are not rendered. Thus, a mask raster is effectively a one bit alpha channel which, when applied to a texture, gives control over the transparency of individual pixels.

The raster read by **RwReadMaskRaster**() is converted to the texture map dimensions of 128 pixels in width by 128 pixels in height (or `n` * 128 pixels for a multi-frame texture where `n` is the number of frames in the texture). Furthermore, the image read will be converted to the current RenderWare render depth.

The file read by this operation should contain a gray scale image of any supported depth. The gray scale value of each pixel is simply thresholded to determine whether it represents an opaque or transparent pixel. Pixels whose value is less than half of the available range of gray values represent transparent pixels. Those whose value is greater than or equal to half the available range of gray values represent opaque pixels.

If `filename` is not a full path, the library searches for the specified file in all directories on its shape path.

See Also

RwBitmapRaster()
RwCreateRaster()
RwCreateTexture()
RwDestroyRaster()
RwDuplicateRaster()
RwGetCameraViewportRaster()
RwGetShapePath()
RwMaskTexture()
RwSetCameraBackdrop()
RwSetShapePath()
RwSetTextureRaster()

`RwTexture *`
`RwReadNamedTexture``(char *name);`

Description

Reads a texture with the specified name and stores it into the current texture dictionary.

Arguments

`name`	Name of a texture.

Return Value

A pointer to a newly created texture if successful, and `NULL` otherwise.

Comments

This function uses the environment variable `RWSHAPEPATH` as its search path. The texture read from disk will replace any texture with the specified name that is already in the current texture dictionary.

The string supplied as the texture name must form the leaf part (i.e., without path or extension) of the pathname for the texture file. Furthermore, for the sake of portability of script files across the different platforms supported by RenderWare, it is best to choose texture file names that are a maximum of eight characters long and which are acceptable to MS-DOS as file names.

An example of a valid texture name is marble, which will match with file names `marble`, `marble.tex`, `marble.ras`, `marble.env`, `marble.bmp` and `marble.env`.

See Also

`RwCreateTexture``()`
`RwDestroyTexture``()`
`RwFindNamedTexture``()`
`RwGetNamedTexture``()`
`RwGetShapePath``()`
`RwReadTexture``()`
`RwSetShapePath``()`

`RwRaster *`
`RwReadRaster``(char *filename, RwRasterOptions options);`

Description

Reads a raster from the specified file. The raster will be processed according to the specified options.

Arguments

`filename`	Name of the raster file.
`options`	A bitfield representing a raster processing option (or bitwise or of options).

Return Value

A pointer to the new raster if successful, and `NULL` otherwise.

Comments

If filename is not a full path, the library searches for the specified file in all directories on its shape path.

The supported raster options are as follows:

`rwAUTODITHERRASTER`	Dither the raster only if the source bitmap is to be resized (`rwFITRASTER` has been specified) or if the bitmap is a different depth from the current RenderWare render depth.
`rwDITHERASTER`	Dither the raster.
`rwFITRASTER`	Resize the raster to texture map dimensions, i.e. 128 x `n` * 128 (where `n` is the number of frames in a multi-frame texture).
`rwGAMMARASTER`	Gamma correct the raster.

See Also

`RwBitmapRaster``()`
`RwCreateRaster``()`
`RwCreateTexture``()`
`RwDestroyRaster``()`
`RwDuplicateRaster``()`
`RwGetCameraViewportRaster``()`
`RwGetDeviceInfo``()`
`RwGetShapePath``()`
`RwReadMaskRaster``()`
`RwSetCameraBackdrop``()`
`RwSetShapePath``()`
`RwSetTextureDithering``()`
`RwSetTextureGammaCorrection``()`
`RwSetTextureRaster``()`

`RwClump *`
`RwReadShape``(char *filename);`

Description

Loads a clump from a script (`.rwx`) file. If filename is not an absolute path, the library searches for the specified file in all directories on its shape path.

Arguments

`filename` Pointer to the filename string.

Return Value

A pointer to the new clump if successful, and `NULL` otherwise

Comments

The clump is added to the default scene.

See Also

`RwClumpBegin``()`
`RwClumpEnd``()`
`RwCreateClump``()`
`RwDefaultScene``()`
`RwDestroyClump``()`
`RwGetShapePath``()`
`RwSetShapePath``()`
`RwWriteShape``()`

RwTexture *
RwReadTexture(char *filename);

Description

Reads a texture from the specified file.

Arguments

filename Name of the texture file.

Return Value

A pointer to the new texture if successful, and NULL otherwise.

Comments

If filename is not a full path, the library searches for the specified file in all directories on its shape path.

Unlike **RwReadNamedTexture**(), the texture read from disk will not be placed in the current texture dictionary.

See Also

RwCreateTexture()
RwDestroyTexture()
RwFindNamedTexture()
RwGetNamedTexture()
RwGetShapePath()
RwGetTextureRaster()
RwReadNamedTexture()
RwReadRaster()
RwSetShapePath()
RwSetTextureRaster()

`RwRaster *`
`RwReleaseRasterPixels``(RwRaster *raster, unsigned char *pixels);`

Description

Releases a pointer to the pixels of a raster previously obtained with **`RwGetRasterPixels`**`()`.

Arguments

`raster` Pointer to the raster.

`pixels` Pointer to the pixels of `raster`.

Return Value

The argument `raster` if successful, and `NULL` otherwise.

Comments

The pointer `pixels` must have been obtained by a call to **`RwGetRasterPixels`**`()` with `raster` as an argument.

The memory used to store the pixels of a raster may be stored in the memory of a peripheral device or may move in main memory. In order that an application can read and write to this memory it must be locked. **`RwGetRasterPixels`**`()` performs this locking. The pointer returned by this function must be released (and the associated memory unlocked) after use by a call to **`RwReleaseRasterPixels`**`()`. Following **`RwReleaseRasterPixels`**`()` the pointer is no longer valid and it must not be cached for later use. To prevent performance degradation it is essential that the pointer is released as soon as possible.

For Windows 3.1x applications the type of the `pixels` pointer can vary with the development environment used. See Appendix B for more information on the pointer type.

See Also

`RwBitmapRaster``()`
`RwCreateRaster``()`
`RwGetRasterDepth``()`
`RwGetRasterHeight``()`
`RwGetRasterPixels``()`
`RwGetRasterStride``()`
`RwGetRasterWidth``()`
`RwReadMaskRaster``()`
`RwReadRaster``()`

`RwClump *`
`RwRemoveChildFromClump``(RwClump *clump);`

Description

Removes the clump from its parents list of children.

Arguments

`clump` Pointer to the clump.

Return Value

The argument `clump` if successful, and `NULL` otherwise.

Comments

Once removed, the clump becomes the root of a hierarchy consisting of itself and any descendants.

See Also

`RwAddChildToClump``()`
`RwGetClumpNumChildren``()`
`RwGetClumpParent``()`
`RwGetClumpRoot``()`
`RwGetFirstChildClump``()`
`RwGetNextClump``()`

`RwClump *`
`RwRemoveClumpFromScene``(RwClump *clump);`

Description

Removes the clump from its scene.

Arguments

`clump` Pointer to the clump.

Return Value

The argument `clump` if successful, and `NULL` otherwise.

Comments

The clump being removed must be the root of its clump hierarchy, i.e., it must not have a parent clump.

The clump is added to the default scene. Clumps cannot be explicitly removed from the default scene.

See Also

`RwAddClumpToScene``()`
`RwDefaultScene``()`
`RwDestroyClump``()`
`RwForAllClumpsInScene``()`
`RwGetSceneNumClumps``()`
`RwRemoveLightFromScene``()`

`RwBool`
`RwRemoveHint``(RwClumpHints hints);`

Description

Removes a hint (or set of hints) from the current clump under construction. A clumps hints enable RenderWare to render a scene containing that clump more efficiently.

Arguments

`hints`	A bitfield representing a hint (or bitwise or of hints).

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

The clump hints are:

`rwCONTAINER`	The clump spatially contains other clumps.
`rwHS`	Action should be taken to prevent hidden surfaces from being visible when the clump is rendered.
`rwEDITABLE`	The clumps geometry is editable (its vertices can be moved and new vertices and polygons added).

This function can only be called within the context of an **`RwClumpBegin`**`()` … **`RwClumpEnd`**`()` block.

See Also

`RwAddHint``()`
`RwClumpBegin``()`
`RwClumpEnd``()`
`RwRemoveHintFromClump``()`
`RwSetHints``()`

```
RwClump *
RwRemoveHintFromClump(RwClump *clump, RwClumpHints hint);
```

Description

Removes a hint (or set of hints) from the clump.

Arguments

`clump`	Pointer to the clump.
`hint`	A bitfield representing a hint (or bitwise or of hints).

Return Value

The argument `clump` if successful, and `NULL` otherwise.

Comments

The clump hints are:

`rwCONTAINER`	The clump spatially contains other clumps.
`rwHS`	Action should be taken to prevent hidden surfaces from being visible when the clump is rendered.
`rwEDITABLE`	The clumps geometry is editable (its vertices can be moved and new vertices and polygons added).

See Also

RwAddHintToClump()
RwGetClumpHints()
RwRemoveHint()
RwSetClumpHints()

`RwLight *`
`RwRemoveLightFromScene``(RwLight *light);`

Description

Removes the light from its scene.

Arguments

`light` Pointer to the light.

Return Value

The argument `light` if successful, and `NULL` otherwise.

Comments

The light is added to the default scene. Lights cannot be explicitly removed from the default scene.

See Also

`RwAddLightToScene``()`
`RwDefaultScene``()`
`RwDestroyLight``()`
`RwForAllLightsInScene``()`
`RwGetSceneNumLights``()`
`RwRemoveClumpFromScene``()`

```
RwMaterial *
RwRemoveTextureModeFromMaterial(RwMaterial *material,
    RwTextureModes mode);
```

Description

Removes a texture mode (or modes) from the material. Texture modes permit fine grain control over the rendering of textures.

Arguments

`material`	Pointer to the material.
`mode`	A bitfield representing a texture mode (or bitwise or of modes).

Return Value

The argument `material` if successful, and `NULL` otherwise.

Comments

The texture modes are:

`rwLIT`	The texture will be lit according to the current light sampling type of the material (`rwFACET` or `rwVERTEX`).
`rwFORESHORTEN`	The texture will be foreshortened in a perspectively correct manner.
`rwFILTER`	A filter will be applied to the texture to reduce the effect of pixelation due to aliasing.

For further information see the Texture Modes section in Chapter 2: Data Types.

See Also

RwAddTextureModeToMaterial()
RwGetMaterialTextureModes()
RwRemoveTextureModeFromPolygon()
RwRemoveTextureModeFromSurface()
RwSetMaterialLightSampling()
RwSetMaterialTexture()
RwSetMaterialTextureModes()

```
RwPolygon3d *
RwRemoveTextureModeFromPolygon(RwPolygon3d *polygon,
    RwTextureModes mode);
```

Description

Removes a texture mode (or modes) from the polygons material. Texture modes permit fine grain control over the rendering of textures.

Arguments

`polygon`	Pointer to the polygon.
`mode`	A bitfield representing a texture mode (or bitwise or of modes).

Return Value

The argument `polygon` if successful, and `NULL` otherwise.

Comments

The texture modes are:

`rwLIT`	The texture will be lit according to the current light sampling type of the material (`rwFACET` or `rwVERTEX`).
`rwFORESHORTEN`	The texture will be foreshortened in a perspectively correct manner.
`rwFILTER`	A filter will be applied to the texture to reduce the effect of pixelation due to aliasing.

For further information see the Texture Modes section in Chapter 2: Data Types.

See Also

RwAddTextureModeToPolygon()
RwGetPolygonTextureModes()
RwRemoveTextureModeFromMaterial()
RwRemoveTextureModeFromSurface()
RwSetPolygonLightSampling()
RwSetPolygonTexture()
RwSetPolygonTextureModes()

RwBool
RwRemoveTextureModeFromSurface(RwTextureModes mode);

Description

Removes a texture mode (or modes) from the current material. Texture modes permit fine grain control over the rendering of textures.

Arguments

mode	A bitfield representing a texture mode (or bitwise or of modes).

Return Value

TRUE if successful, and FALSE otherwise.

Comments

The texture modes are:

rwLIT	The texture will be lit according to the current light sampling type of the material (rwFACET or rwVERTEX).
rwFORESHORTEN	The texture will be foreshortened in a perspectively correct manner.
rwFILTER	A filter will be applied to the texture to reduce the effect of pixelation due to aliasing.

For further information see the Texture Modes section in Chapter 2: Data Types.

This function can only be called within the context of an **RwModelBegin**() … **RwModelEnd**() block.

See Also

RwAddTextureModeToSurface()
RwModelBegin()
RwModelEnd()
RwRemoveTextureModeFromMaterial()
RwRemoveTextureModeFromPolygon()
RwSetSurfaceTexture()
RwSetSurfaceTextureModes()

```
RwUserDraw *
RwRemoveUserDrawFromClump(RwUserDraw *userdraw);
```

Description

Removes the user-draw from its clump.

Arguments

`userdraw` Pointer to the user-draw.

Return Value

The argument `userdraw` if successful, and `NULL` otherwise.

Comments

After being removed from a clump, the user-draw has no owning clump. To make use of such a user-draw, add it to a clump using **RwAddUserDrawToClump**().

See Also

RwAddUserDrawToClump()
RwCreateUserDraw()
RwDestroyUserDraw()
RwDuplicateUserDraw()
RwForAllUserDrawsInClump()
RwGetClumpNumUserDraws()

`RwClump *`
`RwRenderClump``(RwClump *clump);`

Description

Renders the clump into the current cameras image buffer.

Arguments

`clump`	Pointer to the clump.

Return Value

The argument `clump` if successful, and `NULL` otherwise

Comments

Clumps rendered by **`RwRenderClump()`** are illuminated by the lights in the default scene (if any).

Note that this function is not recursive, i.e., it only renders the specified clump and not its descendants.

This function can only be called in the context of an **`RwBeginCameraUpdate()`** … **`RwEndCameraUpdate()`** block.

See Also

`RwBeginCameraUpdate()`
`RwClearCameraViewport()`
`RwDefaultScene()`
`RwEndCameraUpdate()`
`RwRenderScene()`
`RwShowCameraImage()`

`RwScene *`
`RwRenderScene``(RwScene *scene);`

Description

Renders the scene into the current cameras image buffer.

Arguments

`scene` Pointer to the scene.

Return Value

The argument `scene` if successful, and `NULL` otherwise.

Comments

This function can only be called in the context of an **`RwBeginCameraUpdate`**`()` … **`RwEndCameraUpdate`**`()` block.

See Also

`RwBeginCameraUpdate``()`
`RwClearCameraViewport``()`
`RwEndCameraUpdate``()`
`RwRenderClump``()`
`RwShowCameraImage``()`

`RwCamera *`
`RwResetCamera``(RwCamera *camera);`

Description

Resets the camera to its initial position and orientation, at the origin, looking down the negative Z axis.

Arguments

`camera` Pointer to the camera.

Return Value

The argument `camera` if successful, and `NULL` otherwise.

Comments

RwResetCamera`()` performs the following (and only the following) actions:

Moves the cameras position to the origin of world space.

Points the camera down the negative Z axis of world space.

Points the cameras Look Up vector up the positive Y axis of world space.

Sets the cameras view window size to `CREAL(1.0)` by `CREAL(1.0)`.

Sets the cameras view offset to (`CREAL(0.0)`, `CREAL(0.0)`).

Damages the cameras entire viewport.

See Also

RwCreateCamera`()`
RwInvalidateCameraViewport`()`
RwPointCamera`()`
RwSetCameraLookAt`()`
RwSetCameraLookUp`()`
RwSetCameraViewOffset`()`
RwSetCameraViewwindow`()`
RwTransformCamera`()`
RwTransformCameraOrientation`()`

`RwPolygon3d *`
`RwReversePolygonFace``(RwPolygon3d *polygon);`

Description

Reverses the vertex ordering of the polygon.

Arguments

`polygon` Pointer to the polygon.

Return Value

The argument `polygon` if successful, and `NULL` otherwise.

Comments

`RwReversePolygonFace``()` modifies the facedness of a single polygon by reversing the order of the vertices in the polygons vertex list.

As this function modifies the geometry of the clump which owns the polygon, the clump is made editable (the `rwEDITABLE` hint is set) by this function.

See Also

None.

`RwCamera*`
`RwRevolveCamera``(RwCamera *camera, RwReal angle);`

Description

Rotates the camera about its Z axis.

Arguments

`camera`	Pointer to the camera.
`angle`	Angle of rotation (in degrees).

Return Value

The argument `camera` if successful, and `NULL` otherwise.

Comments

A positive value for `angle` will cause the camera to revolve clockwise.

See Also

`RwGetCameraLookRight``()`
`RwGetCameraLookUp``()`
`RwPanCamera``()`
`RwResetCamera``()`
`RwSetCameraLookUp``()`
`RwTiltCamera``()`
`RwTransformCameraOrientation``()`

```
RwBool
RwRotateCTM(RwReal rx, RwReal ry, RwReal rz, RwReal angle);
```

Description

Pre-concatenates a rotation matrix onto the CTM.

Arguments

`rx`	X component of the axis of rotation.
`ry`	Y component of the axis of rotation.
`rz`	Z component of the axis of rotation.
`angle`	Angle of rotation (in degrees).

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

This function can only be called within the context of an **`RwModelBegin`**`()` … **`RwModelEnd`**`()` block.

See Also

`RwIdentityCTM``()`
`RwModelBegin``()`
`RwModelEnd``()`
`RwRotateJointTM``()`
`RwRotateMatrix``()`
`RwRotateMatrixCos``()`
`RwScaleCTM``()`
`RwTransformCTM``()`
`RwTranslateCTM``()`

`RwBool`
`RwRotateJointTM``(RwReal rx, RwReal ry, RwReal rz, RwReal angle);`

Description

Pre-concatenates a rotation matrix onto the current joint transformation matrix.

Arguments

`rx`	X component of the axis of rotation.
`ry`	Y component of the axis of rotation.
`rz`	Z component of the axis of rotation.
`angle`	Angle of rotation (in degrees).

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

This function can only be called within the context of an **`RwModelBegin`**`()` … **`RwModelEnd`**`()` block.

See Also

`RwIdentityJointTM``()`
`RwModelBegin``()`
`RwModelEnd``()`
`RwRotateCTM``()`
`RwRotateMatrix``()`
`RwRotateMatrixCos``()`
`RwTransformJointTM``()`

```
RwMatrix4d *
RwRotateMatrix(RwMatrix4d *matrix, RwReal rx, RwReal ry, RwReal rz, RwReal
    angle, RwCombineOperation op);
```

Description

Builds a rotation matrix and applies it to `matrix`. The operation may be a pre-concatenation, post-concatenation, or replacement.

Arguments

`matrix`	Pointer to the matrix.
`rx`	X component of the axis of rotation.
`ry`	Y component of the axis of rotation.
`rz`	Z component of the axis of rotation.
`angle`	Angle of rotation (in degrees).
`op`	Combination operator.

Return Value

The argument `matrix` if successful, and `NULL` otherwise.

See Also

RwIdentityMatrix()
RwInvertMatrix()
RwMultiplyMatrix()
RwOrthoNormalizeMatrix()
RwQueryRotateMatrix()
RwRotateCTM()
RwRotateJointTM()
RwRotateMatrixCos()
RwScaleMatrix()
RwTransformMatrix()
RwTranslateMatrix()

```
RwMatrix4d *
RwRotateMatrixCos(RwMatrix4d *matrix, RwReal rx, RwReal ry,
    RwReal rz, RwReal cosangle, RwReal rotdir,
    RwCombineOperation op);
```

Description

Builds a rotation matrix and applies it to `matrix`. The angle of rotation is given by the cosine of the angle (`cosangle`) and a direction of rotation (`rotdir`). The operation may be a pre-concatenation, post-concatenation, or replacement.

Arguments

`matrix`	Pointer to the matrix.
`rx`	X component of the axis of rotation.
`ry`	Y component of the axis of rotation.
`rz`	Z component of the axis of rotation.
`cosangle`	Cosine of the angle of rotation
`rotdir`	Direction of rotation (a positive value specifies anti-clockwise rotation and a negative value specifies clockwise rotation).
`op`	Combination operator.

Return Value

The argument `matrix` if successful, and `NULL` otherwise.

Comments

This function should be used in preference to **RwRotateMatrix**() when the cosine of the angle of rotation is known. In such cases **RwRotateMatrixCos**() is more efficient than **RwRotateMatrix**().

See Also

RwIdentityMatrix()
RwInvertMatrix()
RwMultiplyMatrix()
RwOrthoNormalizeMatrix()
RwQueryRotateMatrix()
RwRotateCTM()
RwRotateJointTM()
RwRotateMatrix()
RwScaleMatrix()
RwTransformMatrix()
RwTranslateMatrix()

`RwBool`
`RwScaleCTM``(RwReal sx, RwReal sy, RwReal sz);`

Description

Pre-concatenates a scaling matrix onto the CTM.

Arguments

`sx`	Scale factor in the X axis.
`sy`	Scale factor in the Y axis.
`sz`	Scale factor in the Z axis.

Return Value

`TRUE` if successful, and `FALSE` otherwise

Comments

Note that if no scaling is to be applied `CREAL(1.0)` should be specified rather than `CREAL(0.0)`.

This function can only be called within the context of an **`RwModelBegin`**`()` … **`RwModelEnd`**`()` block.

See Also

`RwIdentityCTM``()`
`RwIdentityJointTM``()`
`RwModelBegin``()`
`RwModelEnd``()`
`RwRotateCTM``()`
`RwRotateJointTM``()`
`RwScaleMatrix``()`
`RwTransformCTM``()`
`RwTransformJointTM``()`
`RwTranslateCTM``()`

`RwMatrix4d *`
`RwScaleMatrix``(RwMatrix4d *matrix, RwReal sx, RwReal sy, RwReal sz, RwCombineOperation op);`

Description

Builds a scaling matrix and applies it to matrix. The operation may be a pre-concatenation, post-concatenation, or replacement.

Arguments

`matrix`	Pointer to the matrix.
`sx`	Scale factor in the X axis.
`sy`	Scale factor in the Y axis.
`sz`	Scale factor in the Z axis.
`op`	Combination operator.

Return Value

The argument `matrix` if successful, and `NULL` otherwise.

Comments

Note that if no scaling is to be applied `CREAL(1.0)` should be specified rather than `CREAL(0.0)`.

See Also

`RwIdentityMatrix``()`
`RwInvertMatrix``()`
`RwMultiplyMatrix``()`
`RwOrthoNormalizeMatrix``()`
`RwRotateMatrix``()`
`RwRotateMatrixCos``()`
`RwScaleCTM``()`
`RwTransformMatrix``()`
`RwTranslateMatrix``()`

```
RwV3d *
RwScaleVector(RwV3d *a, RwReal scale, RwV3d *b);
```

Description

Scales a vector.

Arguments

`a`	Pointer to the vector.
`scale`	Scale factor.
`b`	Pointer to the vector that will receive the result.

Return Value

The argument `b` if successful, and `NULL` otherwise.

See Also

RwAddVector()
RwCrossProduct()
RwDotProduct()
RwNormalize()
RwSubtractVector()
RwTransformVector()

```
RwMatrix4d *
RwScratchMatrix(void);
```

Description

Retrieves the current scratch matrix (the top matrix of the scratch matrix stack).

Arguments

None.

Return Value

A pointer to the scratch matrix.

Comments

The scratch matrix stack is a convenient source of temporary matrices for building transforms.

The matrix returned by **RwScratchMatrix**() must not be destroyed with **RwDestroyMatrix**(). The scratch matrix stack is destroyed by RenderWare when **RwClose**() is called.

See Also

RwClose()
RwCreateMatrix()
RwDestroyMatrix()
RwPopScratchMatrix()
RwPushScratchMatrix()

RwBool
RwSetAxisAlignment(RwAxisAlignment alignment);

Description

Sets the axis alignment type of the current clump under construction.

Arguments

`alignment` The axis alignment type.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

The following axis alignment types are supported:

`rwNOAXISALIGNMENT`	The clump is not axis aligned, it is unconstrained.
`rwALIGNAXISZORIENTX`	The clumps local Z axis is aligned with the Look At vector of the camera, but the orientation of the 2D projection of the clumps local X axis is preserved.
`rwALIGNAXISZORIENTY`	The clumps local Z axis is aligned with the Look At vector of the camera, but the orientation of the 2D projection of the clumps local Y axis is preserved.
`rwALIGNAXISXYZ`	The local X, Y and Z axes of the clump are aligned with the cameras Look Right, Look Up and Look At vectors respectively.

A clump that is axis aligned will be aligned with the view planes of all cameras used to view that clump.

This function can only be called within the context of an **RwClumpBegin**() … **RwClumpEnd**() block.

See Also

RwClumpBegin()
RwClumpEnd()
RwCreateSprite()
RwSetClumpAxisAlignment()

RwCamera *
RwSetCameraBackColor(RwCamera *camera, RwReal r, RwReal g, RwReal b);

Description

Sets the cameras background fill color.

Arguments

camera	Pointer to the camera.
r	Red component of the color in the range CREAL(0.0) to CREAL(1.0).
g	Green component of the color in the range CREAL(0.0) to CREAL(1.0).
b	Blue component of the color in the range CREAL(0.0) to CREAL(1.0).

Return Value

The argument camera if successful, and NULL otherwise.

Comments

This function is identical to **RwSetCameraBackColorStruct**() with the exception that it takes individual RwReals for the red, green and blue components of the color rather than an RwRGBColor structure as the color specification.

The background of the cameras viewport is cleared when **RwClearCameraViewport**() is called.

If the camera does not have a backdrop raster then the cameras entire viewport is filled with the background color. It the camera has a backdrop raster then those areas of the viewport outside the backdrop viewport rectangle will be filled with the background color.

See Also

RwClearCameraViewport()
RwGetCameraBackColor()
RwSetCameraBackColorStruct()
RwSetCameraBackdrop()
RwSetCameraBackdropViewportRect()

```
RwCamera *
RwSetCameraBackColorStruct(RwCamera *camera, RwRGBColor *color);
```

Description

Sets the cameras background fill color.

Arguments

`camera`	Pointer to the camera.
`color`	Pointer to the color.

Return Value

The argument `camera` if successful, and `NULL` otherwise.

Comments

This function is identical to **`RwSetCameraBackColor()`** with the exception that it takes an `RwRGBColor` structure as the color specification rather than individual `RwReals` for the red, green and blue components of the color.

The background of the cameras viewport is cleared when **`RwClearCameraViewport()`** is called.

If the camera does not have a backdrop raster then the cameras entire viewport is filled with the background color. It the camera has a backdrop raster then those areas of the viewport outside the backdrop viewport rectangle will be filled with the background color.

See Also

`RwClearCameraViewport()`
`RwGetCameraBackColor()`
`RwSetCameraBackColor()`
`RwSetCameraBackdrop()`
`RwSetCameraBackdropViewportRect()`

`RwCamera *`
`RwSetCameraBackdrop``(RwCamera *camera, RwRaster *raster);`

Description

Sets the cameras backdrop raster.

Arguments

`camera` Pointer to the camera.

`raster` Pointer to the raster.

Return Value

The argument `camera` if successful, and `NULL` otherwise.

Comments

To ensure that a camera has a visible backdrop it is not only necessary to set the cameras backdrop but also to set the rectangle of the viewport which will be filled by the backdrop. As the default backdrop viewport rectangle has a width and height of `0` the backdrop will not be visible unless a non-empty rectangle is specified. This is accomplished by **`RwSetCameraBackdropViewportRect`**`()`.

Areas of the viewport outside the backdrop viewport rectangle will be filled with the cameras background color.

The backdrop raster associated with a camera is not destroyed automatically when the camera is destroyed. The backdrop raster (if any) must by explicitly destroyed by **`RwDestroyRaster`**`()`.

See Also

`RwDestroyRaster``()`
`RwGetCameraBackdrop``()`
`RwGetCameraBackdropOffset``()`
`RwGetCameraBackdropViewportRect``()`
`RwSetCameraBackColor``()`
`RwSetCameraBackColorStruct``()`
`RwSetCameraBackdropOffset``()`
`RwSetCameraBackdropViewportRect``()`

```
RwCamera *
RwSetCameraBackdropOffset(RwCamera *camera, RwInt32 x, RwInt32 y);
```

Description

Sets the offset (from the origin of the cameras backdrop viewport rectangle) of the cameras backdrop raster.

Arguments

`camera`	Pointer to the camera.
`x`	The horizontal offset (in pixels).
`y`	The vertical offset (in pixels).

Return Value

The argument `camera` if successful, and `NULL` otherwise.

Comments

The X and Y offset (modulo the width and height of the backdrop) specify the pixel in the backdrop which will be mapped to the origin of the backdrop viewport rectangle. Therefore, the effect of increasing the X offset will be to scroll the backdrop to the left and increasing the Y offset will scroll the backdrop up.

See Also

RwGetCameraBackdrop()
RwGetCameraBackdropOffset()
RwGetCameraBackdropViewportRect()
RwSetCameraBackdrop()
RwSetCameraBackdropViewportRect()

```
RwCamera *
RwSetCameraBackdropViewportRect(RwCamera *camera, RwInt32 x,
    RwInt32 y, RwInt32 width, RwInt32 height);
```

Description

Sets the rectangular area of the viewport into which the cameras backdrop raster is rendered.

Arguments

`camera`	Pointer to the camera.
`x`	The X co-ordinate of rectangle (in viewport space co-ordinates).
`y`	The Y co-ordinate of rectangle (in viewport space co-ordinates).
`width`	The width of the rectangle (in viewport space units).
`height`	The height of the rectangle (in viewport space units).

Return Value

The argument `camera` if successful, and `NULL` otherwise.

Comments

The default backdrop viewport rectangle has a width and height of `0`. In order to ensure that the backdrop raster is visible it is necessary to set a viewport rectangle which has a non-zero width and height.

If the backdrop viewport rectangle is larger than the backdrop raster, the raster will be tiled to fill the rectangle. If the backdrop raster is larger than the viewport rectangle it will be cropped to fit the rectangle.

If the backdrop viewport rectangle does not fill the entire viewport, areas of the viewport outside the backdrop rectangle will be filled with the cameras background color.

The backdrop viewport rectangle is not automatically changed when the camera viewport is modified. **`RwSetCameraBackdropViewportRect()`** should be used to modify the backdrop viewport rectangle appropriately when the cameras viewport is modified.

See Also

`RwGetCameraBackColor()`
`RwGetCameraBackdrop()`
`RwGetCameraBackdropOffset()`
`RwGetCameraBackdropViewportRect()`
`RwSetCameraBackColor()`
`RwSetCameraBackColorStruct()`
`RwSetCameraBackdrop()`
`RwSetCameraBackdropOffset()`
`RwSetCameraViewport()`

```
RwCamera *
RwSetCameraData(RwCamera *camera, void *data);
```

Description

Sets the cameras user data pointer.

Arguments

`camera`	Pointer to the camera.
`data`	User data pointer.

Return Value

The argument `camera` if successful, and `NULL` otherwise.

See Also

`RwGetCameraData()`

```
RwCamera *
RwSetCameraFarClipping(RwCamera *camera, RwReal fard);
```

Description

Sets the distance from the camera position to the far (back) clipping plane.

Arguments

`camera`	Pointer to the camera.
`fard`	Distance (in world space units) from the camera to the far clipping plane.

Return Value

The argument `camera` if successful, and `NULL` otherwise.

Comments

The default far clipping distance is a large value which is dependent on the numeric type of the library.

See Also

RwCreateCamera()
RwGetCameraFarClipping()
RwGetCameraNearClipping()
RwSetCameraNearClipping()

```
RwCamera *
RwSetCameraLookAt(RwCamera *camera, RwReal x, RwReal y, RwReal z);
```

Description

Sets the cameras Look At vector, while maintaining its position.

Arguments

`camera`	Pointer to the camera.
`x`	X component of the vector.
`y`	Y component of the vector.
`z`	Z component of the vector.

Return Value

The argument `camera` if successful, and `NULL` otherwise.

See Also

RwCreateCamera()
RwGetCameraLookAt()
RwPanCamera()
RwPointCamera()
RwResetCamera()
RwSetCameraLookUp()
RwTiltCamera()
RwTransformCameraOrientation()

`RwCamera *`
`RwSetCameraLookUp``(RwCamera *camera, RwReal x, RwReal y, RwReal z);`

Description

Sets the cameras Look Up (or V) vector, while maintaining its position.

Arguments

`camera`	Pointer to the camera.
`x`	X component of the vector.
`y`	Y component of the vector.
`z`	Z component of the vector.

Return Value

The argument `camera` if successful, and `NULL` otherwise.

See Also

`RwCreateCamera``()`
`RwGetCameraLookUp``()`
`RwPointCamera``()`
`RwResetCamera``()`
`RwRevolveCamera``()`
`RwSetCameraLookAt``()`
`RwTiltCamera``()`
`RwTransformCameraOrientation``()`

`RwCamera *`
`RwSetCameraNearClipping``(RwCamera *camera, RwReal near);`

Description

Sets the distance from the camera position to the near (front) clipping plane.

Arguments

`camera`	Pointer to the camera.
`near`	Distance (in world space units) from the camera to the near clipping plane.

Return Value

The argument `camera` if successful, and `NULL` otherwise.

Comments

The default near clipping distance is `CREAL(0.05)`. The minimum clipping distance which can be specified is `CREAL(0.025)`.

See Also

`RwCreateCamera()`
`RwGetCameraFarClipping()`
`RwGetCameraNearClipping()`
`RwSetCameraFarClipping()`

```
RwCamera *
RwSetCameraPosition(RwCamera *camera, RwReal x, RwReal y, RwReal z);
```

Description

Sets the cameras position in world space co-ordinates.

Arguments

`camera`	Pointer to the camera.
`x`	X co-ordinate of the new camera position (in world space co-ordinates).
`y`	Y co-ordinate of the new camera position (in world space co-ordinates).
`z`	Z co-ordinate of the new camera position (in world space co-ordinates).

Return Value

The argument `camera` if successful, and `NULL` otherwise.

See Also

RwCreateCamera()
RwGetCameraPosition()
RwResetCamera()
RwTransformCamera()
RwVCMoveCamera()
RwWCMoveCamera()

```
RwCamera *
RwSetCameraProjection(RwCamera *camera, RwCameraProjection model);
```

Description

Sets the cameras projection model.

Arguments

`camera`	Pointer to the camera.
`model`	Camera projection model.

Return Value

The argument `camera` if successful, and `NULL` otherwise.

Comments

The projection types are:

`rwPARALLEL`	Parallel projection.
`rwPERSPECTIVE`	Perspective projection.

See Also

RwCreateCamera()
RwGetCameraProjection()

`RwCamera *`
`RwSetCameraViewOffset``(RwCamera *camera, RwReal x, RwReal y);`

Description

Sets the cameras view offset, thereby shearing the view volume.

Arguments

`camera`	Pointer to the camera.
`x`	View offset displacement in the direction of the cameras "Look Right" vector (in world space units).
`y`	View offset displacement in the direction of the cameras "Look Up" vector (in world space units).

Return Value

The argument `camera` if successful, and `NULL` otherwise.

Comments

The X and Y offsets are measured in world space units in a plane passing through the camera position and parallel to the view window, in the directions of the cameras "Look Right" and "Look Up" vectors respectively. For a perspective view, this moves the apex of the view pyramid whilst its edges remain fixed to the corners of the view window. For a parallel view, this shears the view parallelepiped whilst its edges remain fixed to the corners of the view window.

Successive calls to **`RwSetCameraViewOffset`**`()` specify the offset as absolute values from the initial unsheared camera position; successive offsets are not accumulated as relative displacements.

See Also

`RwGetCameraViewOffset``()`
`RwResetCamera``()`

```
RwCamera *
RwSetCameraViewport(RwCamera *camera, RwInt32 x, RwInt32 y,
     RwInt32 width, RwInt32 height);
```

Description

Defines a rectangular area of the display (screen or window) onto which the cameras view window is mapped.

Arguments

`camera`	Pointer to the camera.
`x`	X co-ordinate of the viewport origin (in device space co-ordinates).
`y`	Y co-ordinate of the viewport origin (in device space co-ordinates).
`width`	Width of the viewport (in device space units).
`height`	Height of the viewport (in device space units).

Return Value

The argument `camera` if successful, and `NULL` otherwise.

Comments

The viewport origin is the top left of the viewport.

See Also

RwCreateCamera()
RwGetCameraViewport()
RwSetCameraBackdropViewportRect()
RwSetCameraViewwindow()

`RwCamera *`
`RwSetCameraViewwindow``(RwCamera *camera, RwReal width, RwReal height);`

Description

Sets the relative size of the view window in the view plane. Larger values give a wider field of view, smaller values a narrower field of view.

Arguments

`camera`	Pointer to the camera.
`width`	Width of the view window (in world space units).
`height`	Height of the view window (in world space units).

Return Value

The argument `camera` if successful, and `NULL` otherwise.

See Also

`RwCreateCamera``()`
`RwGetCameraViewwindow``()`
`RwResetCamera``()`
`RwSetCameraViewport``()`

```
RwClump *
RwSetClumpAxisAlignment(RwClump *clump, RwAxisAlignment alignment);
```

Description

Sets the axis alignment type of the clump.

Arguments

`clump`	Pointer to the clump.
`alignment`	The axis alignment type.

Return Value

The argument `clump` if successful, and `NULL` otherwise.

Comments

The axis alignment types are:

`rwNOAXISALIGNMENT`	The clump is not axis aligned, it is unconstrained.
`rwALIGNAXISZORIENTX`	The clumps local Z axis is aligned with the Look At vector of the camera, but the orientation of the 2D projection of the clumps local X axis is preserved.
`rwALIGNAXISZORIENTY`	The clumps local Z axis is aligned with the Look At vector of the camera, but the orientation of the 2D projection of the clumps local Y axis is preserved.
`rwALIGNAXISXYZ`	The local X, Y and Z axes of the clump are aligned with the cameras Look Right, Look Up and Look At vectors respectively.

A clump that is axis aligned will be aligned with the view planes of all cameras used to view that clump.

See Also

RwCreateSprite()
RwGetClumpAxisAlignment()
RwSetAxisAlignment()

```
RwClump *
RwSetClumpData(RwClump *clump, void *data);
```

Description

Sets the clumps user data pointer.

Arguments

`clump`	Pointer to the clump.
`data`	User data pointer.

Return Value

The argument `clump` if successful, and `NULL` otherwise.

See Also

RwGetClumpData()

`RwClump *`
`RwSetClumpHints``(RwClump *clump, RwClumpHints hints);`

Description

Sets the hints of the clump to those given. A clumps hints enable RenderWare to render a scene containing that clump more efficiently.

Arguments

`clump`	Pointer to the clump.
`hints`	A bitfield representing a hint (or bitwise or of hints).

Return Value

The argument `clump` if successful, and `NULL` otherwise.

Comments

Unlike **`RwAddHintToClump`**`()`, which simply adds one or more hints to a clumps set of hints, **`RwSetClumpHints`**`()` replaces the entire set of hints of a clump with those specified.

The clump hints are:

`rwCONTAINER`	The clump spatially contains other clumps.
`rwHS`	Action should be taken to prevent hidden surfaces from being visible when the clump is rendered.
`rwEDITABLE`	The clumps geometry is editable (its vertices can be moved and new vertices and polygons added).

See Also

`RwAddHintToClump``()`
`RwGetClumpHints``()`
`RwRemoveHintFromClump``()`
`RwSetHints``()`

`RwClump *`
`RwSetClumpState``(RwClump *clump, RwState state);`

Description

Sets the clumps on/off state.

Arguments

`clump`	Pointer to the clump.
`state`	The clump state.

Return Value

The argument `clump` if successful, and `NULL` otherwise.

Comments

The clumps state determines whether the clump will be considered as a candidate for rendering and picking.

The clump states are:

`rwON`	The clump will be a candidate for rendering and picking.
`rwOFF`	The clump will not be a candidate for rendering and picking.

A state of `rwON` should be interpreted as making the clump a candidate for rendering and picking. Such a clump will not appear if it lies outside the view volume and it will not be picked unless one of its polygons is the foremost under the pick position.

The state affects only the clump to which it is applied and not to the clumps children. Thus, to prevent a single clump in a hierarchy from being rendered it is preferable to modify the clumps state rather than to remove it from a scene with **`RwRemoveClumpFromScene`**`()`.

See Also

`RwAddClumpToScene``()`
`RwDestroyClump``()`
`RwGetClumpState``()`
`RwRemoveClumpFromScene``()`

```
RwClump *
RwSetClumpTag(RwClump *clump, RwInt32 tag);
```

Description

Assigns the integer tag to the clump.

Arguments

`clump`	Pointer to the clump.
`tag`	Integer tag value to set.

Return Value

The argument `clump` if successful, `NULL` otherwise.

See Also

RwGetClumpTag()
RwSetPolygonTag()
RwSetTag()

`RwClump *`
`RwSetClumpVertex``(RwClump *clump, RwInt32 index, RwV3d *coords);`

Description

Sets the object space position of the vertex which belongs to clump and has the vertex index `index`.

Arguments

`clump`	Pointer to the clump.
`index`	The vertex index.
`coords`	Pointer to the point that specifies the vertexs position (in object space co-ordinates).

Return Value

The argument `clump` if successful, and `NULL` otherwise.

Comments

The vertex index is an integer greater than `0` and less than or equal to the number of vertices that belong to the clump.

As this function modifies the geometry of the clump, the clump is made editable (the `rwEDITABLE` hint is set) by this function.

See Also

`RwGetClumpVertex``()`
`RwSetClumpVertexNormal``()`
`RwSetClumpVertexUV``()`
`RwSetClumpVertices``()`

```
RwClump *
RwSetClumpVertexNormal(RwClump *clump, RwInt32 index, RwV3d *normal);
```

Description

Sets the unit shading normal at the vertex which belongs to clump and has the vertex index `index`.

Arguments

`clump`	Pointer to the clump.
`index`	The vertex index.
`normal`	Pointer to the vector that specifies the unit shading normal.

Return Value

The argument `clump` if successful, and `NULL` otherwise.

Comments

By default, RenderWare automatically calculates unit shading normals. When a clump is read, created or edited the unit shading normals are recalculated. However, In addition to setting the normal, **`RwSetClumpVertexNormal`**`()` suspends automatic recalculation of the normal at the specified vertex.

To enable automatic recalculation of the normal at a vertex use **`RwCalculateClumpVertexNormal`**`()`.

See Also

`RwCalculateClumpVertexNormal``()`
`RwGetClumpVertexNormal``()`
`RwSetClumpVertex``()`
`RwSetClumpVertexUV``()`

`RwClump *`
`RwSetClumpVertexUV``(RwClump *clump, RwInt32 index, RwReal u, RwReal v);`

Description

Sets the texture (U, V) co-ordinates of the vertex which belongs to clump and has the vertex index `index`.

Arguments

`clump`	Pointer to the clump.
`index`	The vertex index.
`u`	U co-ordinate of the vertexs texture co-ordinates.
`v`	V co-ordinate of the vertexs texture co-ordinates.

Return Value

The argument `clump` if successful, and `NULL` otherwise.

Comments

The vertex index is an integer greater than `0` and less than or equal to the number of vertices that belong to the clump.

See Also

`RwCubicTexturizeClump``()`
`RwEnvMapClump``()`
`RwGetClumpVertexUV``()`
`RwSetClumpVertex``()`
`RwSetClumpVertexNormal``()`
`RwSetMaterialTexture``()`
`RwSetPolygonTexture``()`
`RwSetPolygonUV``()`
`RwSphericalTexturizeClump``()`

```
RwClump *
RwSetClumpVertices(RwClump *clump, RwInt32 *vlist, RwV3d *coords, RwInt32
    nverts);
```

Description

Sets the object space co-ordinates of one or more vertices of the clump.

Arguments

`clump`	Pointer to the clump.
`vlist`	Pointer to an array of vertex indices.
`coords`	Pointer to an array of vertex co-ordinates (in object space co-ordinates).
`nverts`	Number of vertices whose co-ordinates will be modified.

Return Value

The argument `clump` if successful, and `NULL` otherwise.

Comments

Each element of the indices array gives the vertex index of a vertex to modify, the corresponding element of the coords array gives the new co-ordinates of that vertex.

The arrays `indices` and `coords` must be of length `nverts`. There must be at least `nverts` vertices in the clump.

It is considerably more efficient to use a single call to **`RwSetClumpVertices`**`()` then multiple calls to **`RwSetClumpVertex`**`()` when modifying two or more vertices of a clump.

As this function modifies the geometry of the clump, the clump is made editable (the `rwEDITABLE` hint is set) by this function.

For 16-bit applications accessing the RenderWare DLL the vertex index list pointed to by `vlist` must be declared as an array of `RwInt32`s and not `int`s.

See Also

`RwGetClumpVertex``()`
`RwSetClumpVertex``()`
`RwSetClumpVertexNormal``()`
`RwSetClumpVertexUV``()`

```
void
RwSetDebugAssertionState(RwState state);
```

Description

Enables or disables the generation of assertion messages.

Arguments

`state`	The enable/disable flag.

Return Value

None.

Comments

The assertion message states are:

`rwON`	Assertion messages are enabled.
`rwOFF`	Assertion messages are disabled.

See Also

RwGetDebugAssertionState()
RwSetDebugMessageState()
RwSetDebugOutputState()
RwSetDebugScriptState()
RwSetDebugTraceState()

```
void
RwSetDebugMessageState(RwState state);
```

Description

Enables or disables the generation of miscellaneous messages.

Arguments

`state`	The enable/disable flag.

Return Value

None.

Comments

The miscellaneous message states are:

`rwON`	Miscellaneous messages are enabled.
`rwOFF`	Miscellaneous messages are disabled.

See Also

RwGetDebugMessageState()
RwSetDebugAssertionState()
RwSetDebugOutputState()
RwSetDebugScriptState()
RwSetDebugTraceState()

```
void
RwSetDebugOutputState(RwState state);
```

Description

Enables or disables the generation of all types of debugging messages.

Arguments

`state`	The debugging state.

Return Value

None.

Comments

This function is equivalent to calling **`RwSetDebugAssertionState()`**, **`RwSetDebugMessageState()`**, **`RwSetDebugScriptState()`**`RwSetDebugScriptState`and **`RwSetDebugTraceState()`** with the argument `state`.

The message states are:

`rwON`	Messages are enabled.
`rwOFF`	Messages are disabled.

See Also

`RwGetDebugAssertionState()`
`RwGetDebugMessageState()`
`RwGetDebugScriptState()`
`RwSetDebugAssertionState()`
`RwSetDebugMessageState()`
`RwSetDebugScriptState()`
`RwSetDebugTraceState()`

```
void
RwSetDebugScriptState(RwState state);
```

Description

Enables or disables the generation of script trace messages.

Arguments

`state`	The enable/disable flag.

Return Value

None.

Comments

The script trace message states are:

`rwON`	Script trace messages are enabled.
`rwOFF`	Script trace messages are disabled.

See Also

RwGetDebugScriptState()
RwSetDebugAssertionState()
RwSetDebugMessageState()
RwSetDebugOutputState()
RwSetDebugTraceState()

```
void
RwSetDebugSeverity(RwDebugSeverity severity);
```

Description

Sets the minimum severity level for the reporting of debugging messages.

Arguments

`severity` The minimum severity level.

Return Value

None.

Comments

The debug message severity levels are:

`rwINFORM`	Control flow annotations, non-fatal exceptions and fatal exceptions are all enabled.
`rwWARNING`	Non-fatal exceptions and fatal exceptions are enabled.
`rwERROR`	Fatal exceptions are enabled.

See Also

`RwGetDebugSeverity``()`

RwBool
RwSetDebugStream(FILE *stream);

Description

Sets the current debugging stream.

Arguments

stream	File pointer.

Return Value

TRUE if successful, and FALSE otherwise.

Comments

This function is useful in situations where a file pointer is available but a file name is not, i.e., files that have been opened previously or standard streams such as stderr.

See Also

RwCloseDebugStream()
RwOpenDebugStream()

```
void
RwSetDebugTraceState(RwState state);
```

Description

Enables or disables the generation of API function trace messages.

Arguments

`state`	The enable/disable flag.

Return Value

None.

Comments

The API function trace message states are:

`rwON`	API function trace messages are enabled.
`rwOFF`	API function trace messages are disabled.

See Also

RwGetDebugTraceState()
RwSetDebugAssertionState()
RwSetDebugMessageState()
RwSetDebugOutputState()
RwSetDebugScriptState()

RwBool
RwSetHints(RwClumpHints hints);

Description

Sets the hints of the current clump under construction to those given. A clumps hints enable RenderWare to render a scene containing that clump more efficiently.

Arguments

`hints`	A bitfield representing a hint (or bitwise or of hints).

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

The clump hints are:

`rwCONTAINER`	The clump spatially contains other clumps.
`rwHS`	Action should be taken to prevent hidden surfaces from being visible when the clump is rendered.
`rwEDITABLE`	The clumps geometry is editable (its vertices can be moved and new vertices and polygons added).

Unlike **RwAddHint()**, which simply adds one or more hints to the current clumps set of hints, **RwSetHints()** replaces the current clumps entire set of hints with those specified.

This function can only be called within the context of an **RwClumpBegin()** … **RwClumpEnd()** block.

See Also

RwAddHint()
RwClumpBegin()
RwClumpEnd()
RwRemoveHint()
RwSetClumpHints()

`RwLight *`
`RwSetLightBrightness``(RwLight *light, RwReal brightness);`

Description

Sets the lights brightness.

Arguments

`light` Pointer to the light.

`brightness` Brightness, in the range `CREAL(0.0)` to `CREAL(1.0)`.

Return Value

The argument `light` if successful, and `NULL` otherwise.

Comments

This function is identical to **`RwSetLightColor`**`(light, brightness, brightness, brightness)`. For a scene using colored lights, **`RwSetLightColor`**`()` should be used to control the intensity of the light.

See Also

`RwCreateLight``()`
`RwGetLightBrightness``()`
`RwSetLightColor()`
`RwSetLightColorStruct()`
`RwGetLightColor()`

`RwLight *`
`RwSetLightColor``(RwLight *light, RwReal r, RwReal g, RwReal b);`

Description

Sets the color of a light.

Arguments

`light`	Pointer to the light.
`r`	Red component of the color in the range `CREAL(0.0)` to `CREAL(1.0)`.
`g`	Green component of the color in the range `CREAL(0.0)` to `CREAL(1.0)`.
`b`	Blue component of the color in the range `CREAL(0.0)` to `CREAL(1.0)`.

Return Value

The argument `light` if successful, and `NULL` otherwise.

Comments

The value returned by **`RwGetLightBrightness()`** for a light whose color has been set with **`RwSetLightColor()`** is the average intensity of the red, green and blue channels of the lights color.

This function is identical to **`RwSetLightColorStruct()`** with the exception that it takes individual `RwReal`s for the red, green and blue components of the color rather than an `RwRGBColor` structure as the color specification.

In RenderWare V1.4, colored light sources are only available when performing 16-bit rendering. Under 8-bit rendering **`RwGetLightColor()`**, **`RwSetLightColor()`** and **`RwSetLightColorStruct()`** are still available to the API, however, light sources will always be white.

See Also

`RwSetLightBrightness()`
`RwSetLightColorStruct()`
`RwGetLightColor()`

```
RwLight *
RwSetLightColorStruct(RwLight *light, RwRGBColor *color);
```

Description

Sets the color of a light.

Arguments

`light`	Pointer to the light.
`color`	Pointer to the color.

Return Value

The argument `light` if successful, and `NULL` otherwise.

Comments

The value returned by **`RwGetLightBrightness()`** for a light whose color has been set with **`RwSetLightColorStruct()`** is the average intensity of the red, green and blue channels of the lights color.

This function is identical to **`RwSetLightColor()`** with the exception that it takes an `RwRGBColor` structure as the color specification rather than individual `RwReal`s for the red, green and blue components of the color.

In RenderWare V1.4, colored light sources are only available when performing 16-bit rendering. Under 8-bit rendering **`RwGetLightColor()`**, **`RwSetLightColor()`** and **`RwSetLightColorStruct()`** are still available to the API, however, light sources will always be white.

See Also

`RwSetLightBrightness()`
`RwSetLightColor()`
`RwGetLightColor()`

```
RwLight *
RwSetLightConeAngle(RwLight *light, RwReal angle);
```

Description

Sets the angle at which a conical light illuminates objects (measured from the direction vector of the light).

Arguments

`light`	Pointer to the light.
`angle`	Cone angle (in degrees).

Return Value

The argument `light` if successful, and `NULL` otherwise.

Comments

This function is only valid for conical lights (those lights created with the light type `rwCONICAL`).

See Also

RwCreateLight()
RwGetLightConeAngle()
RwGetLightType()

```
RwLight *
RwSetLightData(RwLight *light, void *data);
```

Description

Sets the lights user data pointer.

Arguments

`light`	Pointer to the light.
`data`	User data pointer.

Return Value

The argument `light` if successful, and `NULL` otherwise.

See Also

RwGetLightData()

`RwLight *`
`RwSetLightPosition``(RwLight *light, RwReal x, RwReal y, RwReal z);`

Description

Sets the position of a point or conical light source in world space co-ordinates.

Arguments

`light`	Pointer to the light.
`x`	X co-ordinate of the light position (in world space co-ordinates).
`y`	Y co-ordinate of the light position (in world space co-ordinates).
`z`	Z co-ordinate of the light position (in world space co-ordinates).

Return Value

The argument `light` if successful, and `NULL` otherwise.

Comments

This function is only valid for point and conical lights (those lights created with the light types `rwPOINT` or `rwCONICAL`).

See Also

`RwCreateLight``()`
`RwGetLightPosition``()`
`RwGetLightType``()`
`RwTransformLight``()`

```
RwLight *
RwSetLightState(RwLight *light, RwState state);
```

Description

Turns the light on or off.

Arguments

`light`	Pointer to the light.
`state`	The light state.

Return Value

The argument `light` if successful, and `NULL` otherwise.

Comments

The light states are:

`rwON`	The light is on.
`rwOFF`	The light is off.

See Also

`RwGetLightState``()`

`RwLight *`
`RwSetLightVector``(RwLight *light, RwReal x, RwReal y, RwReal z);`

Description

Sets the illumination vector of a directional or conical light source.

Arguments

`light`	Pointer to the light.
`x`	X component of the light vector.
`y`	Y component of the light vector.
`z`	Z component of the light vector.

Return Value

The argument `light` if successful, and `NULL` otherwise.

Comments

This function is only valid for directional and conical lights (those lights created with the light types `rwDIRECTIONAL` or `rwCONICAL`).

See Also

`RwCreateLight``()`
`RwGetLightType``()`
`RwGetLightVector``()`
`RwTransformLight``()`

```
RwMaterial *
RwSetMaterialAmbient(RwMaterial *material, RwReal ka);
```

Description

Sets the materials ambient reflection coefficient.

Arguments

`material`	Pointer to the material.
`ka`	Ambient reflection coefficient in the range `CREAL(0.0)` to `CREAL(1.0)`.

Return Value

The argument `material` if successful, and `NULL` otherwise.

See Also

RwGetMaterialAmbient()
RwSetMaterialDiffuse()
RwSetMaterialSpecular()
RwSetMaterialSurface()
RwSetPolygonAmbient()
RwSetSurfaceAmbient()

```
RwMaterial *
RwSetMaterialColor(RwMaterial *material, RwReal r, RwReal g,
    RwReal b);
```

Description

Sets the materials color.

Arguments

`material`	Pointer to the material.
`r`	Red component of the color, in the range `CREAL(0.0)` to `CREAL(1.0)`.
`g`	Green component of the color, in the range `CREAL(0.0)` to `CREAL(1.0)`.
`b`	Blue component of the color, in the range `CREAL(0.0)` to `CREAL(1.0)`.

Return Value

The argument `material` if successful, and `NULL` otherwise.

Comments

This function is identical to **`RwSetMaterialColorStruct`**`()` with the exception that it takes individual `RwReal`s for the red, green and blue components of the color rather than an `RwRGBColor` structure as the color specification.

See Also

`RwGetMaterialColor``()`
`RwSetMaterialColorStruct``()`
`RwSetPolygonColor``()`
`RwSetPolygonColorStruct``()`
`RwSetSurfaceColor``()`

`RwMaterial *`
`RwSetMaterialColorStruct``(RwMaterial *material, RwRGBColor *color);`

Description

Sets the materials color.

Arguments

`material` Pointer to the material.

`color` Pointer to the color.

Return Value

The argument `material` if successful, and `NULL` otherwise.

Comments

This function is identical to **`RwSetMaterialColor`**`()` with the exception that it takes an `RwRGBColor` structure as the color specification rather than individual `RwReal`s for the red, green and blue components of the color. This can be useful as a call-back in **`RwForAll`**`…()` functions.

See Also

`RwGetMaterialColor``()`
`RwSetMaterialColor``()`
`RwSetPolygonColor``()`
`RwSetPolygonColorStruct``()`
`RwSetSurfaceColor``()`

```
RwMaterial *
RwSetMaterialDiffuse(RwMaterial *material, RwReal kd);
```

Description

Sets the materials diffuse reflection coefficient.

Arguments

`material`	Pointer to the material.
`kd`	Diffuse reflection coefficient in the range `CREAL(0.0)` to `CREAL(1.0)`.

Return Value

The argument `material` if successful, and `NULL` otherwise.

See Also

RwGetMaterialDiffuse()
RwSetMaterialAmbient()
RwSetMaterialSpecular()
RwSetMaterialSurface()
RwSetPolygonDiffuse()
RwSetSurfaceDiffuse()

```
RwMaterial *
RwSetMaterialGeometrySampling(RwMaterial *material, RwGeometrySampling type);
```

Description

Sets the materials geometry sampling type.

Arguments

`material`	Pointer to the material.
`type`	The geometry sampling type.

Return Value

The argument `material` if successful, and `NULL` otherwise.

Comments

The geometry sampling types are:

`rwPOINTCLOUD`	Render geometry as a cloud of points.
`rwWIREFRAME`	Render geometry as a wireframe of polygon edges.
`rwSOLID`	Render geometry as a solid bounded by filled polygons.

See Also

RwGetMaterialGeometrySampling()
RwSetMaterialLightSampling()
RwSetPolygonGeometrySampling()
RwSetSurfaceGeometrySampling()

```
RwMaterial *
RwSetMaterialLightSampling(RwMaterial *material,
      RwLightSampling type);
```

Description

Sets the materials light sampling type.

Arguments

`material`	Pointer to the material.
`type`	The light sampling type.

Return Value

The argument `material` if successful, and `NULL` otherwise.

Comments

The light sampling types are:

`rwFACET`	Flat shading.
`rwVERTEX`	Smooth shading.

See Also

RwGetMaterialLightSampling()
RwSetMaterialGeometrySampling()
RwSetPolygonLightSampling()
RwSetSurfaceLightSampling()

`RwMaterial *`
`RwSetMaterialOpacity``(RwMaterial *material, RwReal opacity);`

Description

Sets the materials opacity.

Arguments

`material`	Pointer to the material.
`opacity`	Opacity in the range `CREAL(0.0)` to `CREAL(1.0)`.

Return Value

The argument `material` if successful, and `NULL` otherwise.

Comments

An opacity of `CREAL(1.0)` yields an entirely opaque material. An opacity of `CREAL(0.0)` yields an entirely transparent material.

See Also

`RwGetMaterialOpacity``()`
`RwSetPolygonOpacity``()`
`RwSetSurfaceOpacity``()`

```
RwMaterial *
RwSetMaterialSpecular(RwMaterial *material, RwReal ks);
```

Description

Sets the materials specular reflection coefficient.

Arguments

`material`	Pointer to the material.
`ks`	Specular reflection coefficient in the range `CREAL(0.0)` to `CREAL(1.0)`.

Return Value

The argument `material` if successful, and `NULL` otherwise.

See Also

RwGetMaterialSpecular()
RwSetMaterialAmbient()
RwSetMaterialDiffuse()
RwSetMaterialSurface()
RwSetPolygonSpecular()
RwSetSurfaceSpecular()

`RwMaterial *`
`RwSetMaterialSurface``(RwMaterial *material, RwReal ka, RwReal kd, RwReal ks);`

Description

Sets the materials surface attributes (ambient, diffuse, and specular reflection coefficients).

Arguments

`material`	Pointer to the material.
`ka`	Ambient reflection coefficient in the range `CREAL(0.0)` to `CREAL(1.0)`.
`kd`	Diffuse reflection coefficient in the range `CREAL(0.0)` to `CREAL(1.0)`.
`ks`	Specular reflection coefficient in the range `CREAL(0.0)` to `CREAL(1.0)`.

Return Value

The argument `material` if successful, and `NULL` otherwise.

See Also

`RwGetMaterialAmbient``()`
`RwGetMaterialDiffuse``()`
`RwGetMaterialSpecular``()`
`RwSetMaterialAmbient``()`
`RwSetMaterialDiffuse``()`
`RwSetMaterialSpecular``()`
`RwSetPolygonSurface``()`
`RwSetSurface``()`

```
RwMaterial *
RwSetMaterialTexture(RwMaterial *material, RwTexture *texture);
```

Description

Sets the materials texture.

Arguments

`material`	Pointer to the material.
`texture`	Pointer to the texture.

Return Value

The argument `material` if successful, and `NULL` otherwise.

Comments

`NULL` may be passed as the second argument to remove the materials texture.

See Also

RwCreateTexture()
RwFindNamedTexture()
RwGetMaterialTexture()
RwGetNamedTexture()
RwReadNamedTexture()
RwReadTexture()
RwSetPolygonTexture()
RwSetSurfaceTexture()

`RwMaterial *`
`RwSetMaterialTextureModes``(RwMaterial *material, RwTextureModes modes);`

Description

Sets the texture modes of the material. Texture modes permit fine grain control over the rendering of textures.

Arguments

`material`	Pointer to the material.
`modes`	A bitfield representing a texture mode (or bitwise or of modes).

Return Value

The argument `material` if successful, and `NULL` otherwise.

Comments

The texture modes are:

`rwLIT`	The texture will be lit according to the current light sampling type of the material (`rwFACET` or `rwVERTEX`).
`rwFORESHORTEN`	The texture will be foreshortened in a perspectively correct manner.
`rwFILTER`	A filter will be applied to the texture to reduce the effect of pixelation due to aliasing.

For further information see the Texture Modes section in Chapter 2: Data Types.

Unlike **`RwAddTextureModeToMaterial`**`()`, which simply adds one or more texture modes to a materials set of modes, **`RwSetMaterialTextureModes`**`()` replaces a materials entire set of modes with those specified.

See Also

`RwAddTextureModeToMaterial``()`
`RwGetMaterialTextureModes``()`
`RwRemoveTextureModeFromMaterial``()`
`RwSetMaterialLightSampling``()`
`RwSetMaterialTexture``()`
`RwSetPolygonTextureModes``()`
`RwSetSurfaceTextureModes``()`

```
RwMatrix4d *
RwSetMatrixElement(RwMatrix4d *matrix, RwInt32 row, RwInt32 col, RwReal
    element);
```

Description

Sets an individual element of the matrix.

Arguments

`matrix`	Pointer to the matrix.
`row`	Row index in the range `0` to `3`.
`col`	Column index in the range `0` to `3`.
`element`	New matrix element.

Return Value

The argument `matrix` if successful, and `NULL` otherwise.

See Also

`RwGetMatrixElement``()`
`RwSetMatrixElement``()`
`RwSetMatrixElements``()`

```
RwMatrix4d *
RwSetMatrixElements(RwMatrix4d *matrix, RwReal elements[4][4]);
```

Description

Sets the elements of a matrix from a four by four array of `RwReal`s. The first four entries of the array are copied into the top row of the matrix.

Arguments

`matrix`	Pointer to the matrix.
`elements`	Pointer to a four by four array of `RwReal`s holding the values to be copied.

Return Value

The argument `matrix` if successful, and `NULL` otherwise.

Comments

By convention a matrix is taken to transform a row vector by post multiplication.

The final column of the array will normally be [`CREAL(0.0)`, `CREAL(0.0)`, `CREAL(0.0)`, `CREAL(1.0)`].

See Also

`RwGetMatrixElement()`
`RwGetMatrixElements()`
`RwSetMatrixElement()`

```
RwInt32
RwSetPaletteEntries(RwInt32 start, RwInt32 length,
    RwPaletteEntry *palette, RwPaletteOptions options);
```

Description

Sets `length` palette entries of the current RenderWare palette starting at entry `start`.

Arguments

`start`	First palette entry to set.
`length`	Number of entries to set.
`palette`	Pointer to an array of `RwPaletteEntry`s.
`options`	A bitfield representing a palette processing operation.

Return Value

The argument `length` if successful, and `0` otherwise.

Comments

The supported palette options are as follows:

- `rwGAMMAPALETTE` Gamma correct the palette.

Note: Not all platforms allow the application to overwrite all of the palette. Under Windows for example the first 10 and the last 10 entries are reserved by the system. An attempt to set these system entries will result in an error being returned by this function.

The function **`RwGetDeviceInfo`**`()` can be used to determine the first and last palette entries available to the application.

See Also

`RwGetDeviceInfo``()`
`RwGetPaletteEntries``()`

```
RwPolygon3d *
RwSetPolygonAmbient(RwPolygon3d *polygon, RwReal ka);
```

Description

Sets the ambient reflection coefficient of the polygons material.

Arguments

`polygon`	Pointer to the polygon.
`ka`	Ambient reflection coefficient in the range `CREAL(0.0)` to `CREAL(1.0)`.

Return Value

The argument `polygon` if successful, and `NULL` otherwise.

Comments

RenderWare optimizes memory usage by sharing materials across multiple polygons. Setting a polygon material property with this function will cause the polygon to have its own copy of the material, not shared by any other polygons. Unless this is the desired effect, it is more memory efficient to use the corresponding material function **RwSetMaterialAmbient()** to change the underlying polygon material. This change will then be propagated to all polygons which use the material. The following line of code demonstrates how this is achieved:

```
RwSetMaterialAmbient(RwGetPolygonMaterial(polygon), ka);
```

See Also

RwGetPolygonAmbient()
RwSetMaterialAmbient()
RwSetPolygonDiffuse()
RwSetPolygonSpecular()
RwSetPolygonSurface()
RwSetSurfaceAmbient()

```
RwPolygon3d *
RwSetPolygonColor(RwPolygon3d *polygon,
          RwReal r, RwReal g, RwReal b);
```

Description

Sets the color of the polygons material.

Arguments

`polygon`	Pointer to the polygon.
`r`	Red component of the color, in the range `CREAL(0.0)` to `CREAL(1.0)`.
`g`	Green component of the color, in the range `CREAL(0.0)` to `CREAL(1.0)`.
`b`	Blue component of the color, in the range `CREAL(0.0)` to `CREAL(1.0)`.

Return Value

The argument `polygon` if successful, and `NULL` otherwise.

Comments

This function is identical to **`RwSetPolygonColorStruct()`** with the exception that it takes individual `RwReal`s for the red, green and blue components of the color rather than an `RwRGBColor` structure as the color specification.

RenderWare optimizes memory usage by sharing materials across multiple polygons. Setting a polygon material property with this function will cause the polygon to have its own copy of the material, not shared by any other polygons. Unless this is the desired effect, it is more memory efficient to use the corresponding material function **`RwSetMaterialColor()`** to change the underlying polygon material. This change will then be propagated to all polygons which use the material. The following line of code demonstrates how this is achieved:

```
RwSetMaterialColor(RwGetPolygonMaterial(polygon), r, g, b);
```

See Also

`RwSetMaterialColor()`
`RwSetMaterialColorStruct()`
`RwSetPolygonColorStruct()`
`RwSetSurfaceColor()`

`RwPolygon3d *`
`RwSetPolygonColorStruct``(RwPolygon3d *polygon, RwRGBColor *color);`

Description

Sets the color of the polygons material.

Arguments

`polygon`	Pointer to the polygon.
`color`	Pointer to the color.

Return Value

The argument `polygon` if successful, and `NULL` otherwise.

Comments

This function is identical to **`RwSetPolygonColor`**`()` with the exception that it takes an `RwRGBColor` structure as the color specification rather than individual `RwReal`s for the red, green and blue components of the color. This can be useful as a call-back in **`RwForAll`**`…()` functions.

RenderWare optimizes memory usage by sharing materials across multiple polygons. Setting a polygon material property with this function will cause the polygon to have its own copy of the material, not shared by any other polygons. Unless this is the desired effect, it is more memory efficient to use the corresponding material function **`RwSetMaterialColorStruct`**`()` to change the underlying polygon material. This change will then be propagated to all polygons which use the material. The following line of code demonstrates how this is achieved:

```
RwSetMaterialColorStruct(RwGetPolygonMaterial(polygon),
                         color);
```

See Also

`RwGetPolygonColor``()`
`RwSetMaterialColor``()`
`RwSetMaterialColorStruct``()`
`RwSetPolygonColor``()`
`RwSetSurfaceColor``()`

`RwPolygon3d *`
`RwSetPolygonData``(RwPolygon3d *polygon, void *data);`

Description

Sets the polygons user data pointer.

Arguments

`polygon`	Pointer to the polygon.
`data`	User data pointer.

Return Value

The argument `polygon` if successful, and `NULL` otherwise.

See Also

`RwGetPolygonData``()`

`RwPolygon3d *`
`RwSetPolygonDiffuse``(RwPolygon3d *polygon, RwReal kd);`

Description

Sets the diffuse reflection coefficient of the polygons material.

Arguments

`polygon`	Pointer to the polygon.
`kd`	Diffuse reflection coefficient in the range `CREAL(0.0)` to `CREAL(1.0)`.

Return Value

The argument `polygon` if successful, and `NULL` otherwise.

Comments

RenderWare optimizes memory usage by sharing materials across multiple polygons. Setting a polygon material property with this function will cause the polygon to have its own copy of the material, not shared by any other polygons. Unless this is the desired effect, it is more memory efficient to use the corresponding material function **`RwSetMaterialDiffuse`**`()` to change the underlying polygon material. This change will then be propagated to all polygons which use the material. The following line of code demonstrates how this is achieved:

```
RwSetMaterialDiffuse(RwGetPolygonMaterial(polygon), kd);
```

See Also

`RwGetPolygonDiffuse``()`
`RwSetMaterialDiffuse``()`
`RwSetPolygonAmbient``()`
`RwSetPolygonSpecular``()`
`RwSetPolygonSurface``()`
`RwSetSurfaceDiffuse``()`

```
RwPolygon3d *
RwSetPolygonGeometrySampling(RwPolygon3d *polygon,
                             RwGeometrySampling type);
```

Description

Sets the geometry sampling type of the polygons material.

Arguments

`polygon`	Pointer to the polygon.
`type`	The geometry sampling type.

Return Value

The argument `polygon` if successful, and `NULL` otherwise.

Comments

The geometry sampling types are:

`rwPOINTCLOUD`	Render geometry as a cloud of points.
`rwWIREFRAME`	Render geometry as a wireframe of polygon edges.
`rwSOLID`	Render geometry as a solid bounded by filled polygons.

RenderWare optimizes memory usage by sharing materials across multiple polygons. Setting a polygon material property with this function will cause the polygon to have its own copy of the material, not shared by any other polygons. Unless this is the desired effect, it is more memory efficient to use the corresponding material function **RwSetMaterialGeometrySampling**() to change the underlying polygon material. This change will then be propagated to all polygons which use the material. The following line of code demonstrates how this is achieved:

```
RwSetMaterialGeometrySampling(
                    RwGetPolygonMaterial(polygon), type);
```

See Also

RwGetPolygonGeometrySampling()
RwSetMaterialGeometrySampling()
RwSetPolygonLightSampling()
RwSetSurfaceGeometrySampling()

```
RwPolygon3d *
RwSetPolygonLightSampling(RwPolygon3d *polygon, RwLightSampling type);
```

Description

Sets the light sampling type of the polygons material.

Arguments

`polygon`	Pointer to the polygon.
`type`	The light sampling type.

Return Value

The argument `polygon` if successful, and `NULL` otherwise.

Comments

The light sampling types are:

`rwFACET`	Flat shading.
`rwVERTEX`	Smooth shading.

RenderWare optimizes memory usage by sharing materials across multiple polygons. Setting a polygon material property with this function will cause the polygon to have its own copy of the material, not shared by any other polygons. Unless this is the desired effect, it is more memory efficient to use the corresponding material function **RwSetMaterialLightSampling**`()` to change the underlying polygon material. This change will then be propagated to all polygons which use the material. The following line of code demonstrates how this is achieved:

```
RwSetMaterialLightSampling(
                RwGetPolygonMaterial(polygon), type);
```

See Also

RwGetPolygonLightSampling`()`
RwSetMaterialLightSampling`()`
RwSetPolygonGeometrySampling`()`
RwSetSurfaceLightSampling`()`

`RwPolygon3d *`
`RwSetPolygonMaterial``(RwPolygon3d *polygon, RwMaterial *material);`

Description

Sets the polygons material to a reference to material `material`.

Arguments

`polygon`	Pointer to the polygon.
`material`	Pointer to the material.

Return Value

The argument `polygon` if successful, and `NULL` otherwise.

Comments

If the polygons previous materials only reference was the polygon itself then the material will be destroyed by this function.

See Also

`RwGetPolygonMaterial``()`
`RwSetPolygonAmbient``()`
`RwSetPolygonColor``()`
`RwSetPolygonColorStruct``()`
`RwSetPolygonDiffuse``()`
`RwSetPolygonGeometrySampling``()`
`RwSetPolygonLightSampling``()`
`RwSetPolygonOpacity``()`
`RwSetPolygonSpecular``()`
`RwSetPolygonSurface``()`
`RwSetPolygonTexture``()`
`RwSetPolygonTextureModes``()`

```
RwPolygon3d *
RwSetPolygonOpacity(RwPolygon3d *polygon, RwReal opacity);
```

Description

Sets the opacity of the polygons material.

Arguments

`polygon` Pointer to the polygon.

`opacity` Opacity in the range `CREAL(0.0)` to `CREAL(1.0)`.

Return Value

The argument `polygon` if successful, and `NULL` otherwise.

Comments

An opacity of `CREAL(1.0)` yields an entirely opaque polygon. An opacity of `CREAL(0.0)` yields an entirely transparent polygon.

RenderWare optimizes memory usage by sharing materials across multiple polygons. Setting a polygon material property with this function will cause the polygon to have its own copy of the material, not shared by any other polygons. Unless this is the desired effect, it is more memory efficient to use the corresponding material function **RwSetMaterialOpacity**() to change the underlying polygon material. This change will then be propagated to all polygons which use the material. The following line of code demonstrates how this is achieved:

```
RwSetMaterialOpacity(RwGetPolygonMaterial(polygon),
                     opacity);
```

See Also

RwGetPolygonOpacity()
RwSetMaterialOpacity()
RwSetSurfaceOpacity()

`RwPolygon3d *`
`RwSetPolygonSpecular``(RwPolygon3d *polygon, RwReal ks);`

Description

Sets the specular reflection coefficient of the polygons material.

Arguments

`polygon`	Pointer to the polygon.
`ks`	Specular reflection coefficient in the range `CREAL(0.0)` to `CREAL(1.0)`.

Return Value

The argument `polygon` if successful, and `NULL` otherwise.

Comments

RenderWare optimizes memory usage by sharing materials across multiple polygons. Setting a polygon material property with this function will cause the polygon to have its own copy of the material, not shared by any other polygons. Unless this is the desired effect, it is more memory efficient to use the corresponding material function **`RwSetMaterialSpecular`**`()` to change the underlying polygon material. This change will then be propagated to all polygons which use the material. The following line of code demonstrates how this is achieved:

```
RwSetMaterialSpecular(RwGetPolygonMaterial(polygon), ks);
```

See Also

`RwGetPolygonSpecular``()`
`RwSetMaterialSpecular``()`
`RwSetPolygonAmbient``()`
`RwSetPolygonDiffuse``()`
`RwSetPolygonMaterial``()`
`RwSetPolygonSurface``()`
`RwSetSurfaceSpecular``()`

```
RwPolygon3d *
RwSetPolygonSurface(RwPolygon3d *polygon, RwReal ka, RwReal kd, RwReal ks);
```

Description

Sets the surface attributes (ambient, diffuse, and specular reflection coefficients) of the polygons material.

Arguments

`polygon`	Pointer to the polygon.
`ka`	Ambient reflection coefficient in the range `CREAL(0.0)` to `CREAL(1.0)`.
`kd`	Diffuse reflection coefficient in the range `CREAL(0.0)` to `CREAL(1.0)`.
`ks`	Specular reflection coefficient in the range `CREAL(0.0)` to `CREAL(1.0)`.

Return Value

The argument `polygon` if successful, and `NULL` otherwise.

Comments

RenderWare optimizes memory usage by sharing materials across multiple polygons. Setting a polygon material property with this function will cause the polygon to have its own copy of the material, not shared by any other polygons. Unless this is the desired effect, it is more memory efficient to use the corresponding material function **RwSetMaterialSurface**`()` to change the underlying polygon material. This change will then be propagated to all polygons which use the material. The following line of code demonstrates how this is achieved:

```
RwSetMaterialSurface(RwGetPolygonMaterial(polygon),
                                ka, kd, ks);
```

See Also

RwGetPolygonAmbient`()`
RwGetPolygonDiffuse`()`
RwGetPolygonSpecular`()`
RwSetMaterialSurface`()`
RwSetPolygonAmbient`()`
RwSetPolygonDiffuse`()`
RwSetPolygonSpecular`()`
RwSetSurface`()`

```
RwPolygon3d *
RwSetPolygonTag(RwPolygon3d *polygon, RwInt32 tag);
```

Description

Sets the polygons tag.

Arguments

`polygon`	Pointer to the polygon.
`tag`	Integer tag value to set (only the least significant 16 bits are valid).

Return Value

The argument `polygon` if successful, and `NULL` otherwise.

See Also

RwFindTaggedPolygon()
RwGetPolygonTag()
RwPolygonExt()
RwQuadExt()
RwSetClumpTag()
RwTriangleExt()

```
RwPolygon3d *
RwSetPolygonTexture(RwPolygon3d *polygon, RwTexture *texture);
```

Description

Sets the texture of the polygons material.

Arguments

`polygon` Pointer to the polygon.

`texture` Pointer to the texture.

Return Value

The argument `polygon` if successful, and `NULL` otherwise.

Comments

`NULL` may be passed as the second argument to remove the polygons materials texture.

Comments

RenderWare optimizes memory usage by sharing materials across multiple polygons. Setting a polygon material property with this function will cause the polygon to have its own copy of the material, not shared by any other polygons. Unless this is the desired effect, it is more memory efficient to use the corresponding material function **RwSetMaterialTexture**() to change the underlying polygon material. This change will then be propagated to all polygons which use the material. The following line of code demonstrates how this is achieved:

```
RwSetMaterialTexture(RwGetPolygonMaterial(polygon),
                     texture);
```

See Also

RwCreateTexture()
RwFindNamedTexture()
RwGetNamedTexture()
RwGetPolygonTexture()
RwReadNamedTexture()
RwReadTexture()
RwSetMaterialTexture()
RwSetSurfaceTexture()

RwPolygon3d *
RwSetPolygonTextureModes(RwPolygon3d *polygon, RwTextureModes modes);

Description

Sets the texture modes of a polygons material. Texture modes permit fine grain control over the rendering of textures.

Arguments

`polygon`	Pointer to the polygon.
`modes`	A bitfield representing a texture mode (or bitwise or of modes).

Return Value

The argument `polygon` if successful, and `NULL` otherwise.

Comments

The texture modes are:

`rwLIT`	The texture will be lit according to the current light sampling type of the material (`rwFACET` or `rwVERTEX`).
`rwFORESHORTEN`	The texture will be foreshortened in a perspectively correct manner.
`rwFILTER`	A filter will be applied to the texture to reduce the effect of pixelation due to aliasing.

For further information see the Texture Modes section in Chapter 2: Data Types.

Unlike **RwAddTextureModeToPolygon**(), which simply adds one or more texture modes to a polygons materials set of modes, **RwSetPolygonTextureModes**() replaces a polygons materials entire set of modes with those specified.

RenderWare optimizes memory usage by sharing materials across multiple polygons. Setting a polygon material property with this function will cause the polygon to have its own copy of the material, not shared by any other polygons. Unless this is the desired effect, it is more memory efficient to use the corresponding material function **RwSetMaterialTextureModes**() to change the underlying polygon material. This change will then be propagated to all polygons which use the material. The following line of code demonstrates how this is achieved:

```
RwSetMaterialTextureModes(RwGetPolygonMaterial(polygon),
                          modes);
```

See Also

RwAddTextureModeToPolygon()
RwGetPolygonTextureModes()
RwRemoveTextureModeFromPolygon()
RwSetMaterialTextureModes()
RwSetPolygonLightSampling()
RwSetPolygonTexture()
RwSetSurfaceTextureModes()

```
RwPolygon3d *
RwSetPolygonUV(RwPolygon3d *polygon, RwUV *uvarray);
```

Description

Sets the texture (U, V) co-ordinates for the polygons vertices.

Arguments

`polygon`	Pointer to the polygon.
`uvarray`	Pointer to an array of `RwUV` structures.

Return Value

The argument `polygon` if successful, and `NULL` otherwise.

Comments

Note that the array `uvarray` must be large enough to accommodate the texture co-ordinates of all of the polygons vertices.

See Also

RwCubicTexturizeClump()
RwEnvMapClump()
RwGetClumpVertexUV()
RwGetPolygonNumSides()
RwGetPolygonUV()
RwSetClumpVertexUV()
RwSphericalTexturizeClump()
RwVertexExt()

```
RwRaster *
RwSetRasterData(RwRaster *raster, void *data);
```

Description

Sets the rasters user data pointer.

Arguments

`raster`	Pointer to the raster.
`data`	User data pointer.

Return Value

The argument `raster` if successful, and `NULL` otherwise.

See Also

`RwGetRasterData``()`

`RwScene *`
`RwSetSceneData``(RwScene *scene, void *data);`

Description

Sets the scenes user data pointer.

Arguments

`scene`	Pointer to the scene.
`data`	User data pointer.

Return Value

The argument `scene` if successful, and `NULL` otherwise.

See Also

`RwGetSceneData``()`

RwBool
RwSetShapePath(char *path, RwCombineOperation op);

Description

Modifies the shape path. The path may be prepended to (rwPRECONCAT), appended to (rwPOSTCONCAT), or replaced (rwREPLACE).

Arguments

path	Pointer to the new path string.
op	Combination operator.

Return Value

TRUE if successful, and FALSE otherwise.

See Also

RwGetNamedTexture()
RwGetShapePath()
RwReadNamedTexture()
RwReadRaster()
RwReadMaskRaster()
RwReadShape()
RwReadTexture()

```
RwSpline *
RwSetSplineData(RwSpline *spline, void *data);
```

Description

Sets the splines user data pointer.

Arguments

`spline`	Pointer to the spline.
`data`	User data pointer.

Return Value

The argument `spline` if successful, and `NULL` otherwise.

See Also

RwGetSplineData()

`RwSpline *`
`RwSetSplinePoint``(RwSpline *spline, RwInt32 index, RwV3d *point);`

Description

Sets the specified control point of the spline.

Arguments

`spline`	Pointer to the spline.
`index`	Index of the control point to set, in the range `1` ≤ `index` ≤ total number of control points.
`point`	Pointer to the new control point of the spline.

Return Value

The argument `spline` if successful, and `NULL` otherwise.

Comments

Note that passing `1` as the value of argument index will set the first control point.

See Also

`RwCreateSpline``()`
`RwGetSplineNumPoints``()`
`RwGetSplinePoint``()`

RwBool
RwSetSurface(RwReal ka, RwReal kd, RwReal ks);

Description

Sets the surface attributes (ambient, diffuse, and specular reflection coefficients) of the current material.

Arguments

`ka`	Ambient reflection coefficient in the range `CREAL(0.0)` to `CREAL(1.0)`.
`kd`	Diffuse reflection coefficient in the range `CREAL(0.0)` to `CREAL(1.0)`.
`ks`	Specular reflection coefficient in the range `CREAL(0.0)` to `CREAL(1.0)`.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

This function can only be called within the context of an **RwModelBegin**() … **RwModelEnd**() block.

See Also

RwModelBegin()
RwModelEnd()
RwSetMaterialSurface()
RwSetPolygonSurface()
RwSetSurfaceAmbient()
RwSetSurfaceDiffuse()
RwSetSurfaceSpecular()

RwBool
RwSetSurfaceAmbient(RwReal ka);

Description

Sets the ambient reflection coefficient of the current material.

Arguments

ka	Ambient reflection coefficient in the range CREAL(0.0) to CREAL(1.0).

Return Value

TRUE if successful, and FALSE otherwise.

Comments

This function can only be called within the context of an **RwModelBegin**() … **RwModelEnd**() block.

See Also

RwModelBegin()
RwModelEnd()
RwSetMaterialAmbient()
RwSetPolygonAmbient()
RwSetSurface()
RwSetSurfaceDiffuse()
RwSetSurfaceSpecular()

```
RwBool
RwSetSurfaceColor(RwReal r, RwReal g, RwReal b);
```

Description

Sets the current materials color.

Arguments

`r`	Red component of the color, in the range `CREAL(0.0)` to `CREAL(1.0)`.
`g`	Green component of the color, in the range `CREAL(0.0)` to `CREAL(1.0)`.
`b`	Blue component of the color, in the range `CREAL(0.0)` to `CREAL(1.0)`.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

This function can only be called within the context of an **`RwModelBegin`**`()` … **`RwModelEnd`**`()` block.

See Also

`RwModelBegin``()`
`RwModelEnd``()`
`RwSetMaterialColor``()`
`RwSetMaterialColorStruct``()`
`RwSetPolygonColor``()`
`RwSetPolygonColorStruct``()`

`RwBool`
`RwSetSurfaceDiffuse``(RwReal kd);`

Description

Sets the current materials diffuse reflection coefficient.

Arguments

`kd`	Diffuse reflection coefficient in the range `CREAL(0.0)` to `CREAL(1.0)`.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

This function can only be called within the context of an **`RwModelBegin`**`()` … **`RwModelEnd`**`()` block.

See Also

`RwModelBegin``()`
`RwModelEnd``()`
`RwSetMaterialDiffuse``()`
`RwSetPolygonDiffuse``()`
`RwSetSurface``()`

```
RwBool
RwSetSurfaceGeometrySampling(RwGeometrySampling type);
```

Description

Sets the geometry sampling type of the current material.

Arguments

`type`	The geometry sampling type.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

The geometry sampling types are:

`rwPOINTCLOUD`	Render geometry as a cloud of points.
`rwWIREFRAME`	Render geometry as a wireframe of polygon edges.
`rwSOLID`	Render geometry as a solid bounded by filled polygons.

This function can only be called within the context of an **RwModelBegin**() … **RwModelEnd**() block.

See Also

RwModelBegin()
RwModelEnd()
RwSetMaterialGeometrySampling()
RwSetPolygonGeometrySampling()
RwSetSurfaceLightSampling()

`RwBool`
`RwSetSurfaceLightSampling``(RwLightSampling type);`

Description

Sets the light sampling type of the current material.

Arguments

`type`	The light sampling type.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

The light sampling types are:

`rwFACET`	Flat shading.
`rwVERTEX`	Smooth shading.

This function can only be called within the context of an **`RwModelBegin`**`()` … **`RwModelEnd`**`()` block.

See Also

`RwModelBegin``()`
`RwModelEnd``()`
`RwSetMaterialLightSampling``()`
`RwSetPolygonLightSampling``()`
`RwSetSurfaceGeometrySampling``()`

`RwBool`
`RwSetSurfaceOpacity``(RwReal opacity);`

Description

Sets the opacity of the current material.

Arguments

`opacity` Opacity in the range `CREAL(0.0)` to `CREAL(1.0)`.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

An opacity of `CREAL(1.0)` yields an entirely opaque polygon. An opacity of `CREAL(0.0)` yields an entirely transparent polygon.

This function can only be called within the context of an **`RwModelBegin`**`()` … **`RwModelEnd`**`()` block.

See Also

`RwModelBegin``()`
`RwModelEnd``()`
`RwSetMaterialOpacity``()`
`RwSetPolygonOpacity``()`

```
RwBool
RwSetSurfaceSpecular(RwReal ks);
```

Description

Sets the current materials specular reflection coefficient.

Arguments

`ks`	Specular reflection coefficient in the range `CREAL(0.0)` to `CREAL(1.0)`.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

This function can only be called within the context of an **`RwModelBegin()`** … **`RwModelEnd()`** block.

See Also

`RwModelBegin()`
`RwModelEnd()`
`RwSetMaterialSpecular()`
`RwSetPolygonSpecular()`
`RwSetSurface()`

RwBool
RwSetSurfaceTexture(char *name);

Description

Sets the current materials texture to the texture with the given name. **RwGetNamedTexture**() is used to find the texture. If the named texture is found (either in the dictionary stack or in the file system), the current materials texture is set to the texture found.

Arguments

name	Name of a texture.

Return Value

TRUE if successful, and FALSE otherwise.

Comments

For more information on how the named texture is found, see the description of **RwGetNamedTexture**().

This function can only be called within the context of an **RwModelBegin**() … **RwModelEnd**() block.

See Also

RwGetNamedTexture()
RwModelBegin()
RwModelEnd()
RwSetMaterialTexture()
RwSetPolygonTexture()
RwSetSurfaceTextureExt()

RwBool
RwSetSurfaceTextureExt(char *name, char *maskname);

Description

Sets the current materials texture to the texture with the given name, The mask raster read from the file given by maskname is applied to the texture.

RwGetNamedTexture() is used to find the texture. If the named texture is found (either in the dictionary stack or in the file system), the current materials texture is set to the texture found.

RwReadMaskRaster() is used to read the mask raster and **RwMaskTexture**() is used to apply the mask raster to the texture.

Arguments

name	Name of a texture.
maskname	Filename of the mask raster.

Return Value

TRUE if successful, and FALSE otherwise.

Comments

The masking of a texture is a destructive operation. All existing and future references to the texture will be affected by the masking operation.

For more information on how the named texture is searched for, see the description of **RwGetNamedTexture**(). For information on how the mask raster is read see **RwReadMaskRaster**() and for information on how the mask raster is applied see **RwMaskTexture**().

This function can only be called within the context of an **RwModelBegin**() … **RwModelEnd**() block.

See Also

RwGetNamedTexture()
RwMaskTexture()
RwModelBegin()
RwModelEnd()
RwReadMaskRaster()
RwSetMaterialTexture()
RwSetPolygonTexture()
RwSetSurfaceTexture()

```
RwBool
RwSetSurfaceTextureModes(RwTextureModes modes);
```

Description

Sets the texture mode (or modes) of the current material. Texture modes permit fine grain control over the rendering of textures.

Arguments

`modes`	A bitfield representing a texture mode (or bitwise or of modes).

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

The texture modes are:

`rwLIT`	The texture will be lit according to the current light sampling type of the material (`rwFACET` or `rwVERTEX`).
`rwFORESHORTEN`	The texture will be foreshortened in a perspectively correct manner.
`rwFILTER`	A filter will be applied to the texture to reduce the effect of pixelation due to aliasing.

For further information see the Texture Modes section in Chapter 2: Data Types.

Unlike **`RwAddTextureModeToSurface`**`()`, which simply adds one or more texture modes to the current materials set of modes, **`RwSetSurfaceTextureModes`**`()` replaces the current materials entire set of modes with those specified.

This function can only be called within the context of an **`RwModelBegin`**`()` … **`RwModelEnd`**`()` block.

See Also

`RwAddTextureModeToSurface``()`
`RwRemoveTextureModeFromSurface``()`
`RwSetMaterialTextureModes``()`
`RwSetPolygonTextureModes``()`
`RwSetSurfaceLightSampling``()`
`RwSetSurfaceTexture``()`
`RwSetSurfaceTextureExt``()`

RwBool
RwSetTag(RwInt32 tag);

Description

Assigns an integer tag to the current clump under construction.

Arguments

`tag`	The integer tag.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

This function can only be called within the context of an **RwClumpBegin**() … **RwClumpEnd**() block.

See Also

RwClumpBegin()
RwClumpEnd()
RwFindTaggedClump()
RwPolygonExt()
RwQuadExt()
RwSetClumpTag()
RwSetPolygonTag()
RwTriangleExt()

`RwTexture *`
`RwSetTextureData``(RwTexture *texture, void *data);`

Description

Sets the textures user data pointer.

Arguments

`texture`	Pointer to the texture.
`data`	User data pointer.

Return Value

The argument `texture` if successful, and `NULL` otherwise.

See Also

`RwGetTextureData``()`

RwBool
RwSetTextureDictSearchMode(RwSearchMode mode);

Description

Sets the mode for searching the texture dictionary stack.

Arguments

`mode`	The texture dictionary stack search mode.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

This function affects how textures are searched for in the texture dictionary stack. In particular, it affects the behavior of **RwFindNamedTexture**(), **RwGetNamedTexture**(), **RwGetNumNamedTextures**() and the **RwForAllNamedTextures**…() functions.

If the texture dictionary search mode is `rwLOCAL`, then only the current texture dictionary (the top element of the texture dictionary stack) is searched. If the search mode is `rwGLOBAL`, all dictionaries on the dictionary stack, from the current dictionary down, are searched until the named texture is found or there are no more dictionaries left to search.

The texture dictionary search modes are:

`rwLOCAL`	Search only the top most dictionary in the texture dictionary stack.
`rwGLOBAL`	Search all the dictionaries in the texture dictionary stack.

See Also

RwFindNamedTexture()
RwForAllNamedTextures()
RwGetNamedTexture()
RwGetNumNamedTextures()
RwGetTextureDictSearchMode()
RwTextureDictBegin()
RwTextureDictEnd()

`RwBool`
`RwSetTextureDithering``(RwTextureDitherMode mode);`

Description

Sets the current global texture dithering mode to be applied to subsequently loaded textures.

Arguments

`mode`	The texture dithering mode.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

The current texture dithering mode is a global parameter which controls whether textures read from disk are dithered to increase perceived color resolution or not.

The following texture dither modes are supported:

`rwDITHERON`	Activates dithering.
`rwDITHEROFF`	Deactivates dithering.
`rwAUTODITHER`	Adopts the auto-dithering mode of raster reading to decide whether to dither textures.

The default mode is `rwAUTODITHER`.

See Also

`RwGetNamedTexture``()`
`RwGetTextureDithering``()`
`RwGetTextureGammaCorrection``()`
`RwReadNamedTexture``()`
`RwReadRaster``()`
`RwReadTexture``()`
`RwSetTextureGammaCorrection``()`

```
RwTexture *
RwSetTextureFrame(RwTexture *texture, RwInt32 index);
```

Description

Sets the current frame of the texture.

Arguments

`texture`	Pointer to the texture.
`index`	Frame number.

Return Value

The argument `texture` if successful, and `NULL` otherwise.

Comments

For those textures which consist of a sequence of frames this function will set the current frame to be the one with the given sequence number. Sequence numbers are in the range `0 … n – 1`, where `n` is the number of frames in the texture.

The current frame of a texture is the one used in all rendering of polygons associated with the texture.

See Also

`RwGetTextureFrame``()`
`RwGetTextureNumFrames``()`
`RwTextureNextFrame``()`

`RwTexture *`
`RwSetTextureFrameStep``(RwTexture *texture, RwInt32 step);`

Description

Sets the textures current frame step size. This is the number of frames by which the current frame index is incremented or decremented by a call to **`RwTextureNextFrame`**`()`.

Arguments

`texture`	Pointer to the texture.
`step`	Number of frames to increment or decrement per call to **`RwTextureNextFrame`**`()`.

Return Value

The argument `texture` if successful, and `NULL` otherwise.

Comments

A value of `+1` (the default) will play the texture movie forward, one frame at a time. A value of `–1` will play the movie backwards. Other values will play the movie at different speeds.

See Also

`RwGetTextureFrame``()`
`RwGetTextureFrameStep``()`
`RwGetTextureNumFrames``()`
`RwSetTextureFrame``()`
`RwTextureNextFrame``()`

RwBool
RwSetTextureGammaCorrection(RwState mode);

Description

Sets the current global texture gamma correction mode applied to subsequently loaded textures.

Arguments

`mode`	The gamma correction mode.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

The current texture gamma correction mode is a global parameter which controls whether textures read from disk are gamma corrected or not.

The following texture gamma correction modes are supported:

`rwON`	Gamma correct.
`rwOFF`	Do not gamma correct.

The default mode is `rwON`.

See Also

RwGetNamedTexture()
RwGetTextureDithering()
RwGetTextureGammaCorrection()
RwReadNamedTexture()
RwReadRaster()
RwReadTexture()
RwSetTextureDithering()

RwTexture *
RwSetTextureRaster(RwTexture *texture, RwRaster *raster);

Description

Sets the raster of the specified texture to raster.

Arguments

`texture`	Pointer to the texture.
`raster`	Pointer to the raster.

Return Value

The argument `texture` if successful, and `NULL` otherwise.

Comments

The raster selected into a texture provides the actual pixel values of the texture map. **RwSetTextureRaster**() is used to dynamically generate textures from platform specific bitmaps (via **RwBitmapRaster**()) or the viewport of a RenderWare camera (via **RwGetCameraViewportRaster**()).

Rasters cannot be shared between textures. It is an error to specify a raster already selected into a texture.

The textures existing raster will be destroyed by this function.

raster must be of the correct size - 128 by 128 pixels (or 128 by `n`*128 pixels for a multi-frame texture). It is an error to call **RwSetTextureRaster**() with a raster that is not of this size.

See Also

RwBitmapRaster()
RwCreateRaster()
RwCreateTexture()
RwDestroyRaster()
RwDuplicateRaster()
RwGetCameraViewportRaster()
RwReadRaster()
RwSetTextureRaster()

```
RwUserDraw *
RwSetUserDrawAlignment(RwUserDraw *userdraw,
    RwUserDrawAlignmentTypes alignment);
```

Description

Sets the user-draws alignment flags. The user-draws alignment flags determine which part of the user-draws bounding box is used for alignment.

Arguments

`userdraw` Pointer to the user-draw.

`alignment` A bitfield representing a set of alignment flags.

Return Value

The argument `userdraw` if successful, and `NULL` otherwise.

Comments

The alignment flags are:

`0`	Center the user-draw.
`rwALIGNTOP`	Align with the top edge of the user-draw.
`rwALIGNBOTTOM`	Align with the bottom edge of the user-draw.
`rwALIGNLEFT`	Align with the left edge of the user-draw.
`rwALIGNRIGHT`	Align with the right edge of the user-draw.

See Also

RwCreateUserDraw()
RwGetUserDrawAlignment()
RwSetUserDrawParentAlignment()

```
RwUserDraw *
RwSetUserDrawCallback(RwUserDraw *userdraw,
     void (*callback)(RwUserDraw *userdraw, void *camimage,
                          RwRect *rect, void *data));
```

Description

Sets the call-back function that renders the user-draw.

Arguments

`userdraw` Pointer to the user-draw.

`callback` Pointer to the call-back rendering function.

Return Value

The argument `userdraw` if successful, and `NULL` otherwise.

Comments

User-draw call-backs should be declared as follows:

```
void callback(RwUserDraw *userdraw, void *camimage,
              RwRect *rect, void *data);
```

Where the call-backs arguments are as follows:

`userdraw` Pointer to the user-draw to be rendered.

`camimage` The cameras image buffer as returned by **RwGetCameraImage**() for the current camera. camimage is device dependent. For more information, see Appendix B.

`rect` Pointer to a rectangle defining the area of the cameras image buffer into which the call-back may render. This rectangle is specified in viewport space co-ordinates, i.e., (`0`, `0`) is the origin of the viewport.

`data` Pointer to the user data of the user-draw being drawn. This value can be obtained by calling **RwGetUserDrawData**() with userdraw as an argument. data is passed directly to the call-back function for the convenience of the application developer.

Note that the call-back function is always called after all clumps in the scene have been rendered, i.e., when **RwEndCameraUpdate**() is called. Therefore user-draw rendering always appear in front of clump rendering. In the case of overlapping user-draws, the order of rendering is not defined.

See Also

RwCreateUserDraw()
RwGetCameraImage()
RwGetUserDrawCallback()
RwGetUserDrawData()

```
RwUserDraw *
RwSetUserDrawData(RwUserDraw *userdraw, void *data);
```

Description

Sets the user-draws user data pointer.

Arguments

`userdraw`	Pointer to the user-draw.
`data`	User data pointer.

Return Value

The argument `userdraw` if successful, and `NULL` otherwise.

Comments

`data` is passed as the fourth parameter to the user-draws call-back function when the user-draw is being rendered.

See Also

RwGetUserDrawData()
RwSetUserDrawCallback()

```
RwUserDraw *
RwSetUserDrawOffset(RwUserDraw *userdraw, RwInt32 x, RwInt32 y);
```

Description

Sets the X and Y offset in viewport space units relative to the alignment point of the user-draw.

Arguments

`userdraw`	Pointer to the user-draw.
`x`	X offset from the alignment point of the user-draw (in viewport space units).
`y`	Y offset from the alignment point of the user-draw (in viewport space units).

Return Value

The argument `userdraw` if successful, and `NULL` otherwise.

Comments

`x` and `y` may be negative.

See Also

`RwCreateUserDraw``()`
`RwGetUserDrawOffset``()`

```
RwUserDraw *
RwSetUserDrawParentAlignment(RwUserDraw *userdraw, RwUserDrawAlignmentTypes
    Dataalignment);
```

Description

Sets the alignment flags of the user-draws parent. A user-draws parent is either the bounding box of the clump that owns the user-draw or the current cameras viewport.

The alignment flags of the user-draws parent determine which part of the user-draws parent rectangle is aligned with the user-draw. The actual point of alignment between a user-draw and its parent is determined by the user-draws alignment flags and the parents alignment flags.

Arguments

`userdraw` Pointer to the user-draw.

`alignment` A bitfield representing an alignment flag (or bitwise or of flags).

Return Value

The argument `userdraw` if successful, and `NULL` otherwise.

Comments

If the user-draws type is `rwBBOXALIGN` then the user-draws parent is the bounding box of the clump to which the user-draw is attached. If the type is `rwVPALIGN` the user-draws parent is the viewport of the current camera when the user-draw is rendered. If the user-draws type is `rwVERTEXALIGN` or `rwCLUMPALIGN` then the user-draw has no parent and the parent alignment bitfield is ignored.

The alignment flags are:

`0`	Align with the center of the parent.
`rwALIGNTOP`	Align with the top edge of the parent.
`rwALIGNBOTTOM`	Align with the bottom edge of the parent.
`rwALIGNLEFT`	Align with the left edge of the parent.
`rwALIGNRIGHT`	Align with the right edge of the parent.

See Also

RwGetUserDrawParentAlignment()
RwSetUserDrawAlignment()
RwSetUserDrawType()

`RwUserDraw *`
`RwSetUserDrawSize``(RwUserDraw *userdraw, RwInt32 width,`
`    RwInt32 height);`

Description

Sets the width and height (in viewport space units) of the user-draw.

Arguments

`userdraw`	Pointer to the user-draw.
`width`	Width of the user-draw (in viewport space units).
`height`	Height of the user-draw (in viewport space units).

Return Value

The argument `userdraw` if successful, and `NULL` otherwise.

Comments

RenderWare does not clip the user-draws rendering to the specified area. If rendering takes place outside of the designated area garbage may appear on the display.

See Also

`RwCreateUserDraw``()`
`RwGetUserDrawSize``()`

```
RwUserDraw *
RwSetUserDrawType(RwUserDraw *userdraw, RwUserDrawType type);
```

Description

Sets the user-draws type.

Arguments

`userdraw`	Pointer to the user-draw.
`type`	Type of the user-draw.

Return Value

The argument `userdraw` if successful, and `NULL` otherwise.

Comments

The user-draw types are:

`rwCLUMPALIGN`	Align with the origin of the owning clump.
`rwVERTEXALIGN`	Align with a vertex of the owning clump.
`rwBBOXALIGN`	Align with the viewport bounding box of the owning clump.
`rwVPALIGN`	Align with the viewing cameras viewport.

See Also

RwCreateUserDraw()
RwGetUserDrawType()

```
RwUserDraw *
RwSetUserDrawVertexIndex(RwUserDraw *userdraw, RwInt32 index);
```

Description

Sets the index of the clump vertex with which the user-draw is aligned.

Arguments

`userdraw`	Pointer to the user-draw.
`index`	The index of the vertex with which the user-draw is aligned.

Return Value

The argument `userdraw` if successful, and `NULL` otherwise.

Comments

The vertex index is only used if the user-draws type is `rwVERTEXALIGN`, for all other user-draw types it is ignored.

`index` is an index into the vertex list of the clump to which the user-draw is attached.

See Also

RwCreateUserDraw()
RwGetUserDrawType()
RwGetUserDrawVertexIndex()

```
void
RwSetUserError(void);
```

Description

Sets the error status to `E_RW_USER`.

Arguments

None.

Return Value

None.

Comments

RwSetUserError`()` is used when a call-back for **RwForAll**…`()` wishes to signal failure and terminate iteration.

See Also

RwForAllClumpsInHierarchy`()`
RwForAllClumpsInScene`()`
RwForAllLightsInScene`()`
RwForAllNamedTextures`()`
RwForAllPolygonsInClump`()`
RwForAllUserDrawsInClump`()`
RwGetError`()`

```
RwCamera *
RwShowCameraImage(RwCamera *camera, void *param);
```

Description

Copies the damaged regions of the cameras image buffer to the portion of the display (screen or window) specified by the cameras viewport.

Arguments

`camera`	Pointer to the camera.
`param`	Device dependent parameter.

Return Value

The argument `camera` if successful, and `NULL` otherwise.

Comments

For a description of the device dependent parameter, `param`, see Appendix B.

This function often immediately follows an **`RwBeginCameraUpdate`**`()` … **`RwEndCameraUpdate`**`()` block in order to copy the rendering performed within the **`RwBeginCameraUpdate`**`()` … **`RwEndCameraUpdate`**`()` block to the display.

Note that the cameras image buffer is not automatically cleared after the call to **`RwShowCameraImage`**`()`. To clear the image buffer, call **`RwClearCameraViewport`**`()`.

If a number of separate cameras are being used to provide different images simultaneously, it is advisable to do the **`RwBeginCameraUpdate`**`()` … **`RwEndCameraUpdate`**`()` `RwEndCameraUpdate`block for each camera first, then perform all the calls to **`RwShowCameraImage`**`()` `RwShowCameraImage`afterwards.

See Also

`RwBeginCameraUpdate``()`
`RwDamageCameraViewport``()`
`RwEndCameraUpdate``()`
`RwInvalidateCameraViewport``()`
`RwRenderClump``()`
`RwRenderScene``()`
`RwUndamageCameraViewport``()`

`RwBool`
`RwSphere``(RwReal radius, RwInt32 density);`

Description

Adds a sphere to the current clump under construction. The sphere is transformed by the CTM, and the current material is assigned to its polygons. The sphere is centered about the origin.

Arguments

`radius`	Radius of the sphere.
`density`	Density of facets in the sphere. A value of `0` results in a cube. Higher values increase the number of facets exponentially.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

If the spheres radius is negative the polygons forming the sphere will face inward.

This function can only be called within the context of an **`RwClumpBegin`**`()` … **`RwClumpEnd`**`()` **or** **`RwProtoBegin`**`()` … **`RwProtoEnd`**`()` block.

See Also

`RwBlock``()`
`RwClumpBegin``()`
`RwClumpEnd``()`
`RwCone``()`
`RwCylinder``()`
`RwDisc``()`
`RwHemisphere``()`
`RwProtoBegin``()`
`RwProtoEnd``()`

`RwClump *`
`RwSphericalTexturizeClump``(RwClump *clump);`

Description

Sets the texture co-ordinates for every polygon belonging to the clump using the spherical projection method.

A spherical mapping results in the construction of a nominal sphere onto which the texture is applied in a way similar to the projection of a two dimensional atlas map onto the surface of a globe. The resulting sphere is then mapped to the clump by shrink wrapping the clump with the sphere.

Arguments

`clump` Pointer to the clump.

Return Value

The argument `clump` if successful, and `NULL` otherwise.

Comments

This function need only be called the first time a clump is textured and not each time the clump is rendered.

Note that this function does not set the textures associated with the clumps polygons; this must be accomplished separately. The following code fragment illustrates this procedure:

```
RwForAllPolygonsInClumpPointer(clump,
     (RwPolygon3d*( *)())RwSetPolygonTexture, texture);
```

See Also

RwCubicTexturizeClump()
RwForAllPolygonsInClump()
RwGetClumpVertexUV()
RwGetPolygonUV()
RwSetClumpVertexUV()
RwSetPolygonTexture()
RwSetPolygonUV()

RwV3d *

RwSplinePoint(RwSpline *spline, RwSplinePath path, RwReal where, RwV3d *point, RwV3d *vector);

Description

Calculates a point and vector for a position on the spline specified by parameter where.

Arguments

`spline`	Pointer to the spline.
`path`	Type of curve distribution.
`where`	Relative distance along the spline in the range `CREAL(0.0)` to `CREAL(1.0)`.
`point`	Pointer to the point that will receive the spline location point.
`vector`	Pointer to the vector that will receive the spline tangent vector.

Return Value

The argument `point` if successful, and `NULL` otherwise.

Comments

Note that if a vector is not needed, `NULL` can be passed as the value of parameter vector.

See Also

RwCreateSpline()
RwSetSplinePoint()

RwReal
RwSplineTransform(RwSpline *spline, RwSplinePath path, RwReal where, RwV3d *up, RwMatrix4d *matrix);

Description

Calculates a Frenet transform matrix at a specified parameter position on a spline and a returns measure of the paths anti-clockwise curvature at this point. This matrix will transform a clump to the specified parameter position on the path with its "Look At" direction pointing tangent to the path, "Look Up" pointing up and "Look At" pointing toward or away from the center of curvature.

Arguments

`spline`	The spline curve.
`path`	The type of spline path.
`where`	The parameter position.
`up`	An "up" vector. If up is `NULL`, **`RwSplineTransform()`** will produce a transform which aligns a clumps "Look Up" Y vector with the local Y vector of a *Frenet* frame. If up is non-`NULL`, **`RwSplineTransform()`** will produce a transform which aligns a clumps "Look Up" Y vector so as never to roll upside down with respect to this up vector.
`matrix`	Pointer to the matrix the will receive the Frenet transform matrix.

Return Value

The curvature at the specified point if successful, and `NULL` otherwise.

Comments

If up is `NULL`, the returned matrix will always transform a clumps "Look Left" vector to point toward the center of curvature - as though the clump were being swung on a rod extending this direction from this point. Since the clumps "Look At" vector always transforms to a forward tangent along the spline, when the center of curvature lies to the right this can result in the clumps "Look Up" vector rolling upside down since handedness is conserved. This behavior is not always desirable. Such rolling can be suppressed by specifying an appropriate up vector. **`RwSplineTransform()`** will suppress any roll relative to a specified up vector. For example, when modeling the motion of a car over a hilly road circuit, an up vector of [`CREAL(0.0)`, `CREAL(1.0)`, `CREAL(0.0)`] would give a transform in which the car turns around corners and "tilts" over hills but does *not* roll - the wheels stay on the ground. When specifying a non-`NULL` up vector, some restricted rolling or banking may be reintroduced by pre-concatenating a local Z rotation whose angle is driven by the anti-clockwise curvature value returned by **`RwSplineTransform()`**. When large and positive, this indicates a sharp anti-clockwise turn in the plane normal to the up vector; when zero this indicates no turn in the plane; when large and negative this indicates a sharp clockwise turn in the plane. An appropriate bank angle may be found with a function such as `atan(curvature)`.

See Also

`RwCreateSpline()`
`RwDestroySpline()`
`RwDuplicateSpline()`
`RwSplinePoint()`

`void`
`RwSRandom``(RwUInt32 seed)`

Description

Sets pseudo random number sequence start for **`RwRandom`**`()`.

Arguments

`seed`	Value to seed pseudo random number sequence.

Return Value

None.

Comments

Unlike `srand()`, **`RwSRandom`**`()` does not return the old seed; the reason for this is that the amount of state information used is much more than a single word. Like `rand()`, however, **`RwRandom`**`()` will by default produce a sequence of numbers that can be duplicated by calling **`RwSRandom`**`()` with `1` as the seed.

See Also

RwRandom() RwRandom**RwV3d ***
RwSubtractVector(RwV3d *a, RwV3d *b, RwV3d *c);

Description

Subtracts two vectors.

Arguments

`a`	Pointer to the first vector.
`b`	Pointer to the second vector.
`c`	Pointer to the vector that will receive the result.

Return Value

The argument `c` if successful, and `NULL` otherwise.

See Also

RwAddVector()
RwCrossProduct()
RwDotProduct()
RwNormalize()
RwScaleVector()
RwTransformVector()

RwBool
RwTextureDictBegin(void);

Description

Creates a new, empty texture dictionary and pushes it on the texture dictionary stack. The newly created dictionary becomes the current texture dictionary.

Arguments

None.

Return Value

TRUE if successful, and FALSE otherwise.

See Also

RwFindNamedTexture()
RwForAllNamedTextures()
RwGetNamedTexture()
RwGetTextureDictSearchMode()
RwSetTextureDictSearchMode()
RwReadNamedTexture()
RwTextureDictEnd()

RwBool
RwTextureDictEnd(void);

Description

Destroys the current texture dictionary and all the textures that it contains. The texture dictionary stack is restored to its state at the time of the last **RwTextureDictBegin**().

Arguments

None.

Return Value

TRUE if successful, and FALSE otherwise.

Comments

This function must not be called if any of the textures in the current dictionary are still in use. Use **RwSetMaterialTexture**() or **RwSetPolygonTexture**() with a parameter of NULL to remove all textures in the dictionaries from their materials before calling **RwTextureDictEnd**().

See Also

RwDestroyTexture()
RwFindNamedTexture()
RwForAllNamedTextures()
RwGetNamedTexture()
RwGetTextureDictSearchMode()
RwReadNamedTexture()
RwSetMaterialTexture()
RwSetPolygonTexture()
RwSetTextureDictSearchMode()
RwTextureDictBegin()

`RwTexture *`
`RwTextureNextFrame``(RwTexture *texture);`

Description

Increments or decrements the current frame index by the current frame step.

Arguments

`texture` Pointer to the texture.

Return Value

The argument `texture` if successful, and `NULL` otherwise.

Comments

Note that the current frame index will not go outside the range `0` … `n-1` (where `n` is the number of frames in the texture), instead the index will wrap around in either direction.

See Also

`RwSetTextureFrame``()`
`RwSetTextureFrameStep``()`
`RwTextureNextFrame``()`

```
RwCamera *
RwTiltCamera(RwCamera *camera, RwReal angle);
```

Description

Rotates the camera about its X axis.

Arguments

`camera`	Pointer to the camera.
`angle`	Angle of rotation (in degrees).

Return Value

The argument `camera` if successful, and `NULL` otherwise.

Comments

A positive value for `angle` will cause the camera to tilt down.

See Also

RwGetCameraLookAt()
RwGetCameraLookUp()
RwPanCamera()
RwPointCamera()
RwResetCamera()
RwRevolveCamera()
RwSetCameraLookAt()
RwSetCameraLookUp()
RwTransformCameraOrientation()

RwBool
RwTransformBegin(void);

Description

Pushes a copy of the CTM onto the transformation stack.

Arguments

None.

Return Value

TRUE if successful, and FALSE otherwise.

Comments

This function can only be called within the context of an **RwModelBegin**() … **RwModelEnd**() block.

See Also

RwClumpBegin()
RwClumpEnd()
RwJointTransformBegin()
RwModelBegin()
RwModelEnd()
RwProtoBegin()
RwProtoEnd()
RwTransformEnd()

```
RwCamera *
RwTransformCamera(RwCamera *camera, RwMatrix4d *matrix,
                  RwCombineOperation op)
```

Description

Applies a transformation matrix to the cameras current position and orientation.

Arguments

`camera`	Pointer to the camera.
`matrix`	Pointer to the transformation matrix.
`op`	Combination operator.

Return Value

The argument `camera` if successful and `NULL` otherwise.

Comments

This function may be used to align a camera with a clump or a light. The following code fragment demonstrates this.

```
RwGetClumpLTM(Clump, RwScratchMatrix());
RwTransformCamera(Camera, RwScratchMatrix(), rwREPLACE);
```

See Also

RwCreateCamera()
RwGetCameraLookAt()
RwGetCameraLookRight()
RwGetCameraLookUp()
RwGetCameraPosition()
RwGetCameraLTM ()
RwGetClumpLTM()
RwGetLightLTM()
RwResetCamera()
RwSetCameraLookAt()
RwSetCameraLookUp()
RwSetCameraPosition()
RwTransformClump()
RwTransformLight()

```
RwCamera *
RwTransformCameraOrientation(RwCamera *camera, RwMatrix4d *matrix);
```

Description

Applies a transformation matrix to the cameras current orientation (which is determined by the Look At and Look Up vectors). This function does not affect the cameras position.

Arguments

`camera` Pointer to the camera.

`matrix` Pointer to the transformation matrix.

Return Value

The argument `camera` if successful, and `NULL` otherwise.

See Also

RwGetCameraLookAt()
RwGetCameraLookRight()
RwGetCameraLookUp()
RwPanCamera()
RwPointCamera()
RwResetCamera()
RwRevolveCamera()
RwSetCameraLookAt()
RwSetCameraLookUp()
RwTransformCamera()

```
RwClump *
RwTransformClump(RwClump *clump, RwMatrix4d *matrix, RwCombineOperation op);
```

Description

Applies a transformation matrix to the clumps modeling matrix.

Arguments

`clump`	Pointer to the clump.
`matrix`	Pointer to the transformation matrix.
`op`	Combination operator.

Return Value

The argument `clump` if successful, and `NULL` otherwise.

See Also

`RwGetClumpLTM()`
`RwGetClumpMatrix()`
`RwTransformClumpJoint()`

```
RwClump *
RwTransformClumpJoint(RwClump *clump, RwMatrix4d *matrix, RwCombineOperation
    op);
```

Description

Applies a transformation matrix to the clumps joint (articulation) matrix.

Arguments

`clump`	Pointer to the clump.
`matrix`	Pointer to the transformation matrix.
`op`	Combination operator.

Return Value

The argument `clump` if successful, and `NULL` otherwise.

See Also

RwGetClumpJointMatrix()
RwGetClumpLTM()
RwTransformClump()

RwBool
RwTransformCTM(RwMatrix4d *matrix);

Description

Replaces the CTM with the specified matrix.

Arguments

`matrix` Pointer to a transformation matrix.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

This function can only be called within the context of an **RwModelBegin**() … **RwModelEnd**() block.

See Also

RwIdentityCTM()
RwModelBegin()
RwModelEnd()
RwRotateCTM()
RwScaleCTM()
RwTransformJointTM()
RwTranslateCTM()

`RwBool`
`RwTransformEnd``(void);`

Description

Restores the previous value of the CTM. Also has the effect of restoring the transformation stack to its state at the time of the last **`RwTransformBegin`**`()`.

Arguments

None.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

This function can only be called within the context of an **`RwModelBegin`**`()` … **`RwModelEnd`**`()` block.

See Also

`RwClumpBegin``()`
`RwClumpEnd``()`
`RwJointTransformEnd``()`
`RwModelBegin``()`
`RwModelEnd``()`
`RwPopScratchMatrix``()`
`RwProtoBegin``()`
`RwProtoEnd``()`
`RwTransformBegin``()`

`RwBool`
`RwTransformJointTM``(RwMatrix4d *matrix);`

Description

Replaces the current joint transformation matrix with the specified matrix.

Arguments

`matrix` Pointer to a transformation matrix.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

This function can only be called within the context of an **`RwModelBegin`**`()` … **`RwModelEnd`**`()` block.

See Also

`RwIdentityJointTM``()`
`RwModelBegin``()`
`RwModelEnd``()`
`RwRotateJointTM``()`
`RwTransformCTM``()`

```
RwLight *
RwTransformLight(RwLight *light, RwMatrix4d *matrix,
                 RwCombineOperation op)
```

Description

Applies a transformation matrix to the lights current position and direction vector (if applicable)

Arguments

`light`	Pointer to the light.
`matrix`	Pointer to the transformation matrix.
`op`	Combination operator.

Return Value

The argument `light` if successful and `NULL` otherwise.

See Also

RwCreateLight()
RwGetClumpLTM()
RwGetCameraLTM()
RwGetLightLTM()
RwGetLightPosition()
RwGetLightVector()
RwSetLightPosition()
RwSetLightVector()
RwTransformCamera()
RwTransformClump()

```
RwMatrix4d *
RwTransformMatrix(RwMatrix4d *dest, RwMatrix4d *source, RwCombineOperation
    op);
```

Description

Applies the transformation matrix `source` to matrix `dest`.

Arguments

`dest`	Pointer to the matrix to be transformed.
`source`	Pointer to the transformation matrix.
`op`	Combination operator.

Return Value

The argument `dest` if successful, and `NULL` otherwise.

Comments

If `op` is `rwREPLACE`, **`RwTransformMatrix()`** is equivalent to **`RwCopyMatrix()`** (although note that the order of the source and destination matrices is reversed). Otherwise, it is equivalent to **`RwMultiplyMatrix()`**, but does not require an intermediate matrix to hold the result.

See Also

`RwCopyMatrix()`
`RwIdentityMatrix()`
`RwInvertMatrix()`
`RwMultiplyMatrix()`
`RwOrthoNormalizeMatrix()`
`RwRotateMatrix()`
`RwRotateMatrixCos()`
`RwScaleMatrix()`
`RwTranslateMatrix()`

```
RwV3d *
RwTransformPoint(RwV3d *point, RwMatrix4d *matrix);
```

Description

Applies a transformation matrix to a point.

Arguments

`point`	Pointer to the point.
`matrix`	Pointer to the transformation matrix.

Return Value

The argument `point` if successful, and `NULL` otherwise.

See Also

RwTransformVector()

```
RwV3d *
RwTransformVector(RwV3d *vector, RwMatrix4d *matrix);
```

Description

Applies a transformation matrix to a vector. However, since a vector does not have a position in space, the translation component of the matrix is ignored.

Arguments

`vector`	Pointer to the vector.
`matrix`	Pointer to the transformation matrix.

Return Value

The argument `vector` if successful, and `NULL` otherwise.

See Also

RwAddVector()
RwCrossProduct()
RwDotProduct()
RwNormalize()
RwScaleVector()
RwSubtractVector()
RwTransformPoint()

`RwBool`
`RwTranslateCTM``(RwReal tx, RwReal ty, RwReal tz);`

Description

Pre-concatenates a translation matrix onto the CTM.

Arguments

`tx`	Translation parallel to the X axis.
`ty`	Translation parallel to the Y axis.
`tz`	Translation parallel to the Z axis.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

This function can only be called within the context of an **`RwModelBegin`**`()` … **`RwModelEnd`**`()` block.

See Also

`RwIdentityCTM``()`
`RwModelBegin``()`
`RwModelEnd``()`
`RwRotateCTM``()`
`RwScaleCTM``()`
`RwTransformCTM``()`

```
RwMatrix4d *
RwTranslateMatrix(RwMatrix4d *matrix, RwReal tx, RwReal ty,
    RwReal tz, RwCombineOperation op);
```

Description

Builds a translation matrix and applies it to `matrix`. The operation may be a pre-concatenation, post-concatenation, or replacement.

Arguments

`matrix`	Pointer to the matrix.
`tx`	Translation parallel to the X axis.
`ty`	Translation parallel to the Y axis.
`tz`	Translation parallel to the Z axis.
`op`	Combination operator.

Return Value

The argument `matrix` if successful, and `NULL` otherwise.

See Also

RwIdentityMatrix()
RwInvertMatrix()
RwMultiplyMatrix()
RwOrthoNormalizeMatrix()
RwRotateMatrix()
RwRotateMatrixCos()
RwScaleMatrix()
RwTransformMatrix()
RwTranslateCTM()

`RwBool`
`RwTriangle``(RwInt32 v1, RwInt32 v2, RwInt32 v3);`

Description

Adds a triangle to the current clump under construction. The current material is assigned to the triangle.

Arguments

`v1`	First vertex of the triangle.
`v2`	Second vertex of the triangle.
`v3`	Third vertex of the triangle.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

This function is exactly equivalent to calling **`RwPolygon`**`()` with an array of three vertex indices.

This function can only be called within the context of an **`RwClumpBegin`**`()` … **`RwClumpEnd`**`()` or **`RwProtoBegin`**`()` … **`RwProtoEnd`**`()` block.

See Also

`RwAddPolygonToClump``()`
`RwClumpBegin``()`
`RwClumpEnd``()`
`RwPolygon``()`
`RwPolygonExt``()`
`RwProtoBegin``()`
`RwProtoEnd``()`
`RwQuad``()`
`RwQuadExt``()`
`RwVertex``()`
`RwVertexExt``()`

RwBool
RwTriangleExt(RwInt32 v1, RwInt32 v2, RwInt32 v3, RwInt32 tag);

Description

Adds a triangle with the specified integer tag to the clump under construction. The current material is assigned to the triangle.

Arguments

`v1`	First vertex of the triangle.
`v2`	Second vertex of the triangle.
`v3`	Third vertex of the triangle.
`tag`	Integer tag to set (only the least significant 16 bits are valid).

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comments

This function is exactly equivalent to calling **RwPolygonExt**() with an array of three vertex indices.

This function can only be called within the context of an **RwClumpBegin**() … **RwClumpEnd**() or **RwProtoBegin**() … **RwProtoEnd**() block.

See Also

RwAddPolygonToClump()
RwClumpBegin()
RwClumpEnd()
RwFindTaggedPolygon()
RwGetPolygonTag()
RwPolygon()
RwPolygonExt()
RwProtoBegin()
RwProtoEnd()
RwQuad()
RwQuadExt()
RwSetPolygonTag()
RwSetTag()
RwTriangle()
RwVertex()
RwVertexExt()

`RwCamera *`
`RwUndamageCameraViewport``(RwCamera *camera, RwInt32 x, RwInt32 y, RwInt32 width, RwInt32 height);`

Description

Marks a rectangular area of the viewport as undamaged (not in need of updating).

Arguments

`camera`	Pointer to the camera.
`x`	X co-ordinate of the rectangles top left corner (in viewport space co-ordinates).
`y`	Y co-ordinate of the rectangles top left corner (in viewport space co-ordinates).
`width`	Width of the rectangle (in viewport space units).
`height`	Height of the rectangle (in viewport space units).

Return Value

The argument `camera` if successful, and `NULL` otherwise.

See Also

`RwClearCameraViewport``()`
`RwDamageCameraViewport``()`
`RwGetClumpViewportRect``()`
`RwInvalidateCameraViewport``()`
`RwShowCameraImage``()`

RwCamera *
RwVCMoveCamera(RwCamera *camera, RwReal x, RwReal y, RwReal z);

Description

Moves the camera position by the given delta (x, y, z) values (in camera space units) with respect to the cameras orientation. For instance, a positive z value moves the camera forward.

Arguments

camera	Pointer to the camera.
x	Amount to move the camera along its X axis (in camera space units).
y	Amount to move the camera along its Y axis (in camera space units).
z	Amount to move the camera along its Z axis (in camera space units).

Return Value

The argument camera if successful, and NULL otherwise.

See Also

RwGetCameraPosition()
RwResetCamera()
RwSetCameraPosition()
RwTransformCamera()
RwWCMoveCamera()

```
RwInt32
RwVertex(RwReal x, RwReal y, RwReal z);
```

Description

Adds a vertex, transformed by the CTM, to the current clump under construction.

Arguments

`x`	X co-ordinate of the vertex.
`y`	Y co-ordinate of the vertex.
`z`	Z co-ordinate of the vertex.

Return Value

The index of the new vertex if successful, and `0` otherwise.

Comments

This function can only be called within the context of an **RwClumpBegin**`()` … **RwClumpEnd**`()` or **RwProtoBegin**`()` … **RwProtoEnd**`()` block.

See Also

RwAddVertexToClump`()`
RwClumpBegin`()`
RwClumpEnd`()`
RwPolygon`()`
RwPolygonExt`()`
RwProtoBegin`()`
RwProtoEnd`()`
RwQuad`()`
RwQuadExt`()`
RwTriangle`()`
RwTriangleExt`()`
RwVertexExt`()`

RwInt32
RwVertexExt(RwReal x, RwReal y, RwReal z, RwUV *uv, RwV3d *normal);

Description

Adds a vertex, transformed by the CTM, to the clump under construction and specifies the vertexs texture co-ordinates and normal vector.

Arguments

x	X co-ordinate of the vertex.
y	Y co-ordinate of the vertex.
z	Z co-ordinate of the vertex.
uv	Pointer to the RwUV structure holding the vertexs texture co-ordinates.
normal	Pointer to the RwV3d structure holding the X, Y, and Z components of the normal vector.

Return Value

The index of the new vertex if successful, and 0 otherwise.

Comments

NULL may be passed as the value of uv or normal if the application programmer does not wish to set the value of either or both of these parameters.

This function can only be called within the context of an **RwClumpBegin**() … **RwClumpEnd**() or **RwProtoBegin**() … **RwProtoEnd**() block.

See Also

RwAddVertexToClump()
RwClumpBegin()
RwClumpEnd()
RwGetClumpVertexUV()
RwGetClumpVertexNormal()
RwPolygon()
RwPolygonExt()
RwProtoBegin()
RwProtoEnd()
RwQuad()
RwQuadExt()
RwSetClumpVertexUV()
RwTriangle()
RwTriangleExt()
RwVertex()

`RwCamera *`
`RwWCMoveCamera``(RwCamera *camera, RwReal x, RwReal y, RwReal z);`

Description

Moves the camera position by the given delta `(x, y, z)` values, with respect to the world space co-ordinate system.

Arguments

`camera`	Pointer to the camera.
`x`	Amount to move the camera parallel to the world X axis (in world space units).
`y`	Amount to move the camera parallel to the world Y axis (in world space units).
`z`	Amount to move the camera parallel to the world Z axis (in world space units).

Return Value

The argument `camera` if successful, and `NULL` otherwise.

See Also

RwGetCameraPosition()
RwResetCamera()
RwSetCameraPosition()
RwTransformCamera()
RwVCMoveCamera()

`RwBool`
`RwWriteShape``(char *path, RwClump *clump);`

Description

Writes the clump as a script (`.rwx`) file with the given name.

Arguments

`path`	Pointer to the filename.
`clump`	Pointer to the clump.

Return Value

`TRUE` if successful, and `FALSE` otherwise.

Comment

When a clump read from a script file is written out, certain aspects of the structure of the original input script file are not preserved. However, this does not affect the appearance or the behavior of the clump.

It is recommended that the extension `.rwx` be used for script files. However, this is not enforced by the library.

See Also

`RwClumpBegin``()`
`RwClumpEnd``()`
`RwCreateClump``()`

The Scripting Language

Related Topics

Script Keywords

Miscellaneous Notes

Object Builder API Functions

Script Keywords

This section gives a brief summary of each scripting language keyword. For further information on the operation of each keyword see the description of the keywords associated Object Builder API function.

Related Topics

AddHint <hint>

AddTextureMode <mode>

Ambient <ka>

AxisAlignment <alignment>

Block <width> <height> <depth>

ClumpBegin

ClumpEnd

Color <red> <green> <blue>

Cone <height> <radius> <nsides>

Cylinder <height> <baserad> <toprad> <nsides>

Diffuse <kd>

Disc <height> <radius> <nsides>

GeometrySampling <sampling>

Hemisphere <radius> <density>

Hints <hints>

Identity

IdentityJoint

Include <filename>

IncludeGeometry <filename>

JointTransformBegin

JointTransformEnd

LightSampling <sampling>

MaterialBegin

MaterialEnd

ModelBegin

ModelEnd

Opacity <opacity>

Polygon <nsides> <v1> ...

PolygonExt <nsides> <v1> ...

ProtoBegin <name>

ProtoEnd

ProtoInstance <name>

ProtoInstanceGeometry <name>

Quad <v1> <v2> <v3> <v4>

QuadExt <v1> <v2> <v3> <v4> [Tag <tag>]

RemoveHint <hint>

RemoveTextureMode <mode>

Rotate <x> <y> <z> <angle>

RotateJoint <x> <y> <z> <angle>

Scale <x> <y> <z>

Specular <ks>

Sphere <radius> <density>

Surface <ka> <kd> <ks>

Tag <tag>

Texture <name>

TextureDithering <mode>

TextureExt <name> [Mask <mask>]

TextureGammaCorrection <mode>

TextureModes <modes>

Trace <mode>

Transform <elements>

TransformBegin

TransformEnd

TransformJoint <elements>

Translate <x> <y> <z>

Triangle <v1> <v2> <v3>

TriangleExt <v1> <v2> <v3> [Tag <tag>]

Vertex <x> <y> <z>

VertexExt <x> <y> <z> [Normal <i> <j> <k>]   [UV <u> <v>]

AddHint <hint>

Description

Adds the specified hint (or hints) to the set of hints of the current clump under construction.

Arguments

`hint`	A space separated list of hints where each hint is one of **`Container`**, **`HS`** or **`Editable`**.

API Equivalent

`RwAddHint``()`

AddTextureMode <mode>

Description

Adds the specified texture mode (or modes) to the set of texture modes of the current material.

Arguments

`mode`	A space separated list of texture modes where each texture mode is one of **`Lit`** , **`Foreshorten`** or **`Filter`**.

API Equivalent

`RwAddTextureModeToSurface``()`

Ambient <ka>

Description

Sets the ambient coefficient of reflectance of the current material.

Arguments

`ka`	The ambient coefficient.

API Equivalent

`RwSetSurfaceAmbient``()`

AxisAlignment <alignment>

Description

Sets the axis alignment type of the current clump under construction.

Arguments

`alignment` The clump axis alignment. One of **`None`**, **`ZOrientX`**, **`ZOrientY`** or **`XYZ`**.

API Equivalent

`RwSetAxisAlignment``()`

Block <width> <height> <depth>

Description

Adds polygons representing a block of the given dimensions to the current clump under construction.

Arguments

`width`	The width of the block.
`height`	The height of the block.
`depth`	The depth of the block.

Comments

The vertices of the block are transformed by the current transformation matrix (CTM). The current material is applied to the polygons of the block.

It is not legal to specify a block dimension of zero.

API Equivalent

`RwBlock``()`

ClumpBegin

Description

Begins the construction of a new clump.

Arguments

None.

Comments

The clump begun with **`ClumpBegin`** will be the target of all operations on the current clump under construction until a matching **`ClumpEnd`** is found.

If **`ClumpBegin`** is nested within another **`ClumpBegin`** or **`ProtoBegin`** block a new child clump will be created.

API Equivalent

`RwClumpBegin``()`

ClumpEnd

Description

Ends the construction of the current clump.

Arguments

None.

API Equivalent

`RwClumpEnd()`

Color <red> <green> <blue>

Description

Sets the color of the current material to the given color.

Arguments

`red`	The red component of the color
`green`	The green component of the color.
`blue`	The blue component of the color.

API Equivalent

`RwSetSurfaceColor``()`

Cone <height> <radius> <nsides>

Description

Adds polygons representing a cone to the current clump under construction.

Arguments

`height`	The height of the cone (up the Y axis).
`radius`	The radius of the cone (in the X-Z plane).
`nsides`	The number of sides.

Comments

The vertices of the cone are transformed by the current transformation matrix (CTM). The current material is applied to the polygons of the cone.

If a negative radius is given, the polygons forming the cone will face towards the axis of the cone.

API Equivalent

`RwCone``()`

Cylinder <height> <baserad> <toprad> <nsides>

Description

Adds polygons representing a cylinder to the current clump under construction.

Arguments

`height`	The height of the cylinder (up the Y axis).
`baserad`	The radius of the cylinder base (in the X-Z plane).
`toprad`	The radius of the cylinder top.
`nsides`	The number of sides.

Comments

The vertices of the cylinder are transformed by the current transformation matrix (CTM). The current material is applied to the polygons of the cylinder.

If a negative radius is given, the polygons forming the cylinder will face towards the axis of the cylinder.

API Equivalent

`RwCylinder``()`

Diffuse <kd>

Description

Sets the diffuse coefficient of reflectance of the current material.

Arguments

`kd` The diffuse coefficient.

API Equivalent

RwSetSurfaceDiffuse()

Disc <height> <radius> <nsides>

Description

Adds polygons representing a disc to the current clump under construction.

Arguments

`height`	The displacement (up the Y axis) of the disc.
`radius`	The radius of the disc.
`nsides`	The number of sides.

Comments

This keyword is normally used for capping cones and cylinders.

The vertices of the disc are transformed by the current transformation matrix (CTM). The current material is applied to the polygons of the disc.

If a negative radius is given the polygons forming the disc will face downwards.

API Equivalent

`RwDisc``()`

GeometrySampling <sampling>

Description

Sets the geometry sampling type of the current material to the type specified.

Arguments

`sampling` The geometry sampling type. One of **`PointCloud`**, **`WireFrame`** or **`Solid`**.

API Equivalent

`RwSetSurfaceGeometrySampling``()`

Hemisphere <radius> <density>

Description

Adds polygons representing a hemisphere to the current clump under construction.

Arguments

`radius`	The radius of the hemisphere.
`density`	Controls the accuracy of the hemisphere.

Comments

The base of the hemisphere lies on the X-Z plane.

The density controls the number of polygons used to approximate the hemisphere. A density of 0 results in a pyramid. The number of polygons used for the approximation varies exponentially with `density`.

The vertices of the hemisphere are transformed by the current transformation matrix (CTM). The current material is applied to the polygons of the hemisphere.

If a negative radius is given the polygons forming the hemisphere will face inward.

API Equivalent

`RwHemisphere``()`

Hints <hints>

Description

Sets the set of hints of the current clump under construction to those specified.

Arguments

`hints`	`NULL` or a space separated list of hints where each hint is one of **`Container`**, **`HS`** or **`Editable`**.

API Equivalent

`RwSetHints``()`

Identity

Description

Sets the current transformation matrix (CTM) to the identity matrix.

Arguments

None.

API Equivalent

RwIdentityCTM()

IdentityJoint

Description

Sets the current joint transformation matrix to the identity matrix.

Arguments

None.

API Equivalent

`RwIdentityJointTM``()`

Include <filename>

Description

Merges the clump constructed by reading the specified script file with the current clump under construction.

Arguments

`filename` Filename of the script file to read.

Comments

Include does not create a new child clump. It merges the polygons of the clump read with the polygons of the current clump under construction. If a new child clump is desired surround the **`Include`** keyword with a **`ClumpBegin`** … **`ClumpEnd`** block.

The materials of the polygons read from the script file are preserved.

API Equivalent

`RwInclude``()`

IncludeGeometry <filename>

Description

Merges the clump constructed by reading the specified script file with the current clump under construction. The materials of the merged polygons are replaced with the current material.

Arguments

`filename` Filename of the script file to read.

Comments

`IncludeGeometry` does not create a new child clump. It merges the polygons of the clump read with the polygons of the current clump under construction. If a new child clump is desired surround the **`IncludeGeometry`** keyword with a **`ClumpBegin`** … **`ClumpEnd`** block.

The materials of the polygons read from the script file are replaced with the current material.

API Equivalent

`RwIncludeGeometry``()`

JointTransformBegin

Description

Pushes a copy of the current joint transformation matrix onto the joint matrix stack in order that the existing value may be restored at a later stage.

Arguments

None.

API Equivalent

`RwJointTransformBegin()`

JointTransformEnd

Description

Restores the current joint transformation matrix from the last matrix pushed onto the joint matrix stack with **`JointTransformBegin`**. This discards any modifications to the current joint transformation matrix made since the last **`JointTransformBegin`**.

Arguments

None.

API Equivalent

`RwJointTransformEnd``()`

LightSampling <sampling>

Description

Sets the light sampling type of the current material to the type specified.

Arguments

`sampling` The light sampling type. One of **`Facet`** or **`Vertex`**.

API Equivalent

`RwSetSurfaceLightSampling``()`

MaterialBegin

Description

Pushes a copy of the current material onto the material stack in order that the existing value may be restored at a later stage.

Arguments

None.

API Equivalent

`RwMaterialBegin()`

MaterialEnd

Description

Restores the current material from the last material pushed onto the material stack with **`MaterialBegin`**. This discards any modifications to the current material made since the last **`MaterialBegin`**.

Arguments

None.

API Equivalent

`RwMaterialEnd``()`

ModelBegin

Description

Begins a new modeling context.

Arguments

None.

Comments

With the exception of the **`Trace`** keyword, **`ModelBegin`** must be the first keyword in a script file.

API Equivalent

`RwModelBegin``()`

ModelEnd

Description

Ends a modeling context.

Arguments

None.

Comments

ModelEnd must be the last keyword in a script file.

API Equivalent

`RwModelEnd()`

Opacity <opacity>

Description

Sets the opacity of the current material.

Arguments

`opacity` The opacity of the material.

Comments

A value of `1.0` yields an entirely opaque material. A value of `0.0` yields an entirely transparent material. Intermediate values yield varying degrees of semi-transparency.

API Equivalent

`RwSetSurfaceOpacity``()`

Polygon <nsides> <v1> ... `<vn>`

Description

Creates a new polygon with the given vertices and adds it to the current clump under construction.

Arguments

`nsides`	The number of sides of the polygon.
`v1` ... `vn`	The indices of the polygons vertices.

Comments

`v1` ... `vn` are indices into the vertex list of the current clump under construction. The number of indices specified must be equal to `nsides`.

The current material is applied to the new polygon.

`Polygon` is a synonym of **`PolygonExt`**. The optional integer tag may also be specified when using the **`Polygon`** keyword.

API Equivalent

`RwPolygon``()`

PolygonExt <nsides> <v1> ... `<vn>` [Tag `<tag>`]

Description

Creates a new polygon with the given vertices and adds it to the current clump under construction. An integer tag may be specified in order to mark the polygon.

Arguments

`nsides`	The number of sides of the polygon.
`v1` ... `vn`	The indices of the polygons vertices.
`tag`	An optional integer tag to assign to the polygon.

Comments

`v1` ... `vn` are indices into the vertex list of the current clump under construction. The number of indices specified must be equal to nsides.

The current material is applied to the new polygon.

The integer tag is optional. To specify a tag use the optional keyword modifier **`Tag`** followed by an integer.

API Equivalent

`RwPolygonExt``()`

ProtoBegin <name>

Description

Begins the definition of a prototype.

Arguments

`name`	The name of the prototype.

Comments

The prototype begun with **`ProtoBegin`** will be the target of all operation on the current clump under construction until a matching **`ProtoEnd`** is found.

`ProtoBegin` must not be nested within a **`ClumpBegin`** … **`ClumpEnd`** or **`ProtoBegin`** … **`ProtoEnd`** block.

The new prototype will override the definition of any existing prototype with the same name.

API Equivalent

`RwProtoBegin``()`

ProtoEnd

Description

Ends the definition of the current prototype.

Arguments

None.

API Equivalent

`RwProtoEnd()`

ProtoInstance <name>

Description

Merges the prototype of the given name with the current clump under construction.

Arguments

`name` The name of the prototype to merge.

Comments

ProtoInstance does not create a new child clump. It merges the polygons of the prototype with the polygons of the current clump under construction. If a new child clump is desired, surround the **`ProtoInstance`** keyword with a **`ClumpBegin`** … **`ClumpEnd`** block.

The materials of the polygons of the prototype are preserved.

API Equivalent

`RwProtoInstance``()`

ProtoInstanceGeometry <name>

Description

Merges the prototype of the specified name with the current clump under construction. The materials of the merged polygons are replaced with the current material.

Arguments

`name`	The name of the prototype to merge.

Comments

`ProtoInstanceGeometry` does not create a new child clump. It merges the polygons of the prototype with the polygons of the current clump under construction. If a new child clump is desired, surround the **`ProtoInstanceGeometry`** keyword with a **`ClumpBegin`** … **`ClumpEnd`** block.

The materials of the polygons of the prototype are replaced with the current material.

API Equivalent

`RwProtoInstanceGeometry``()`

Quad <v1> <v2> <v3> <v4>

Description

Creates a new quadrilateral with the given vertices and adds it to the current clump under construction.

Arguments

`v1`	The first vertex index of the quadrilateral.
`v2`	The second vertex index of the quadrilateral.
`v3`	The third vertex index of the quadrilateral.
`v4`	The fourth vertex index of the quadrilateral.

Comments

`v1`, `v2`, `v3` and `v4` are indices into the vertex list of the current clump under construction.

The current material is applied to the new quadrilateral.

`Quad` is a synonym of **`QuadExt`**. The optional integer tag may also be specified when using the **`Quad`** keyword.

This keyword is semantically identical to **`Polygon`** 4 `<v1>` `<v2>` `<v3>` `<v4>`

API Equivalent

`RwQuad``()`

QuadExt <v1> <v2> <v3> <v4> [Tag <tag>]

Description

Creates a new quadrilateral with the given vertices and adds it to the current clump under construction. An integer tag may be specified in order to mark the polygon.

Arguments

v1	The first vertex index of the quadrilateral.
v2	The second vertex index of the quadrilateral.
v3	The third vertex index of the quadrilateral.
v4	The fourth vertex index of the quadrilateral.
`tag`	An optional integer tag to assign to the quadrilateral.

Comments

`v1`, `v2`, `v3` and `v4` are indices into the vertex list of the current clump under construction.

The current material is applied to the new quadrilateral.

The integer tag is optional. To specify a tag use the optional keyword modifier **`Tag`** followed by an integer.

This keyword is semantically identical to **`PolygonExt`** 4 `<v1>` `<v2>` `<v3>` `<v4>` **`Tag`** `<tag>`

API Equivalent

`RwQuadExt``()`

RemoveHint <hint>

Description

Removes the specified hint (or hints) from the set of hints of the current clump under construction.

Arguments

`hint` A space separated list of hints where each hint is one of **`Container`**, **`HS`** or **`Editable`**.

API Equivalent

`RwRemoveHint``()`

RemoveTextureMode <mode>

Description

Removes the texture mode (or modes) specified from the set of texture modes of the current material.

Arguments

`mode`	A space separated list of texture modes where each texture mode is one of **`Lit`**, **`Foreshorten`** or **`Filter`**.

API Equivalent

`RwRemoveTextureModeFromSurface``()`

Rotate <x> <y> <z> <angle>

Description

Build a transformation matrix representing a rotation of angle degrees about the specified vector and pre-concatenate it onto the current transformation matrix (CTM).

Arguments

`x`	The X component of the vector to rotate about.
`y`	The Y component of the vector to rotate about.
`z`	The Z component of the vector to rotate about.
`angle`	The angle of rotation (in degrees).

API Equivalent

`RwRotateCTM``()`

RotateJoint <x> <y> <z> <angle>

Description

Builds a transformation matrix representing a rotation of angle degrees about the specified vector and pre-concatenate it onto the current joint transformation matrix.

Arguments

`x`	The X component of the vector to rotate about.
`y`	The Y component of the vector to rotate about.
`z`	The Z component of the vector to rotate about.
`angle`	The angle (in degrees) of rotation.

API Equivalent

`RwRotateJointTM``()`

Scale <x> <y> <z>

Description

Builds a scale matrix and pre-concatenates it onto the current transformation matrix (CTM).

Arguments

`x`	The scale factor along the X axis.
`y`	The scale factor along the Y axis.
`z`	The scale factor along the Z axis.

API Equivalent

`RwScaleCTM``()`

Specular <ks>

Description

Sets the specular coefficient of reflectance of the current material.

Arguments

`ks` The specular coefficient.

API Equivalent

`RwSetSurfaceSpecular``()`

Sphere <radius> <density>

Description

Adds polygons representing a sphere to the current clump under construction.

Arguments

`radius`	The radius of the sphere.
`density`	Controls the accuracy of the sphere.

Comments

The density controls the number of polygons used to approximate the sphere. A density of zero results in a cube. The number of polygons used for the approximation varies exponentially with density.

The vertices of the sphere are transformed by the current transformation matrix (CTM). The current material is applied to the polygons of the sphere.

If a negative radius is given the polygons forming the sphere will face inward.

API Equivalent

`RwSphere``()`

Surface <ka> <kd> <ks>

Description

Sets the ambient, diffuse and specular coefficients of reflectance of the current material.

Arguments

`ka`	The ambient coefficient.
`kd`	The diffuse coefficient.
`ks`	The specular coefficient.

API Equivalent

`RwSetSurface``()`

Tag <tag>

Description

Sets the integer tag of the current clump under construction to the specified value.

Arguments

`tag`	The integer tag.

API Equivalent

`RwSetTag``()`

Texture <name>

Description

Sets the texture of the current material to the texture with the name specified. The current materials texture can be removed by specifying a texture name of `NULL`.

Arguments

`name`	The texture name (or `NULL`).

Comments

The texture is found by searching the current texture dictionary stack and shape path. For a discussion of the algorithm used to find the texture see the description of **`RwGetNamedTexture`**`()`.

`Texture` is a synonym of **`TextureExt`**. A mask may be also be specified (using the **`Mask`** keyword modifier) with the **`Texture`** keyword.

API Equivalent

`RwSetSurfaceTexture``()`

TextureDithering <mode>

Description

Sets the global texture dithering mode applied to all subsequently loaded textures.

Arguments

`mode`	The texture dithering mode. One of **`Auto`**, **`On`** or **`Off`**.

API Equivalent

`RwSetTextureDithering``()`

TextureExt <name> [Mask <mask>]

Description

Sets the texture of the current material to the texture with the name specified. The texture is masked by the mask raster with the given filename.

Arguments

`name`	The texture name.
`mask`	The filename of the mask raster.

Comments

The texture is found by searching the current texture dictionary stack and search path. For a discussion of the algorithm used to find the texture see the description of **`RwGetNamedTexture`**`()`.

The raster is found by searching the current shape path. For a discussion of method used to find the mask raster and to apply it to the texture see the descriptions of **`RwReadMaskRaster`**`()` and **`RwMaskTexture`**`()` respectively.

API Equivalent

`RwSetSurfaceTextureExt``()`

TextureGammaCorrection <mode>

Description

Sets the global texture gamma correction mode applied to all subsequently loaded textures.

Arguments

`mode` The texture gamma correction mode. One of **`On`** or **`Off`**.

API Equivalent

`RwSetTextureGammaCorrection``()`

TextureModes <modes>

Description

Sets the current materials set of texture modes to the modes specified.

Arguments

`modes`	`NULL` or a space separated list of texture modes where each texture modes is one of **`Lit`**, **`Foreshorten`** or **`Filter`**.

API Equivalent

`RwSetSurfaceTextureModes``()`

Trace <mode>

Description

Sets the script tracing mode.

Arguments

`mode` The script tracing mode. One of **`On`** or **`Off`**.

Comments

Script tracing is only available under debugging versions of the RenderWare library. This keyword has no effect under retail versions of the library.

The **`Trace`** keyword is the only keyword permitted in a file before **`ModelBegin`**.

API Equivalent

None.

Transform <elements>

Description

Replaces the elements of the current transformation matrix (CTM) with the specified matrix elements.

Arguments

`elements` A space separated list of sixteen real matrix elements.

API Equivalent

RwTransformCTM()

TransformBegin

Description

Pushes a copy of the current transformation matrix (CTM) onto the matrix stack in order that the existing value may be restored at a later stage.

Arguments

None.

API Equivalent

`RwTransformBegin()`

TransformEnd

Description

Restores the current transformation matrix (CTM) from the last transformation pushed onto the matrix stack with **`TransformBegin`**. This discards any modifications to the current transformation matrix (CTM) made since the last **`TransformBegin`**.

Arguments

None.

API Equivalent

`RwTransformEnd``()`

TransformJoint <elements>

Description

Replaces the elements of the current joint transformation matrix with the specified matrix elements.

Arguments

`elements` A space separated list of sixteen real matrix elements.

API Equivalent

`RwTransformJointTM``()`

Translate <x> <y> <z>

Description

Builds a translation matrix and pre-concatenates it onto the current transformation matrix (CTM).

Arguments

`x`	The translation along the X axis.
`y`	The translation along the Y axis.
`z`	The translation along the Z axis.

API Equivalent

`RwTranslateCTM``()`

Triangle <v1> <v2> <v3>

Description

Creates a new triangle with the given vertices and adds it to the current clump under construction.

Arguments

`v1`	The first vertex index of the triangle.
`v2`	The second vertex index of the triangle.
`v3`	The third vertex index of the triangle.

Comments

`v1`, `v2` and `v3` are indices into the vertex list of the current clump under construction.

The current material is applied to the new triangle.

`Triangle` is a synonym of **`TriangleExt`**. The integer tag may also be specified when using the **`Triangle`** keyword.

API Equivalent

`RwTriangle``()`

TriangleExt <v1> <v2> <v3> [Tag <tag>]

Description

Creates a new triangle with the given vertices and adds it to the current clump under construction. An integer tag may be specified in order to mark the polygon.

Arguments

`v1`	The first vertex index of the triangle.
`v2`	The second vertex index of the triangle.
`v3`	The third vertex index of the triangle.
`tag`	An optional integer tag to assign to the triangle.

Comments

`v1`, `v2` and `v3` are indices into the vertex list of the current clump under construction.

The current material is applied to the new triangle.

The integer tag is optional. To specify a tag use the optional keyword modifier **`Tag`** followed by an integer.

API Equivalent

`RwTriangleExt``()`

Vertex <x> <y> <z>

Description

Creates a new vertex and adds it to the current clump under construction.

Arguments

`x`	The X coordinate of the vertex.
`y`	The Y coordinate of the vertex.
`z`	The Z coordinate of the vertex.

Comments

The vertex is transformed by the current transformation matrix (CTM).

`Vertex` is a synonym of **`VertexExt`**. The texture coordinates and unit shading normal may also be specified (with the optional keyword modifiers **`UV`** and **`Normal`** respectively) with the **`Vertex`** keyword.

API Equivalent

`RwVertex``()`

VertexExt <x> <y> <z> [Normal <i> <j> <k>] [UV <u> <v>]

Description

Creates a new vertex and adds it to the current clump under construction. The unit shading normal and texture coordinates are set to the specified values.

Arguments

`x`	The X coordinate of the vertex.
`y`	The Y coordinate of the vertex.
`z`	The Z coordinate of the vertex.
`i`	The X component of the optional unit shading normal.
`j`	The Y component of the optional unit shading normal.
`k`	The Z component of the optional unit shading normal.
`u`	The U coordinate of the optional texture coordinates.
`v`	The V coordinate of the optional texture coordinates.

Comments

The vertex is transformed by the current transformation matrix (CTM).

The unit shading normal and texture coordinates are optional. To specify a normal use the optional keyword modifier **`Normal`**.To specify texture coordinates use the optional keyword modifier **`UV`**.

API Equivalent

`RwVertexExt``()`

Miscellaneous Notes

This section contains certain important notes concerning the use of the scripting language.

- [illegible]
- Scripting keywords which take vertices as arguments, such as **`Triangle`**, require a vertex to be identified by its vertex number. Vertex numbers start at **one** and each time that the **`Vertex`** (or **`VertexExt`**) keyword is used within a script, they are incremented by one. The number of the first vertex created by the **`Vertex`** (or **`VertexExt`**) keyword is one, the second is two, and so on. Note that the vertex numbering is not affected by any scripting keywords which add vertices (such as **`Include`** or **`Sphere`**) other than **`Vertex`** (or **`VertexExt`**).
- [illegible]
- The **`Texture`** keyword accepts `NULL` as its argument, in which case any texture associated with the current material is removed. From that point on, no texture is applied to the geometry created.

The following is a simple, example script which builds a clump that consists of a red cube:

```
ModelBegin
   ClumpBegin
        Surface 0.2 0.3 0.7 # shiny
        Color 1.0 0.0 0.0   # red
        Block 0.5 0.5 0.5
   ClumpEnd
ModelEnd
```

The **`Surface`** keyword sets ambient, diffuse, and specular reflection coefficients of the current material to 0.2, 0.3 and 0.7 respectively.

The current materials color is set to red using the **`Color`** keyword.

The **`Block`** keyword adds a block (whose width, height, and depth are all 0.5) to the current clump under construction. The current transformation (in this case the identity) is applied to the polygons being added and the materials of these polygons are set to the current material.

Object Builder API Functions

The following table summarizes the Object Builder API functions which mirror each script keyword. For a detailed description of each Object Builder API function see the Function Reference section of this manual.

Script Keyword	API Function
`AddHint`	**`RwAddHint()`**
`AddTextureMode`	**`RwAddTextureModeToSurface()`**
`Ambient`	**`RwSetSurfaceAmbient()`**
`AxisAlignment`	**`RwSetAxisAlignment()`**
`Block`	**`RwBlock()`**
`ClumpBegin`	**`RwClumpBegin()`**
`ClumpEnd`	**`RwClumpEnd()`**
`Color`	**`RwSetSurfaceColor()`**
`Cone`	**`RwCone()`**
`Cylinder`	**`RwCylinder()`**
`Diffuse`	**`RwSetSurfaceDiffuse()`**
`Disc`	**`RwDisc()`**
`GeometrySampling`	**`RwSetSurfaceGeometrySampling()`**
`Hemisphere`	**`RwHemisphere()`**
`Hints`	**`RwSetHints()`**
`Identity`	**`RwIdentityCTM()`**
`IdentityJoint`	**`RwIdentityJointTM()`**
`Include`	**`RwInclude()`**
`IncludeGeometry`	**`RwIncludeGeometry()`**
`JointTransformBegin`	**`RwJointTransformBegin()`**
`JointTransformEnd`	**`RwJointTransformEnd()`**
`LightSampling`	**`RwSetSurfaceLightSampling()`**
`MaterialBegin`	**`RwMaterialBegin()`**
`MaterialEnd`	**`RwMaterialEnd()`**
`ModelBegin`	**`RwModelBegin()`**
`ModelEnd`	**`RwModelEnd()`**

Opacity	**RwSetSurfaceOpacity()**
Polygon	**RwPolygon()**
PolygonExt	**RwPolygonExt()**
ProtoBegin	**RwProtoBegin()**
ProtoEnd	**RwProtoEnd()**
ProtoInstance	**RwProtoInstance()**
ProtoInstanceGeometry	**RwProtoInstanceGeometry()**
Quad	**RwQuad()**
QuadExt	**RwQuadExt()**
RemoveHint	**RwRemoveHint()**
RemoveTextureMode	**RwRemoveTextureModeFromSurface()**
Rotate	**RwRotateCTM()**
RotateJoint	**RwRotateJointTM()**
Scale	**RwScaleCTM()**
Specular	**RwSetSurfaceSpecular()**
Sphere	**RwSphere()**
Surface	**RwSetSurface()**
Tag	**RwSetTag()**
Texture	**RwSetSurfaceTexture()**
TextureDithering	**RwSetTextureDithering()**
TextureExt	**RwSetSurfaceTextureExt()**
TextureGammaCorrection	**RwSetTextureGammaCorrection()**
TextureModes	**RwSetSurfaceTextureModes()**
Trace	None
Transform	**RwTransformCTM()**
TransformBegin	**RwTransformBegin()**
TransformEnd	**RwTransformEnd()**
TransformJoint	**RwTransformJointTM()**
Translate	**RwTranslateCTM()**
Triangle	**RwTriangle()**

`TriangleExt`	**`RwTriangleExt()`**
`Vertex`	**`RwVertex()`**
`VertexExt`	**`RwVertexExt()`**

Trace has no direct Object Builder equivalent. However, the API function RwSetDebugScriptState() performs a similar function.

Platform Specific Information

Related Topics

MS Windows Specific Information
MS Dos Specific Information
Other Platforms

MS Windows Specific Information

Related Topics

Requirements
Environment Variables
RenderWare Library Configuration
RenderWare Dynamic Link Libraries (DLLs)
Compilers
Libraries and Include Files
Watcom C/386 Compiler
Microsoft Visual C++ V1.5
Borland C++ V4.0
Microsoft Visual C++ V2.0
RenderWare and Windows Bitmap Types
RenderWare and Windows Palettes
Device-Specific API Parameters

Requirements

The fixed-point RenderWare library requires an IBM PC compatible with an Intel Pentium, 80486DX, 80486SX, 80386DX or 80386SX CPU (or equivalent), 4Mb of memory, and a color VGA or SuperVGA display adapter.

The recommended minimum configuration for the fixed-point RenderWare library is an Intel 80486SX/25 with 8Mb of memory. For highest performance rendering a display adapter running in 8-bit (256 color) mode is recommended. For highest quality rendering a display adapter running in 16-bit (65536 color) mode is recommended.

The floating-point RenderWare library requires an IBM PC compatible with an Intel Pentium, 80486DX or 80386 and 387 math co-processor (or equivalent), 4Mb of memory, and a color VGA or SuperVGA display adapter.

The recommended minimum configuration for the floating-point RenderWare library is an Intel 80486DX/25 with 8Mb of memory. For highest performance rendering a display adapter running in 8-bit (256 color) mode is recommended. For highest quality rendering a display adapter running in 16-bit (65536 color) mode is recommended.

[illegible] memory

[illegible] the Windows operating system.

- Windows 95 [illegible]
- Windows for Workgroups Version 3.1 or 3.11
- Windows NT Version 3.5.

In order to build programs with RenderWare you will need one of the compilers detailed in the compilers section of this Appendix.

Environment Variables

The RenderWare library makes use of several environment variables, `RWSHAPEPATH`, `RWDEBUGSTREAM` etc. These environment variables are optional. The library will operate correctly if they are not set.

However, if the environment variables are to be employed they must be set before entering Windows. Setting the environment variable from a DOS window running under Windows will not work. It is strongly recommended that the necessary environment variables be set in the host machines `AUTOEXEC.BAT`.

RenderWare Library Configuration

Support for the RenderWare Windows initialization file `winrw.ini` has been removed from this version of RenderWare. All library configuration is now accomplished through the API function **`RwOpenExt`**`()`.

RenderWare Dynamic Link Libraries (DLLs)

- `rwld.dll` Floating-point, debugging RenderWare DLL.
- `rwlw.lib` Floating-point, Watcom C import library.

Applications wishing to use the DLL must link against the appropriate static library.
The static libraries load the appropriate DLLs using the standard Windows **`LoadLibrary`**`()` function. Hence, the DLLs can be placed anywhere on the **`LoadLibrary`**`()` search path. However, the recommended location for the DLLs is in the Windows System directory (this will normally be `c:\windows\system`). The RenderWare setup program will install the retail DLLs in the Windows System directory at installation time.

Related Topics

The Debugging DLLs
Limitations of the DLL
Troubleshooting the DLL

The Debugging DLLs

The mechanism for utilizing the debugging kernel of the RenderWare library is different depending on whether static or dynamic linking is to be employed. For statically linked programs special static debugging libraries are supplied with RenderWare. To switch from using the retail to debugging libraries the program must be relinked against a different library. When using the DLL, no such relinking is necessary. The static import libraries provided by RenderWare are suitable for either retail of debugging versions of RenderWare. The static import libraries always attempt to load the same DLL (the fixed-point import libraries load `rwx.dll` and the floating-point libraries load `rwl.dll`). Therefore, to switch from using the retail DLL to the debugging DLL simply overwrite the existing retail DLL in `c:\windows\system` with the debugging DLL. For example, for fixed-point applications, switching from retail to debugging RenderWare kernels is achieved as follows;

```
copy c:\rwwin\lib\rwxd.dll c:\windows\system\rwx.dll
```

(assuming RenderWare for windows was installed in `c:\rwwin` and the Window System directory is `c:\windows\system`).

Switching back to retail libraries simply involves overwriting the debugging DLL with the retail one;

```
copy c:\rwwin\lib\rwx.dll c:\windows\system\rwx.dll
```

To simplify this process, RenderWare provides two simple batch files (`rwn2d.bat` and `rwd2n.bat`) which will switch between retail and debugging DLLs and back again. These batch files are located in the `lib` directory of the RenderWare distribution. For example, to switch to debugging DLLs;

```
cd c:\rwwin\lib
rwn2d
```

To switch back;

```
cd c:\rwwin\lib
rwd2n
```

(assuming RenderWare for Windows was installed in `c:\rwwin`).

Please note, these batch files assume that the RenderWare DLLs are installed in the Windows System directory and that the Windows System directory is `c:\windows\system`.

Limitations of the DLL

Currently the DLL supports the vast majority of the RenderWare API. However, the DLL does not support RenderWares *user-draw* functionality and the debugging function **RwSetDebugStream**(). The unsupported API calls are as follows;

RwAddUserDrawToClump()
RwCreateUserDraw()
RwDestroyUserDraw()
RwDuplicateUserDraw()
RwForAllUserDrawsInClump() (and its variants)
RwGetClumpNumUserDraws()
RwGetUserDrawAlignment()
RwGetUserDrawCallback()
RwGetUserDrawData()
RwGetUserDrawOffset()
RwGetUserDrawOwner()
RwGetUserDrawParentAlignment()
RwGetUserDrawSize()
RwGetUserDrawType()
RwGetUserDrawVertexIndex()
RwRemoveUserDrawFromClump()
RwSetDebugStream()
RwSetUserDrawAlignment()
RwSetUserDrawCallback()
RwSetUserDrawData()
RwSetUserDrawOffset()
RwSetUserDrawParentAlignment()
RwSetUserDrawSize()
RwSetUserDrawType()
RwSetUserDrawVertexIndex()

Furthermore, the RenderWare DLLs currently allow access to a single client application only. Once one application using the DLL has successfully opened the RenderWare library other applications will not be permitted to open the library. A dialog box will be displayed stating that the RenderWare DLL is already in use and **RwOpen**() (or **RwOpenExt**())will fail.

Troubleshooting the DLL

- *During the development process my application crashed. I have fixed the problem but each time I try to run the application I get a dialog box stating that RenderWare is already in use and RwOpen() fails*
  When an application crashes it will, almost certainly leave RenderWare open and in an unstable state. All future requests to open the library will be refused. The only safe solution to this problem is to restart Windows. However, as this can be laborious when debugging applications the DLL version of RenderWare provides an additional function, **_rwResetReferenceCount**`()` which will let an application connect to the RenderWare DLL after a crash. Although **_rwResetReferenceCount**`()` lets an application connect to the RenderWare DLL, the DLL may be in an unstable state and so the connecting application may still crash. Hence, **_rwResetReferenceCount**`()` must be used with caution and it should never be used in a shipping product. **_rwResetReferenceCount**`()` is not a formal part of the RenderWare API is not exported by the RenderWare include files. The recommended way of using **_rwResetReferenceCount**`()` is as follows;

```
extern void _rwResetReferenceCount(void);
#if defined(_DEBUG)
_rwResetReferenceCount();
#endif
if (!RwOpen("MSWindows", NULL))
{
...
```

- *I have used `rwn2d.bat` to switch to the debugging versions of the RenderWare DLLs and yet I still dont get any debugging information*
  If you are using a file sharing mechanism such as `share.exe` and you have a RenderWare application running or a RenderWare application has crashed leaving the DLL in memory then the DLL file will be locked and it will not be possible to overwrite it. For this reason it is often better to exit Windows before switch to the debugging DLLs.
- *I am using Watcom C and the DLL. The function RwGetRasterPixels() does not return a valid pointer*
  When using Watcom C with the DLL, **RwGetRasterPixels**`()` returns a 16-bit far pointer to the raster pixels. To use this pointer from Watcom C it is necessary to convert the pointer to a 32-bit far pointer using the Watcom supplied macro `MK_FP32()`. Furthermore, when releasing these pixels back to RenderWare with **RwReleaseRasterPixels**`()` it is necessary to convert this pointer back to a 16-bit far pointer using the macro `MK_FP16()`. For example;

```
BYTE far *pixels;

pixels = (BYTE far *)MK_FP32(RwGetRasterPixels(raster));
...
/* Use the pointer. */
...
RwReleaseRasterPixels(raster, MK_FP16(pixels));
```

Compilers

The following table shows the compilers and versions of the Windows operating system that are currently supported by RenderWare. Your choice of compiler and RenderWare library will depend on both the preferred development environment and the target application environment. These need not be the same.

Compilers

	Development Platforms	**Target Platforms**
Watcom C/386 (V9.5 or V10.0) Libraries are provided for both static linking or binding to the 32 bit RenderWare DLL.	Windows 3.1x	Windows 3.1x Windows 95 (using WinG or DIBs) Windows NT 3.5 (using WinG or DIBs)
Microsoft Visual C++ V1.5 (16 bit) 16 bit bindings are provided for linking to the 32 bit RenderWare DLL	Windows 3.1x	Windows 3.1x Windows 95 (using WinG or DIBs) Windows NT 3.5 (using WinG or DIBs)
Microsoft Visual C++ V2.0 (32 bit) Static libraries are provided	Windows 95 Windows NT 3.5	Windows 3.1x (using Win32s and WinG or DIBs) Windows 95 Windows NT 3.5
Borland C++ V4.0 (16 bit) 16 bit bindings are provided for linking to the 32 bit RenderWare DLL	Windows 3.1x	Windows 3.1x Windows 95 (using WinG or DIBs) Windows NT 3.5 (using WinG or DIBs)

Libraries and Include Files

For the purposes of describing how to build programs, the following assumptions are made:

- the RenderWare include files are installed in `\rwwin\include`.
- the RenderWare library files are installed in `\rwwin\lib`.

Source files must include the RenderWare include file:

```
#include <rwlib.h>
```

If your application use the platform specific **RwOpenExt()** options or **RwGetDeviceInfo()** information types then the application will also need to include the Windows specific header file `rwwin.h`. However, it is recommended that, to ensure future compatibility, all source files which use RenderWare API functions include `rwwin.h` after including the standard RenderWare include file:

```
#include <rwlib.h>
#include <rwwin.h>
```

The following RenderWare libraries are provided for building Windows applications using RenderWare.

RenderWare Libraries

	Watcom C/386 Libraries
`rwwrlp.lib`	Register passing, floating point, production library.
`rwwrld.lib`	Register passing, floating point, debugging library.
`rwwrxp.lib`	Register passing, fixed point, production library.
`rwwrxd.lib`	Register passing, fixed point, debugging library.
`rwwslp.lib`	Stack passing, floating point, production library.
`rwwsld.lib`	Stack passing, floating point, debugging library.
`rwwsxp.lib`	Stack passing, fixed point, production library.
`rwwsxd.lib`	Stack passing, fixed point, debugging library.
`rwxw.lib`	Fixed point DLL binding library
`rwlw.lib`	Floating point DLL binding library
	Microsoft Visual C++ V1.5 Libraries
`rwxv.lib`	Fixed point DLL binding library
`rwlv.lib`	Floating point DLL binding library
	Borland C++ V4.0 Libraries
`rwxb.lib`	Fixed point DLL binding library
`rwlb.lib`	Floating point DLL binding library
	Microsoft Visual C++ V2.0 Libraries
`rwnlp.lib`	Floating point, production library.
`rwnld.lib`	Floating point, debugging library.
`rwnxp.lib`	Fixed point, production library.
`rwnxd.lib`	Fixed point, debugging library.

Watcom C/386 Compiler

There are two versions of the Watcom compiler currently in popular use. These are V9.5 and V10.0. Both of these compilers require patches to their base release to work with RenderWare. The minimum patch levels required for RenderWare are V9.5c and V10.0a. However, it should be noted that there is a bug in the Watcom 10.0a compiler which prevents its use with the stack based libraries.

Both of the Watcom compilers use the same options for building RenderWare applications. [illegible]

- If you are building a floating-point program you must define the symbol `RWFLOAT` in all modules which make RenderWare function calls or manipulate RenderWare real numbers. You must also link against one of the static libraries `rwwrlp.lib`, `rwwrld.lib`, `rwwslp.lib`, `rwwsld.lib` or `rwlw.lib`.

The following table gives the compiler options which are mandatory when building RenderWare applications:

Mandatory Watcom Compiler Options

	Fixed-Point	Floating-Point

Register Based	`/zW`
	`/5r, /4r or /3r`
	`/mf`
	`/fpc`
	`/DRWFIXED`
		`/zW`
		`/5r, /4r or /3r`
		`/mf`
		`/DRWFLOAT`

Stack Based	`/zW`
	`/5s, /4s or /3s`
	`/mf`
	`/fpc`
	`/DRWFIXED`
		`/zW`
		`/5s, /4s or /3s`
		`/mf`
		`/DRWFLOAT`

The following table gives the compiler options which are recommended but not mandatory when building RenderWare applications:

Recommended Watcom Compiler Options

	Fixed-Point	Floating-Point

Register Based	`/s`
	`/j`
	`/ei`
	`/oneatx`
		`/7`
		`/s`
		`/j`
		`/ei`
		`/oneatx`

Stack Based	`/s`
	`/j`
	`/ei`
	`/oneatx`
		`/7`
		`/s`
		`/j`
		`/ei`
		`/oneatx`

The following linker flags are mandatory:

```
option stack=32768
```
(A 32k stack is the minimum required)

```
system win386
```

For example, when using the fixed-point, register based version of the RenderWare library, the command line to compile the file `foo.c` to the object file `foo.obj` is:

```
wcc386p /I=\rwwin\include /zW /4r /mf /fpc
          /DRWFIXED /s /j /ei /s /oneatx /fo=foo.obj foo.c
```

The command line to link `foo.obj` with the fixed-point, register based Windows 3.1x version of the RenderWare library (`rwwrxp.lib`) to produce `foo.exe` is:

```
wlink option stack=32768 system win386 name foo
          file foo.obj, \rwwin\lib\rwwrxp.lib
```

The command line to bind foo.exe with its resources, producing the final executable `foo.exe`, is:

```
wbind foo -R foo.res
```

For programs with no resources, use the `-n` option:

```
wbind foo -n
```

Microsoft Visual C++ V1.5

16-bit Windows applications are linked to the 32-bit RenderWare DLL via a static library … correct compiler options are specified for the library

Key Points When Building Programs

- If you are building a fixed-point program you must define the symbol `RWFIXED` in all modules which make RenderWare function calls or manipulate RenderWare real numbers. You must also link against the static library `rwxv.lib`. Furthermore, in order that you may run on machines without a floating-point unit you must select a "Floating-Point Calls" options which does not assume the presence of an x87 coprocessor. We recommend you use the "Emulate" option.
- If you are building a floating-point program you must define the symbol `RWFLOAT` in all modules which make RenderWare function calls or manipulate RenderWare real numbers. You must also link against the static library `rwlv.lib`. To get the best speed in your floating-point application we recommend you use the "Inline 80x87 Instructions" (`/FPi87`) option.

The following table gives the compiler options which are mandatory when building RenderWare applications with Microsoft Visual C++ V1.5. All of these options can be accessed through the Options/Project menu in Visual C++.

Mandatory Microsoft Visual C++ V1.5 Compiler Options

	Value		Command Line	
	Fixed Point	Floating Point	Fixed Point	Floating Point
Compiler Options				
Code Generation Calling Convention	C/C++		/Gd	
Memory Model Model	Large		/AL	
Preprocessor Symbols and Macros	RWFIXED	RWFLOAT	/DRWFIXED	/DRWFLOAT
Preprocessor Include Path	\rwwin\include		/I\rwwin\include	
Linker Options				
Input Libraries	,\rwwin\lib\rwxv.lib	,\rwwin\lib\rwlv.lib	/LIB:\rwwin\lib\rwxv.lib	/LIB:\rwwin\lib\rwlv.lib

The following table gives the compiler options which are recommended but not mandatory when building RenderWare applications with Microsoft Visual C++ V1.5. All of these options can be accessed through the Options/Project menu in Visual C++.

Recommended Microsoft Visual C++ V1.5 Compiler Options

	Value		Command Line	
	Fixed Point	Floating Point	Fixed Point	Floating Point
Compiler Options				
Code Generation CPU	80386		/G3	

Code Generation Code Generator	Optimizing			/f-
Code Generation Floating Point Calls	Use Emulator	Inline 8087 Instructions	default	/FPi87
Custom Options Warning Level	Level 4			/W4

Borland C++ V4.0

[illegible]

- If you are building a floating-point program you must define the symbol `RWFLOAT` in all modules which make RenderWare function calls or manipulate RenderWare real numbers. You must also link against the static library `rwlb.lib`.

The following table gives the compiler options which are mandatory when building RenderWare applications with Borland C++ V4.0. The menu that is used to access the options is show as XX:YY, this means the YY option on menu XX.

Mandatory Borland C++ V4.0 Compiler Options

	Value		Command Line	
	Fixed Point	Floating Point	Fixed Point	Floating Point
Project: New Project				
Target Type Platform	`Windows 3.x (16)`			
Target Type Target Model	`Large`		`-ml`	
Options: Project				
Directories Include	`...;\rwwin\include`		`-I\rwwin\include`	
Compiler Defines	`...; RWFIXED`	`...; RWFLOAT`	`-DRWFIXED`	`-DRWFLOAT`
Compiler Code Generation	`Allocate Enums as Ints`		`-b`	
16 bit Compiler Calling Convention	`C`		`-pc`	
Other				
Add the following node to your project .	`\rwwin\lib\rwxb.lib`	`\rwwin\lib\rwlb.lib`		

The following table gives the compiler options which are recommended but not mandatory when building RenderWare applications with Borland C++ V4.0. The menu that is used to access the options is show as XX:YY, this means the YY option on menu XX.

Recommended Borland C++ V4.0 Compiler Options

Options: Project	Value	Command Line
Compiler Floating Point	`Fast floating point`	`-ff`
16 bit Compiler Instruction Set	`80386 (or i486)`	`-3/-4`
Optimizations Specific	`Select Code In Favor Of Executable Speed`	`-O2`
Optimizations Specific	`Common Sub-expressions`	`-Og`

	Optimize Globally	
Optimizations Speed	Inline intrinsic functions	-Oi
	Invariant code motion	-Om
	Copy Propagation	-Op
	Induction variables	-Ov
Messages	All	-w

Microsoft Visual C++ V2.0

[illegible]

- If you are building a floating-point program you must define the symbol `RWFLOAT` in all modules which make RenderWare function calls or manipulate RenderWare real numbers. You must also link against one of the floating point static libraries `rwnlp.lib` or `rwnld.lib`.

The following table gives the compiler options which are mandatory when building RenderWare applications with Microsoft Visual C++ V2.0. All of these options can be accessed through the Project/Settings menu in Visual C++ V2.0.

Mandatory Microsoft Visual C++ V2.0 Compiler Options

	Value		Command Line	
	Fixed Point	Floating Point	Fixed Point	Floating Point
Compiler Options				
Code Generation Calling Convention	`_cdecl`		default	
Code Generation Use Run-time Library	`Multithreaded`		`/MT`	
Preprocessor Preprocessor Definitions	`..., RWFIXED`	`..., RWFLOAT`	`/DRWFIXED`	`/DRWFLOAT`
Preprocessor Include Directories	`\rwwin\include`		`/I\rwwin\include`	
Link Options				
Input Object/Library Modules	`\rwwin\lib\rwnxp.lib`	`\rwwin\lib\rwnlp.lib`	`\rwwin\lib\rwnxp.lib`	`\rwwin\lib\rwnlp.lib`

The following table gives the compiler options which are recommended but not mandatory when building RenderWare applications with Microsoft Visual C++ V2.0. All of these options can be accessed through the Project/Settings menu in Visual C++.

Recommended Microsoft Visual C++ V2.0 Compiler Options

	Value	Command Line
Compiler Options		
Code Generation Processor	`Pentium`	`/G5`
General Optimizations	`Maximize Speed`	`/O2`
General Warning Level	`Level 4`	`/W4`

RenderWare and Windows Bitmap Types

When RenderWare is operating as a software only rendering service it makes extensive use of the bitmap handling facilities of the underlying operating system. These operating system bitmaps are used to store the results of RenderWares rendering and, when software double buffering, to copy this rendering to the output display.

The Microsoft Windows operating system family provide several different bitmap types namely; Device Dependent Bitmaps (`HBITMAP`s), Device Independent Bitmaps (DIBs), WinG bitmaps and DIB Sections. These bitmap types vary widely in the facilities they offer and in the speed of the operations which act upon them. Furthermore, not all of these bitmap types are available on all versions of Windows and those that are provide different capabilities and performance on different hardware configurations. Therefore, there is not a single best bitmap type. The decision on which bitmap type must be taken at run-time on the basis of a number of factors including the host operating system and the color resolution of the display device.

RenderWare V1.4 provides an intelligent mechanism for selecting the fastest bitmap type available when a RenderWare executable is run. This mechanism allows, for example, a Win32 executable built under Windows NT 3.5 to run efficiently under Windows 3.1x.

For the majority of applications the selection of bitmap type will be entirely transparent to the application developer. However, for those developers who need to interact with RenderWare at a low-level to achieve special effects this section discusses the algorithm used to select the bitmap type. The means by which a developer can fine tune this process and enquire about the chosen bitmap type are also discussed. Initially a short description of each bitmap type will be given.

- Device Dependent Bitmaps (`HBITMAP`s)
- Device Independent Bitmaps (DIBs)
- DIB Sections
- WinG bitmaps

  WinG is a fast bitmap handling mechanism designed for Windows 3.1x (but also available for Windows 95 and Windows NT 3.5). Although available on all Windows operating systems, WinG is not a core part of any of the operating systems and so may not be present on a target machine (although WinG is freely redistributable and can be included with a shipping product). Also, WinG only supports 8-bit bitmaps so it cannot be used when RenderWare is performing 16-bit rendering.

  On Windows 95 and Windows NT 3.5 WinG is simply a reduced functionality front-end to DIB Sections and hence DIB Sections should be used in preference to WinG for Win32 applications running under Windows 95 or Windows NT 3.5. Furthermore, on Windows 3.1 `HBITMAP`s are faster and are normally preferred. WinG does have one significant advantage over `HBITMAP`s in that it supports fast bitmap stretching. See the section on page - for details.

Related Topics

How the Bitmap Type is Chosen
Overriding the Choice of Bitmap Type
Determining the Choice of Bitmap Type
Bitmap Stretching

How the Bitmap Type is Chosen

The choice of bitmap handling mechanism is based on three factors:

- Executable type (Win16/Win386 or Win32)
- Run-time operating system (Windows 3.1x or Windows95/Windows NT 3.5)
- Depth of the output device (normally 4, 8, 16 or 24-bit)

The following table summarizes the choice of bitmap type for each combination of the above factors. Whenever the bitmap type is given as WinG or DIBs WinG will be used if it has been installed, otherwise RenderWare will fail over to using DIBs.

	Running under Windows 3.1x		**Running under Windows 95 or Windows NT 3.5**	
	8 or 16-bit output device	4 or 24-bit output device	4 or 8-bit output device	16 or 24-bit output device
Win16/ Win386 Executables	`HBITMAP`s (8 and 16-bit rendering)	WinG or DIBs (8-bit rendering only)	WinG or DIBs (8-bit rendering only)	
Win32 Executables	WinG or DIBs (8-bit rendering only)		DIB Sections (8-bit rendering)	DIB Sections (16-bit rendering)

Overriding the Choice of Bitmap Type

In certain circumstances it may be desirable to override RenderWares default choice of bitmap type. For example, if you wish to print or save the results of RenderWares rendering it may be more convenient to use DIBs than the other bitmap types. Furthermore, WinGs fast bitmap stretching can give a significant performance boost when rendering to a large viewport (see the section on page -).

RenderWare allows the application developer to specify that they wish to favor either WinG or DIBs over RenderWares default choice of bitmap type. This is done by supplying additional arguments (`rwWINUSEDIBS` or `rwWINUSEWING`) when opening the library with the API call **`RwOpenExt()`**.

To specify DIBs, `rwWINUSEDIBS` should be given as one of the additional arguments to **`RwOpenExt()`**. For example;

```
RwOpenArgument arg;

arg.option = rwWINUSEDIBS;
arg.value  = 0L; /* This value is ignored for rwWINUSEDIBS */
if (!RwOpenExt(MSWindows, NULL, 1, &arg))
{
     ...
```

To specify WinG, `rwWINUSEWING` should be given as one of the additional arguments to **`RwOpenExt()`**. For example;

```
RwOpenArgument arg;

arg.option = rwWINUSEWING;
arg.value = 0L; /* This value if ignored for rwWINUSEWING */
if (!RwOpenExt(MSWindows, NULL, 1, &arg))
{
     ...
```

It is important to note that if `rwWINUSEWING` is specified and WinG has not been installed on the host machine, **`RwOpenExt()`** will not fail. RenderWare will, instead, attempt to find a bitmap type using the default mechanism. If it is essential that an application use WinG (and no other bitmap type), the application should specify `rwWINUSEWING` and, after the library has been opened, check to see if WinG has been selected. If it has not, the application can then close RenderWare, display an error message and exit. The next section describes how to determine which bitmap type has been selected.

Determining the Choice of Bitmap Type

It is sometimes important to determine which bitmap type RenderWare has chosen. This can be done using the API function **`RwGetDeviceInfo`**`()` and the device information types `rwWINUSINGDIBS` and `rwWINUSINGWING`.

To determine whether RenderWare is using DIBs the following code fragment would be used;

```
RwBool usingDIBs;

RwGetDeviceInfo(rwWINUSINGDIBS, &usingDIBs, sizeof(usingDIBs));
if (usingDIBs)
{
     /* RenderWare is using DIBs... */
```

To determine whether RenderWare is using WinG the following code fragment would be used;

```
RwBool usingWinG;

RwGetDeviceInfo(rwWINUSINGWING, &usingWinG, sizeof(usingWinG));
if (usingWinG)
{
     /* RenderWare is using WinG... */
```

If neither `rwWINUSINGDIBS` or `rwWINUSINGWING` yields a non-zero result then the default bitmap type is being used. For a Win16/Win386 executable the default type is `HBITMAP`s and for a Win32 executable the default type is DIB Sections.

Bitmap Stretching

When performing software only rendering, the size of viewport to which RenderWare renders can have a significant impact on performance, particularly if a high per-pixel cost rendering mode is employed (such as lit, smooth shaded, foreshortened texture mapping). One approach to improving performance is *bitmap stretching* where RenderWare renders to a small viewport and the rendering is then stretched up to fill a significantly larger rectangle on the output device.

In RenderWare control over bitmap stretching is provided by the `rwWINSETOUTPUTSIZE` device control. This device control specifies the width and height of the rectangle on the output device that the viewport will be stretched to fill (control over rendering resolution is still achieved through the function call **`RwSetCameraViewport()`**).

When specifying stretching, the arguments to **`RwDeviceControl()`** are as follows:

`action` `rwWINSETOUTPUTSIZE`

`param1` Not used (pass 0)

`param2` A pointer to an `RwWinOutputSize` structure as described below;

```
typedef struct
{
     RwInt32   width;  /* Width of the output */
     RwInt32   height; /* Height of the output */
     RwCamera *camera; /* Camera whose output is
                          to be stretched */
} RwWinOutputSize;
```

`size` The size of `RwWinOutputSize` (i.e., `sizeof(RwWinOutputSize)`).

This control specifies the actual output width and height desired for the given camera. For example, the following sets the output size of the camera `Camera` to 640 by 480 pixels;

```
RwWinOutputSize winOutputSize;

winOutputSize.width  = (RwInt32)640;
winOutputSize.height = (RwInt32)480;
winOutputSize.camera = Camera;
RwDeviceControl(rwWINSETOUTPUTSIZE, 0, &winOutputSize,
                    sizeof(winOutputSize));
```

RenderWare's rendering resolution is set by the viewport width and height specified by a call to **`RwSetCameraViewport()`**. For example, the following code fragment sets the rendering resolution to 320 by 240 pixels:

```
RwSetCameraViewport(Camera, 0, 0, 320, 240);
```

The above code fragments will result in RenderWare rendering at a resolution of 320 by 240 and stretching the result to a rectangle of size 640 by 480 (offset at (0, 0) from the output devices origin).

The stretching is actually perfomed when **`RwShowCameraImage()`** is called to perform software double buffering. The offset from the origin of the device context passed to **`RwShowCameraImage()`** of the output is given by the viewport offset specified in the call to **`RwSetCameraViewport()`**.

[illegible]

- [illegible]
- Bitmap stretching operates much more quickly when the bitmap is stretched by a power of 2. Therefore, it is best to set the viewport width and height of the camera to half the width and height of the output window. For example, when setting RenderWare's viewport in response to a `WM_SIZE` message: [illegible]
- The output size specified by **`RwDeviceControl()`** only effects the very last stage of bitmap copying performed by **`RwShowCameraImage()`**. The stretching specified by **`RwDeviceControl()`** is not taken into account by any other RenderWare API functions which take viewport coordinates as arguments. This is particularly important to remember when picking using the API functions **`RwPickScene()`**`RwPickScene`and

RwPickClump(). The mouse coordinates passed to an application by Windows will be in device (client window) coordinates. If a RenderWare application is making use of bitmap stretching the mouse coordinates must be transformed into the cameras viewport space before being passed to RenderWare. For example (when output is being stretched by a factor of 2) picking would be performed as follows:

```
case WM_LBUTTONDOWN:
     {
#if defined(WIN32)
         POINTS pos;
         pos = MAKEPOINTS(lParam);
#else
         POINT pos;
         pos = MAKEPOINT(lParam);
#endif
         RwPickScene(Scene, pos.x / 2, pos.y / 2,
                     Camera, &pick);
```

RenderWare and Windows Palettes

RenderWare maintains its own Windows palette object (`HPALETTE`). Each time **`RwShowCameraImage()`** is called this palette is selected into the specified device context and realized. By default, RenderWare realizes its palette as a foreground palette, i.e., `FALSE` is passed as the third parameter of `SelectPalette()`. With RenderWare V1.4 it is possible to change this default and have RenderWare realize its palette as a background palette. This is useful when building applications where RenderWare has to co-exist with other windows which also have their own palettes (particularly MDI applications). In such situations it is not acceptable for the RenderWare window to realize its palette as a foreground palette on each call to **`RwShowCameraImage()`**. Instead, RenderWare should realize its palette as a background palette and the application must take responsibility for realizing the RenderWare palette as a foreground palette at the appropriate time, i.e., in response to a `WM_QUERYNEWPALETTE` message.

To following code instructs RenderWare to realize its palette as a background palette;

```
RwDeviceControl(rwWINBACKGROUNDPALETTE, TRUE, NULL, 0L);
```

If an application uses the above control it is essential that it take responsibility for realizing the RenderWare palette as a foreground palette when the RenderWare window receives the `WM_QUERYNEWPALETTE` message.

The following code fragment demonstrates this process;

```
case WM_QUERYNEWPALETTE:
     {
          HPALETTE rwPalette;
          HPALETTE oldPalette;
          HDC      dc;
          int      numChanges;

          RwGetDeviceInfo(rwPALETTE, &rwPalette,
                   sizeof(rwPalette));
          dc = GetDC(window);
          oldPalette = SelectPalette(dc, rwPalette, TRUE);
          numChanges = RealizePalette(dc);
          SelectPalette(dc, oldPalette, FALSE);
          ReleaseDC(window, dc);
          if (numChanges > 0)
               InvalidateRect(window, NULL, FALSE);
     }
     break;
```

The following code fragment switches RenderWare back to realizing its palette as a foreground palette;

```
RwDeviceControl(rwWINBACKGROUNDPALETTE, FALSE, NULL, 0L);
```

It is possible to determine whether RenderWare is realizing its palette in the foreground and background using the `rwWINISBACKGROUNDPALETTE` device information type. For example;

```
RwBool isBackPal;

RwGetDeviceInfo(rwWINISBACKGROUNDPALETTE, &isBackPal,
     sizeof(isBackPal));
if (isBackPal)
{
     /* Background palette realization... */
}
else
{
     /* Foreground palette realization... */
}
```

Device-Specific API Parameters

A small number of RenderWare API functions have device dependent parameters or return values. This section describes these device dependent parameters and return values under Windows.

```
RwCamera *
RwBeginCameraUpdate(RwCamera *cam, void *param);
```

Arguments

`param` A handle to the output window (if any). If the results of the rendering operations following **`RwBeginCameraUpdate()`** are to be displayed in an output window by **`RwShowCameraImage()`** the handle of the output window must be passed as the argument `param`. If the rendering is not to be displayed in a window (for example, if it is to be printed) pass `param` as `NULL`. Note, it is also necessary pass a DC associated with the output window when **`RwShowCameraImage()`** is called.

For example;

```
RwBeginCameraUpdate(camera, (void *)window);
```

```
RwRaster *
RwBitmapRaster(void *bitmap, RwRasterOptions options);
```

Arguments

`bitmap` A pointer to a structure of type `RwWinBitmapRaster` as described below.

```
typedef struct {
     HDC      hdc;
     HBITMAP  hBitmap;
} RwWinBitmapRaster;
```

The device dependent bitmap can be of any size but should have a depth equal to the depth of display adapter on which Windows is running.

If running on an 8-bit display, the device context must have a palette object selected into it. The palette object provides the color table of the device dependent bitmap.

It is essential that the device dependent bitmap is not selected into the device context when **`RwBitmapRaster`**`()` is invoked.

```
RwCamera *
RwCreateCamera(RwInt32 maxwidth, RwInt32 maxheight, void *param);
```

Arguments

`param`	Under Windows, camera image buffer sharing is not currently supported. This parameter should always be `NULL`.

RwInt32
RwDeviceControl(RwDeviceAction action, RwInt32 param1, void *param2, RwInt32 size);

Arguments

action Under Windows, RenderWare provides the following device controls actions:

- rwWINSETOUTPUTSIZE

  This action is used to control the stretching of a camera's viewport [illegible] if [illegible] Appendix [illegible] bitmap stretching in general see the section on page [illegible] of this [illegible]

- rwWINBACKGROUNDPALETTE

  This action is used to control how RenderWare realizes its Windows palette. To make RenderWare realize its palette as a background palette pass a non-zero value for param1. To make RenderWare realize its palette as a foreground palette pass zero for param1. In either case param2 and size are ignored. This control returns TRUE if successful, and FALSE otherwise. See the section on page - for further details.

Comments

Please note, these control actions are highly device dependent and, as such, may be significantly modified or even dropped from future releases of RenderWare.

```
RwCamera *
RwDuplicateCamera(RwCamera *cam, void *param);
```

Arguments

`param` Under Windows, camera image buffer sharing is not currently supported. This parameter should always be `NULL`.

```
void *
RwGetCameraImage(RwCamera *cam);
```

Return Value

The image buffer of a RenderWare camera can be either a memory device context with a memory bitmap selected into it or a device independent bitmap (DIB).

To determine whether **RwGetCameraImage()** returns a memory device context or DIB use **RwGetDeviceInfo()** as follows:

```
RwCamera *cam;
RwInt32 usingDIBs;
HDC hdc;
BITMAPINFOHEADER *dib;

RwGetDeviceInfo(rwWINUSINGDIBs, &usingDIBs,  sizeof(usingDIBs));
if (usingDIBs)
     dib = (BITMAPINFOHEADER *)RwGetCameraImage(cam);
else
     hdc = (HDC)RwGetCameraImage(cam);
```

RwBool
RwGetDeviceInfo(RwDeviceInfo info, void *value, RwInt32 size);

Comments

The Windows specific aspects of each device information type are as follows:

`rwRENDERDEPTH`	The current render depth. See the section on page - for a description of how the render depth is derived.
`rwINDEXEDRENDERING`	As the generic **RwGetDeviceInfo**().
`rwPALETTEBASED`	Rendering is palette based if and only if the video adapter on which Windows is running is palette based. This will normally be the case if the video adapter is in 8-bit (256) color mode. In other modes rendering will not be palette based.

The following options apply when the output device is palette based:

`rwPALETTE`	Returns the GDI palette object RenderWare selects into the output device context. `value` should point to a variable of type `HPALETTE`. On return from **RwGetDeviceInfo**() this variable will contain the handle of the GDI palette object. Applications should not modify this palette object using the Windows API function `SetPaletteEntries()`. Instead the RenderWare V1.4 API function **RwSetPaletteEntries**() should be used to modify the RenderWare palette.
`rwPALETTESIZE`	As the generic **RwGetDeviceInfo**().
`rwFIRSTPALETTEENTRY`	As the generic **RwGetDeviceInfo**().
`rwLASTPALETTEENTRY`	As the generic **RwGetDeviceInfo**().

RenderWare V1.4 support the following Windows specific information types:

`rwWINIMAGEISDIB`	A pointer to a boolean (RwBool) which will be non-zero if **RwGetCameraImage**() returns a device independent bitmap (DIB) and zero if a memory device context is returned. See the description of **RwGetCameraImage**()RwGetCameraImagein this section for further details. `rwWINIMAGEISDIB` is synonymous with `rwWINUSINGDIBS`. `rwWINIMAGEISDIB` is obsolete and will be removed from future versions of RenderWare. `rwWINUSINGDIBS` should be used in its place.
`rwWINISBACKGROUNDPALETTE`	A pointer to a boolean (RwBool) which will be non-zero if RenderWare realizes its palette as a background palette and zero if it realizes its palette as a foreground palette. See the section on page - for further details.
`rwWINUSINGDIBS`	A pointer to a boolean (RwBool) which will be non-zero if RenderWare is using device independent bitmaps (DIBs) and zero otherwise. See the section on page - in this Appendix for a detailed description of this information type. `rwWINUSINGDIBS` is synonymous with `rwWINIMAGEISDIB`. `rwWINIMAGEISDIB` is obsolete and will be removed from future versions of RenderWare. `rwWINUSINGDIBS` should be used in its place.

`rwWINUSINGWING`	A pointer to a boolean (`RwBool`) which will be non-zero if RenderWare is using WinG and zero otherwise See the section on page - in this Appendix for a detailed description of this information type.

RwBool

RwOpen(char *devname, void *param);

Arguments

devname The device name as a null-terminated string. Names currently supported under Windows are:

```
MSWindows
MSWindowsWinG
NullDevice
```

MSWindowsWinG is identical to MSWindows except that it will attempt to use WinG as RenderWares bitmap type (see the section on page - for a discussion of the different bitmap types). However, MSWindowsWinG is obsolete and the **RwOpenExt**() option rwWINUSEWING should be used in its place.

The NullDevice driver allows the library to be used when output to a display is not required (for instance, when reading from or writing to files).

param For Windows platforms, this parameter should be NULL.

Comments

The RenderWare Windows initialization file winrw.ini has been removed from this release of RenderWare. Therefore, all library configuration must be achieved using the API function **RwOpenExt**()

```
RwBool
RwOpenExt(char *devname, void *param, RwInt32 numargs,
    RwOpenArgument *args);
```

Arguments

`devname`	The device name as a null-terminated string. Names currently supported under Windows are:
	`MSWindows`
	`MSWindowsWinG`
	`NullDevice`
	`MSWindowsWinG` is identical to `MSWindows` except that it will attempt to use WinG as RenderWares bitmap type (see the section on page - for a discussion of the different bitmap types). However, `MSWindowsWinG` is obsolete and the **`RwOpenExt()`** option `rwWINUSEWING` should be used in its place.
	The `NullDevice` driver allows the library to be used when output to a display is not required (for instance, when reading from or writing to files).

`param`	For Windows platforms, this parameter should be `NULL`.
`numargs`	The number of optional arguments specified.
`args`	An array of optional open arguments.

Comments

`RwOpenExt()` takes a pointer to an array of `RwOpenArgument` structures. `RwOpenArgument` is defined as follows:

```
typedef struct
{
    RwOpenOption option;
    void *value;
} RwOpenArgument;
```

`option` is one of the identifiers defined below and `value` is a parameter specific to the option type. If the parameter type is integer, the integer value should be cast to a `void *`. For example;

```
args[0].value = (void *)10;
```

The configuration options supported under Windows are:

`rwGAMMACORRECT`	Controls whether RenderWare performs gamma correction on its color palette. `value` should be non-zero to enable gamma correction and zero to disable gamma correction.
`rwWINUSEDIBS`	If this option is specified RenderWare will use DIBs to handle its bitmaps. `value` is ignored. See the section on page - for a full discussion of this option.
`rwWINUSEWING`	If this option is specified RenderWare will use WinG to handle its bitmaps. `value` is ignored. If WinG is not installed on the target system RenderWare will attempt to fail over to an alternative bitmap type. See the section on page - for a full discussion of this option.
`rwWINSETWINGDIBORIENT`	Conventionally Device Independent Bitmaps (DIBs) have

a bottom-up orientation, i.e., pixel (0, 0) is at the bottom left of the bitmap rather than the top left. This is not only inconsistent with the rest of Windows but can also slow bitmap operations greatly. For this reason, WinG DIBs can have either bottom-up or top-down orientation. Normally, RenderWare uses the orientation recommended by WinG at run time and the choice of orientation is entirely transparent to the application programmer. However, if it is necessary to integrate an existing 2D rendering package with RenderWare, problems may arise if that package makes assumptions about the orientation of DIBs. For this reason, this option allows the application programmer to override WinGs recommended DIB orientation. If you wish to force a bottom-up orientation pass `value` as a positive (non-zero) integer. If you wish to force a top-down orientation pass `value` as a negative integer. Please note that although this option may make integration with RenderWare easier it may adversely effect performance. Furthermore, this option is only relevant when using WinG. All other bitmap types ignore it.

The following example demonstrates opening the RenderWare library to use WinG with a top-down DIB orientation and with gamma correction enabled:

```
RwOpenArgument args[3];

args[0].option = rwGAMMACORRECT;
args[0].value = (void *)TRUE;
args[1].option = rwWINUSEWING;
args[2].option = rwWINSETWINGDIBORIENT;
args[2].value = (void *)-1;
RwOpenExt(MSWindows, NULL, 2, args);
```

RwBool
RwOpenDebugStream(char *filename);

Arguments

filename Specifying DEBUG: as the name of a debugging stream will result in debugging output being issued by the Windows API function **OutputDebugString**().

Note that when linking against the debugging version of the library, the default debugging stream is \rw.log.

RwCamera *
RwShowCameraImage(RwCamera *cam, void *param);

Arguments

param A handle to the output device context. For example;

```
hdc = GetDC(hwnd);
RwShowCameraImage(cam, (void *)hdc);
ReleaseDC(hwnd, hdc);
```

MS Dos Specific Information

Related Topics

Requirements
Environment Variables
RenderWare Library Configuration
Include Files
Building Programs: Watcom C/386 Compiler
Device-Specific API Parameters

Requirements

The fixed-point RenderWare library requires an IBM PC compatible with an Intel Pentium, 80486DX, 80486SX, 80386DX or 80386SX CPU (or equivalent), 4Mb of memory, and a color VGA or SuperVGA display adapter.

The recommended minimum configuration for the fixed-point RenderWare library is an Intel 80486SX/25 with 4Mb of memory. For highest performance rendering a display adapter running in 8-bit (256 color) mode is recommended. For highest quality rendering a display adapter running in 16-bit (65536 color) mode is recommended.

The floating-point RenderWare library requires an IBM PC compatible with an Intel Pentium, 80486DX or 80386 and 80387 math co-processor (or equivalent), 4Mb of memory and a color VGA or SuperVGA display adapter.

The recommended minimum configuration for the floating-point RenderWare library is an Intel 80486DX/25 with 4Mb of memory. For highest performance rendering a display adapter running in 8-bit (256 color) mode is recommended. For highest quality rendering a display adapter running in 16-bit (65536 color) mode is recommended.

For optimum performance, choose a CPU with a fast external clock speed over a clock-doubled CPU (e.g., 486DX/50 rather then 486DX2/66), 256K external cache, 70ns memory and a fast PCI or VESA-Local Bus display adapter.

Version 9.5c or 10.0a of the Watcom C/386 compiler are required to build programs.

Environment Variables

The RenderWare library makes use of several environment variables, `RWDEBUGSTREAM`, `RWSHAPEPATH` etc. These environment variables are optional. The library will operate correctly if they are not set. It is strongly recommended that the necessary environment variables be set in the host machine's `AUTOEXEC.BAT`.

RenderWare Library Configuration

Previous versions of RenderWare provided control over certain aspects of library [illegible] dosrw.ini.

- [illegible] RenderWare applications, requiring different configurations, simultaneously.
- As the initialization file was plain ASCII text, it was possible for an end user to modify the file and, hence, adversely effect the operation of the RenderWare application.

RenderWare V1.3 addressed these problems by providing API level control over the library configuration with the new **RwOpenExt**() function. This function, an alternative to **RwOpen**(), supports the specification of additional, optional arguments controlling library configuration. However, to ensure backwards compatibility and continued ease of configuration, the function **RwOpen**() still parses the RenderWare configuration file.

RenderWare looks for its configuration file, `dosrw.ini` in the directory pointed to by the `RWHOME` environment variable. (If `RWHOME` is not defined, RenderWare will look in the current working directory for the initialization file dosrw.ini). The format of the configuration file is as follows:

```
[general]
Width = <width>
Height = <height>
Depth = <depth>
Gamma = yes | no
```

The `Gamma` keyword specifies whether RenderWare is to produce a gamma corrected palette. The default value is `no`.

The `Width`, `Height` and `Depth` keywords are used to set the resolution that the video card is going to be placed. The width, height and depth values must be defined and together must specify a resolution that the card can achieve. Supported resolutions are:

Type	Width	Height	Depth
VGA	320	200	8
VESA	640	400	8
VESA	640	480	8
VESA	800	600	8
VESA	1024	768	8
VESA	1280	1024	8
VESA	320	200	16
VESA	640	400	16
VESA	640	480	16
VESA	800	600	16
VESA	1024	768	16
VESA	1280	1024	16
VESA	320	200	15
VESA	640	400	15
VESA	640	480	15
VESA	800	600	15
VESA	1024	768	15
VESA	1280	1024	15

Note that 15 bit modes are a form of 16 bit mode except the color is in the 1:5:5:5 format.

That is one bit unused and 5 bits of color for red, green and blue. Note that RenderWare renders internally to 5:6:5 format and so a fairly costly color conversion process is required in the DOS device driver. 16 bit modes should always be used where possible instead of 15 bit modes for optimal performance.

VGA modes (320 by 200, 8-bit) are available on all VGA cards. The availability of the other resolutions depends on the kind of card installed, and what kind of VESA driver is present.

The VESA modes will not function without a VESA driver being installed (V1.0 or later). If your card does not have a VESA BIOS built in it is likely that there is a VESA TSR (terminate and stay resident) driver to make it appear as if it has a VESA BIOS. Install the TSR and RenderWare for DOS should function correctly.

UNIVBE is a universal VESA TSR package and comes as part of the RenderWare package. The program supports a wide range of video cards. If your card is supported then it is best to use the UNIVBE TSR driver as often it allows more VESA resolutions than a cards native VESA BIOS.

UNIVBE is shareware and no registration is required for its use by RenderWare customers. If UNIVBE is bundled with a product then an agreement must be made with SciTech. See the UNIVBE documentation for details.

RenderWare V1.4 for DOS has been extensively tested with the UNIVBE package, and thus is likely to provide the best performance and stability in this configuration. To find if your card is supported by UNIVBE look in the UNIVBE documentation.

The default mode is a width of 320, height of 200 and depth of 8 (as this mode is available on all VGA cards with or without a VESA driver).

Include Files

For the purposes of describing how to build programs, the following assumptions are made:

- the RenderWare include files are installed in `\rwdos\include`
- the RenderWare library files are installed in `\rwdos\lib`

Source files must include the RenderWare include file:

```
#include <rwlib.h>
```

Version 1.3 of RenderWare introduced a new, MS DOS specific include file `rwdos.h`. This include file is only necessary if your application uses platform specific **`RwOpenExt`**`()` options or **`RwGetDeviceInfo`**`()` information types. However it is recommended that, to ensure future compatibility, all source files which use the RenderWare API functions include `rwdos.h` after including the standard RenderWare include file:

```
#include <rwlib.h>
#include <rwdos.h>
```

Also included is the header file `doswrap.h` which provides simpler access to DOS specific device controls. If you wish to use these functions include `doswrap.h` after `rwlib.h` and `rwdos.h`.

Building Programs: Watcom C/386 Compiler

There are two versions of the Watcom compiler currently in popular use. These are V9.5 and V10.0. Both of these compilers require patches to their base release to work with RenderWare. The minimum patch levels required by RenderWare are 9.5c and 10.0a. However, it should be noted that there is a bug in the Watcom 10.0a compiler which prevents its use with the stack based libraries.

Both of the Watcom compilers use the same options for building RenderWare applications. Under MS DOS, RenderWare V1.4 includes fixed and floating-point libraries with both register and stack based calling conventions. It is essential that the correct compiler options are specified for the library being linked against.

The following table gives the compiler options which are mandatory when building RenderWare applications:

Mandatory Watcom Compiler Options

	Fixed-Point	Floating-Point

Register Based	`/5r, /4r or /3r`
	`/mf`
	`/fpc`
	`/DRWFIXED`
		`/5r, /4r or /3r`
		`/mf`
		`/DRWFLOAT`

Stack Based	`/5s, /4s or /3s`
	`/mf`
	`/fpc`
	`/DRWFIXED`
		`/5s, /4s or /3s`
		`/mf`
		`/DRWFLOAT`

The following table gives the compiler options which are recommended but not mandatory when building RenderWare applications:

Recommended Watcom Compiler Options

	Fixed-Point	Floating-Point

Register Based	`/s`
	`/j`
	`/ei`
	`/oneatx`
		`/7`
		`/s`
		`/j`
		`/ei`
		`/oneatx`

Stack Based	`/s`
	`/j`
	`/ei`
	`/oneatx`
		`/7`
		`/s`
		`/j`
		`/ei`
		`/oneatx`

The following linker flags are mandatory:

`option stack=32768` (A 32k stack is the minimum required)

For example, when using the fixed-point, register based version of the RenderWare library, the command line to compile the file `foo.c` to the object file `foo.obj` is:

```
wcc386p /I=\rwdos\include /4r /mf /fpc /DRWFIXED /s /j
          /ei /s /oneatx /fo=foo.obj foo.c
```

The command line to link `foo.obj` with the fixed-point, register based MS DOS version of the RenderWare library (`rwdrxp.lib`) to produce `foo.exe` is:

```
wlink option stack=32768 name foo file foo.obj,
      \rwdos\lib\rwdrxp.lib
```

Device-Specific API Parameters

A small number of RenderWare API functions have device dependent parameters or return values. This section describes these device dependent parameters and return values under MS DOS.

```
RwCamera *
RwBeginCameraUpdate(RwCamera *cam, void *param);
```

Arguments

`param`	`param` is ignored. It should be passed as `NULL`. For example; `RwBeginCameraUpdate(camera, NULL);`

RwCamera *
RwCreateCamera(RwInt32 maxwidth, RwInt32 maxheight, void *param)

Arguments

param — Must be either NULL or the image buffer of an existing camera (as returned by **RwGetCameraImage()**).

If param is NULL the new camera will allocate its own image buffer of the given width and height. For example in the typical case of output to the full screen, the following code fragment could be used:

```
RwInt32 scrheight, scrwidth;

RwGetDeviceInfo(rwSCRHEIGHT, &scrheight);
RwGetDeviceInfo(rwSCRWIDTH, &scrwidth);
cam = RwCreateCamera(scrwidth, scrheight, NULL);
```

If param is the value returned by calling **RwGetCameraImage()** with an existing camera as an argument, then the new camera will not allocate its own image buffer but will share the image buffer of the existing camera. The existing camera must have exactly the same maximum width and height as those specified in the call to **RwCreateCamera()**. For example, to create a new camera (cam2) sharing the image buffer of the camera (cam) created above the following code fragment could be used:

```
cam2 = RwCreateCamera(scrwidth, scrheight,
RwGetCameraImage(cam));
```

Sharing image buffers can reduce resource (particularly memory) consumption considerably when multiple cameras are employed. However , it does incur additional application housekeeping. Specifically, each time the shared image buffer is to be used for a different camera the entire viewport must be invalidated to prevent data from the previous camera persisting. Furthermore, it is essential that the camera which created the image buffer, i.e., the camera that was created by a call to **RwCreateCamera()** with NULL passed as param, is destroyed after all other cameras sharing the image buffer are destroyed.

It is strongly recommended, therefore, that except in exceptional circumstances, each camera should create its own image buffer.

```
RwInt32
RwDeviceControl(RwDeviceAction action, RwInt32 param1,
      void *param2, RwInt32 size);
```

Arguments

`action` Under MS DOS, RenderWare provides the following device control actions.

- `rwSCRGETCOLOR`

  This device control is used for resolution independent selection of colors from the colors available. On being given a color specified as an array of three numbers denoting the red green and blue components of the color, this control returns the device dependent color value, which when copied directly to the screen will display the correct color. `param1` is ignored (pass `0L`). `param2` should be a pointer of three `RwReal`s specifying the color. A red component value of `CREAL(1.0)` means the maximum amount of red. A red component of `CREAL(0.0)` means no red. `size` should be the size of the data pointed to by `param2` (i.e., `sizeof(RwReal) * 3`).

  For example, the following;

  ```
  RwReal red[]={CREAL(1.0), CREAL(0.0), CREAL(0.0)};
  RwInt32 colorred;

  colorred = RwDeviceControl(rwSCRGETCOLOR, 0L, red,
       sizeof(RwReal) * 3);
  ```

  sets `colorred` to be the device dependent color value for red whatever resolution the video adapter is in.

- `rwPRINTCHAR`

  This device control provides a simple way of displaying a character in any resolution. `param1` is ignored (passed as `0L`) and `param2` should point to a structure `RwPrintChar` (as described below) and `size` should be the size of an `RwPrintChar` structure (i.e., `sizeof(RwPrintChar)`).

  ```
  typedef struct
  {
       RwInt32 x;
       RwInt32 y;
       char c;
       RwInt32 color;
  } RwPrintChar;
  ```

  `x` is the x coordinate of the character in pixels (0 is the left of the screen.). `y` is the y coordinate of the character in pixels (0 is the top of the screen.). `c` is that character you wish to display. `color` is the device dependent color you wish the character to be displayed in. Thus in an 8-bit mode it is the index of the color you require. In a 16 bit mode, the color is encoded into the low 16 bits of color. The simplest way of finding a color via (R, G, B) (red, green, blue) values is via the `rwSCRGETCOLOR` device control.

  The (x, y) coordinate specifies where the top left corner of the characters image will appear.

  For example

```
RwPrintChar print;
RwReal white[]=
        {CREAL(1.0),CREAL(1.0),CREAL(1.0)};
print.x = 10;
print.y = 30;
print.color =
    RwDeviceControl(rwSCRGETCOLOR, 0L, white,
    sizeof(RwReal) * 3);
print.c = 'A';

RwDeviceControl(rwPRINTCHAR, 0L, &print, sizeof(print));
```

Will print the character 'A' at (10, 30) in white. Note that all characters in the ... 8 by 8

This control returns TRUE if successful and FALSE otherwise.

- rwPOINTERREMOVE

  This device control removes the mouse pointer from the display. param1 and param2 and size are all ignored (pass 0L, NULL and 0L respectively).

  For example;

  ```
  RwDeviceControl(rwPOINTERREMOVE, 0L, NULL, 0L);
  ```

  Will remove the mouse pointer from the display.

  This control returns TRUE if successful and FALSE otherwise.

- rwPOINTERDISPLAY

  This device control displays the mouse pointer. param1 is ignored (pass as 0L), param2 should point to an RwMousePointer structure (as described below) and size should be the size of an RwMousePointer structure (i.e., sizeof(RwMousePointer)).

  ```
  typedef struct
  {
      RwInt32 x;
      RwInt32 y;
      RwInt32 buttons;
  } RwMousePointer;
  ```

  This will display the mouse pointer at the current mouse position (removing the mouse pointer at its previous position if necessary). The structure is filled with the position and the current button status of the mouse. For the mouse button, bit 1 will be set if the left button is pressed and bit 3 will be set if the right button is pressed (for three button mice, bit 5 will be set if the middle button is pressed). The x and y co-ordinates are the screen coordinates of the hotspot of the mouse pointer.

  For example:

  ```
  RwMousePointer mouse;

  RwDeviceControl(rwPOINTERDISPLAY, 0L, &mouse,
      sizeof(mouse));
  ```

  Will display the mouse at its current coordinates and fill the mouse structure with the current mouse status.

  This control returns TRUE if successful and FALSE otherwise.

- rwPOINTERSETREGION

  This device control sets the rectangle in which the mouse can be used. It also has the ability to control the mouse movement speed. Initially when the library is opened the mouse is allowed to move over the entire screen area. This device control can restrict this area. param1 is ignored (pass 0L), for normal mouse movement. Setting param1 to a positive value will mean that for a pixel movement the mouse requires 2^param1 mouse events (or mickeys), i.e., if param1 is set to 3 then 8 mickeys are required for each pixel movement. If param1 is set to a negative number then the speed of the pointer is accelerated by 2^-param1, i.e., if param1 is set to -4 for each mickey the mouse pointer will move 16 pixels. param2 should point to an RwRect structure and size should be the size of an RwRect structure (i.e., sizeof(RwRect)). This defines the area the hot spot of the mouse can traverse.

For example;

```
RwRect area;

area.x = 20;
area.y = 50;
area.w = 100;
area.h = 10;
RwDeviceControl(rwPOINTERSETREGION, -1L, &area,
     sizeof(area));
```

only allows the mouse pointer hotspot to occupy the region 20 to 120 for the x co-ordinate and 50 to 60 for the y co-ordinate. The mouse will now move 2 pixels

This control returns TRUE if successful and FALSE otherwise.

- rwPOINTERSETCLIPREGION

  This device control sets he clipping region for the mouse sprite image. param1 is ignored (pass 0L). param2 should point to an RwRect structure and size should be the size of an RwRect structure (i.e., sizeof(RwRect)). This defines the area the mouse sprite image should be clipped to.

  For example;

```
RwRect clip;

clip.x = 40;
clip.y = 80;
clip.w = 40;
clip.h = 90;
RwDeviceControl(rwPOINTERSETCLIPREGION, 0L, &clip,
     sizeof(clip));
```

sets the clipping region to be between 40 and 80 for the x co-ordinate and 80 to 170 for the y co-ordinate

This control returns TRUE if successful and FALSE otherwise.

- rwPOINTERSETIMAGE

  This device control changes the mouse pointer image. param1 is ignored (pass 0L), param2 should point to an RwPointerImage structure (as described below) and size should be the size of an RwPointerImage structure (i.e., sizeof(RwPointerImage)) .

```
typedef struct PointerImage
{
     RwInt32 hotx;
     RwInt32 hoty;
     RwInt32 w;
     RwInt32 h;
     void *image;
} RwPointerImage;
```

  hotx and hoty define the hot spot on the image. What this means is that this point becomes origin of the mouse image. w and h are the images width and height respectively. image points to an raw pixmap image (device dependent) which can be created by the rwBITMAPTORAW and rwCHARMAPTORAW device controls. If image is set to NULL then the pointer image is set to the default image pointer with its outline color stored in w and its remaining color stored in h.

For example to change an image to a 5 by 5 image with its origin at the center;

```
RwPointerImage pimage;

pimage.hotx = 2;
pimage.hoty = 2;
pimage.w = 5;
pimage.h = 5;
pimage.image = rawpixmap;
RwDeviceControl(rwPOINTERIMAGE, 0L, &pimage,
     sizeof(pimage));
```

This control returns `TRUE` if successful and `FALSE` otherwise.

- `rwBITMAPTORAW`

The device control converts a bitmap into a form that can be used for as a pointer sprite. `param1` is ignored (pass `0L`). `param2` should be set to point to an `RwImageConvert` structure (as described below) and `size` should be the size of an `RwImageConvert` structure (i.e., `sizeof(RwImageConvert)`).

```
typedef struct
{
     void *inimage;
     RwInt32 w;
     RwInt32 h;
     RwInt32 colora;
     RwInt32 colorb;
     void *outstorage;
} RwImageConvert;
```

`inimage` should be set to point at the pixmap you wish to convert. Note that the `inimage` pixmap is rounded horizontally to 8 pixels. That is each raster line in the source is always made up of an integer number of bytes. This means if an image is 15 pixels wide and one pixel high the bitmap will take 2 whole bytes. `w` and `h` are set to the images width and height respectively. `colora` is the color bits set to 1 are set to. Bits set to 0 will be transparent. `colorb` should be set to 0. `outstorage` should be set to `NULL` if you wish RenderWare to allocate memory for the transformed image. Otherwise it should be set to the area where the image should be written. On exit `outstorage` will always point to where the translated image was stored.

For example;

```
charimage[] =
 {0xff, 0x81, 0x81, 0x81, 0x81, 0x81, 0x81, 0xff};
RwReal white[] =
        {CREAL(1.0), CREAL(1.0),CREAL(1.0)};
RwImageConvert cimage;
void *rawimage;
cimage.inimage = image;
cimage.w = 8;
cimage.h = 8;
cimage.colora =
    RwDeviceControl(rwSCRGETCOLOR, 0L, white,
    sizeof(RwReal) * 3);
cimage.outstorage = NULL;
RwDeviceControl(rwBITMAPTORAW, 0L, &cimage,
    sizeof(cimage));
rawimage = cimage.outstorage;
```

Creates a white square image

This control returns `TRUE` if successful and `FALSE` otherwise.

- `rwCHARMAPTORAW`

This device control works in a similar way to the `rwBITMAPTORAW` device control explained above. Here the image consists of characters where spaces denote 'transparent' areas, 'a' denotes a pixel set to color a and 'b' denotes a pixel set to color b.

For example, to create an outlined square raw image;

```
char image[]="\
    aaaaaa \
    abbbbbba\
    abaaaaba\
    aba  aba\
    aba  aba\
    abaaaaba\
    abbbbbba\
    aaaaaa ";
RwReal white[]=
        {CREAL(1.0), CREAL(1.0),CREAL(1.0)};
RwReal black[]=
        {CREAL(0.0), CREAL(0.0), CREAL(0.0)};
RwImageConvert cimage;
void *rawimage;

cimage.inimage = image;
cimage.w = 8;
cimage.h = 8;
cimage.colora =
    RwDeviceControl(rwSCRGETCOLOR, 0L, black,
    sizeof(RwReal) * 3);
cimage.colorb =
    RwDeviceControl(rwSCRGETCOLOR, 0L, white,
    sizeof(RwReal) * 3);
cimage.outstorage = NULL;

RwDeviceControl(rwBITMAPTORAW, 0L, &cimage,
    sizeof(cimage));

rawimage = cimage.outstorage;
```

Creates a white square outlined in black.

This control returns `TRUE` if successful and `FALSE` otherwise.

- rwPOINTERDISPLAYAT

  This action displays the pointer image at a specified position on the screen. param1 is ignored, param2 should be a pointer to an RwMousePointer structure and size should be the size of an RwMousePointer structure.

  The x field of the RwMousePointer structure is the screen X coordinate that the pointer image is to be placed at. The y field is the screen Y coordinate that the pointer image is to be placed at.

  If the image is currently displayed at another position on the display it is removed before the image is displayed at the new position. The pointer clipping rectangle applies to the newly displayed image. The mouse bounding region will not have any affect on the positioning. The mouse pointer will remain at the new position until,

  1) rwPOINTERDISPLAY is used - then the pointer image will move to the current mouse position.

  2) rwPOINTERDISPLAYAT is used - the pointer image at the previous position is removed and the pointer image is displayed at the new position.

3) rwPOINTERREMOVE is used - the pointer is removed from the display.

This control returns TRUE if successful and FALSE otherwise.

- rwSETPOINTERPOSITION

  This action sets the internal mouse absolute position (as read by rwPOINTERGETPOSITION or rwPOINTERDISPLAY). param1 is ignored, param2 should be a pointer to an RwMousePointer structure and size should be the size of an RwMousePointer structure.

  The x field of the RwMousePointer structure is the screen X coordinate that the mouse is to be placed at. The y field is the screen Y coordinate that the mouse is to be placed at.

The mouse bounding region is not taken into account when the position is set. Note [illegible] rwPOINTERDISPLAY call.

This control returns TRUE if successful and FALSE otherwise.

- rwGETPOINTERPOSITION

  This action returns the current absolute position of the mouse along with the mouse button status. param1 is ignored, param2 should be a pointer to an RwMousePointer structure and size should be the size of an RwMousePointer structure. This structure will be filled in with the current absolute position of the mouse as well as the current mouse button status. This will have no effect on any currently displayed pointer image.

  This control returns TRUE if successful and FALSE otherwise.

- rwGETPOINTERRELATIVE

  This action returns the current relative position of the mouse along with the mouse button status. param1 is ignored, param2 should be a pointer to an RwMousePointer structure and size should be the size of an RwMousePointer structure.

  The structure is filled with the number of 'mickeys' that have occurred between the previous call to this function (or the start of the application) and the current call. The x field in the structure holds the number of mickeys moved, since the last call, in the X direction. The y field holds the number of mickeys moved, since the last call, in the Y direction. The button entry will hold the current mouse button status.

  A 'mickey' is the highest resolution value of change given by the mouse.

It is a relative value - positive values mean right or down, negative values mean left or up. The larger the mickey value the greater the distance the mouse has moved.

Note: The mickey values can used to produce the 'absolute' position of the mouse if, after the first call, the corresponding relative mickey values are added to a running total of mickeys in the X and Y directions. The running totals are then the 'absolute' position of the mouse.

This control returns `TRUE` if successful and `FALSE` otherwise.

`RwCamera *`
`RwDuplicateCamera``(RwCamera *cam, void *param)`

Arguments

`param` `NULL` or the image buffer of an existing camera (as returned by calling **`RwGetCameraImage()`**).

The duplicated camera will not share the image buffer of the camera being duplicated unless the image buffer of the existing camera is passed as param. If `NULL` is passed as param the new camera will create its own image buffer.

For an example of how to specify param, see the discussion of **`RwCreateCamera()`** in this section.

```
void *
RwGetCameraImage(RwCamera *cam);
```

Return value

The image buffer of a RenderWare camera under MS DOS is always an RwRaster object. It can therefore be queried via the RwRaster access functions.

RwBool
RwGetDeviceInfo(RwDeviceInfo info, void *value, RwInt32 size)

Comments

The MS DOS specific aspects of each device information type are as follows;

rwSCRDEPTH	The depth (in bits) of the resolution that the VGA adapter is in.
rwSCRWIDTH	The width (in pixels) of the resolution the VGA adapter is in.
rwSCRHEIGHT	The height (in pixels) of the resolution the VGA adapter is in.
rwINDEXEDRENDERING	As the generic **RwGetDeviceInfo**().
rwPALETTEBASED	Rendering is palette based if and only if the video adapter is running in a palette based mode. This will normally be the case if the video adapter is in 8-bit (256) color mode. In other modes rendering will not be palette based.

The following options apply when the output device is palette based:

rwPALETTE	Returns a pointer to a table of 256*3 characters. Each set of consecutive three characters constitutes a color in the form (red, green, blue). That is the first character is the red intensity of the color, the second green and the third blue. The first triple is for color zero. The next for color one etc.
rwPALETTESIZE	As the generic **RwGetDeviceInfo**().
rwLASTPALETTEENTRY	As the generic **RwGetDeviceInfo**().
rwFIRSTPALETTEENTRY	As the generic **RwGetDeviceInfo**().

RwBool

RwOpen(char *devname, void *param);

Arguments

devname — The device name is a null-terminated string. Names currently supported under MS DOS are :

```
"DOS",
"DOSMOUSE"
"NullDevice"
```

The "NullDevice" driver allows the library to be used when output to a display is not required (for instance, when reading from or to files).

"DOSMOUSE" will open the library with the mouse driver active. This will mean that the library can only be accessed if a Microsoft compatible mouse driver is not required.

"DOS" performs the same function as "DOSMOUSE" except the mouse is not accessed, and so a mouse driver is not required.

param — param should be set to a pointer to a RwInt32. If the library does not open, then the RwInt32 pointed to will be set to one of the following symbols;

E_RW_DOS_MODE_UNAVAILABLE

Unable to access the video mode requested.

E_RW_DOS_NO_VESA_BIOS

No VESA BIOS is available, install a VESA TSR for your video card.

E_RW_INCOMPATIBLE_BIOS

The VESA BIOS is not a recent enough release to be usable. (must be 1.0 or greater).

E_RW_NO_MOUSE

No Microsoft mouse driver found.

These symbols are located in rwdos.h header file.

Comments

RwOpen() parses the RenderWare initialization file dosrw.ini as described in "RenderWare Library Configuration" on page -

```
RwBool
RwOpenExt(char *devname, void *param, RwInt32 numargs,
    RwOpenArgument *args);
```

Arguments

`devname`	The device name as a null-terminated string. Names currently supported under MS DOS are:
	`"DOS"`
	`"DOSMOUSE"`
	`"NullDevice"`
	
	The "`NullDevice`" driver allows the library to be used when output to a display is not required (for instance, when reading from or writing to files).
	
	"`DOSMOUSE`" will open the library with the mouse driver active. This will mean that the library can only be accessed if a Microsoft compatible mouse driver is not required.
	
	"`DOS`" performs the same function as "`DOSMOUSE`" except the mouse is not accessed, and so a mouse driver is not required.

`param`	`param` should be set to a pointer to an `RwInt32`. The long will be set to an error code if the library cannot be opened. See **`RwOpen()`** for details.
`numargs`	The number of optional arguments specified.
`args`	An array of optional open arguments.

Comments

As discussed in "RenderWare Library Configuration" on page -, **`RwOpenExt()`** does not read the RenderWare initialization file `dosrw.ini`. Library configuration control is achieved by specifying a number of additional arguments to **`RwOpenExt()`**.

`RwOpenExt()` takes a pointer to an array of `RwOpenArgument` structures. `RwOpenArgument` is defined as follows:

```
typedef struct
{
    RwOpenOption option;
    void *value;
} RwOpenArgument;
```

`option` is one of the identifiers defined below and value is a parameter specific to the option type. If the parameter type is integer, the integer value should be cast to a `void *`.

For example:

```
args[0].value = (void *)10;
```

The configuration options supported under MS DOS are:

`rwGAMMACORRECT`	Controls whether RenderWare performs gamma correction on its color palette. value should be non-zero to enable gamma correction, and zero to disable gamma correction.
`rwSCRWIDTH`	Requests a video mode of the graphics adapter with a width of value.
`rwSCRHEIGHT`	Requests a video mode of the graphics adapter with a

height of value.

`rwSCRDEPTH`	Requests a video mode of the graphics adapter with a depth of value.

The following example demonstrates opening the RenderWare library with a resolution of 640 by 480 in 8-bit color.

```
RwOpenArgument args[3];
LONG nERROR;

args[0].option = rwSCRWIDTH;
args[0].value = (void *)640;
args[1].option = rwSCRHEIGHT;
args[1].value = (void *)480;
args[2].option = rwSCRDEPTH;
args[2].value = (void *)8;

RwOpenExt("DOSMOUSE", &nERROR, 3, args);
```

`RwBool`
`RwOpenDebugStream``(char *filename)`

Arguments

`filename` Specifying "`MONO`:" as the name of a debugging stream will result in debugging output being issued to a monochrome display adapter (assuming a configuration with both a VGA card and monochrome display adapter).

Note that when linking against the debugging version of the library, the default debugging stream is `\rw.log`.

```
RwCamera *
RwShowCameraImage(RwCamera *cam, void *param);
```

Arguments

`param` `param` is ignored. It should be passed as `NULL`.

For example;

```
RwShowCameraImage(cam, NULL);
```

Other Platforms

If you are using an SDK for a platform other than MS Windows or MS DOS please see the Release Notes supplied with that SDK.

Error Codes

Related Topics

Error Descriptions
Error Identifiers and Codes

Error Descriptions

Note that in the discussion of the following error codes, the comments involving Object Builder functions **RwModelBegin**(), **RwModelEnd**(), **RwProtoBegin**(), **RwProtoEnd**(), **RwClumpBegin**(), **RwClumpEnd**(), **RwTransformBegin**() and **RwTransformEnd**() also apply to their script keyword counterparts.

E_RW_BADOPEN
An error occurred while opening the specified file.

E_RW_COMPLEXPOLYGON
The specified polygon has too many sides. The maximum number of sides a polygon can have in RenderWare V1.4 is 255.

E_RW_DEFSCENE
An attempt was made to destroy the default scene or explicitly remove a clump or light from it.

E_RW_DEGEN
An attempt was made to create a degenerate clump (a clump with no children and no geometry).

E_RW_DEGENPOLYGON
A degenerate polygon (one with less than three sides) was specified.

E_RW_INTERNAL
An internal (library) error has occurred. Contact RenderWare technical support.

E_RW_INVAXISALIGNMENT
An invalid clump axis alignment type was specified. The legal values are `rwNOAXISALIGNMENT`, `rwALIGNAXISZORIENTX`, `rwALIGNAXISZORIENTY` and `rwALIGNAXISXYZ`.

E_RW_INVCAMERAPROJECTION
An invalid camera projection type was specified. The legal values are `rwPERSPECTIVE` and `rwPARALLEL`.

E_RW_INVCOP
An invalid **RwCombineOperation** was specified. The legal values are `rwREPLACE`, `rwPRECONCAT` and `rwPOSTCONCAT`.

E_RW_INVDEVICE
An invalid device name was specified in a call to **RwOpen**() or **RwOpenExt**().

E_RW_INVDEVICEACTION
An invalid action was specified in a call to **RwDeviceControl**(). For a description of legal actions see Appendix B.

E_RW_INVDEVICEINFO
An invalid information type was specified in a call to **RwGetDeviceInfo**(). For a description of legal information types see the description of **RwGetDeviceInfo**() and Appendix B.

E_RW_INVFRAME
An invalid texture frame number was specified. A valid frame index is greater than or equal to zero and less than the number of frames in the texture.

E_RW_INVFRAMESTEP
An invalid texture frame step was specified. A valid step size is less than the absolute number of frames in the texture.

E_RW_INVGEOMETRYSAMPLING
An invalid geometry sampling type was specified. The legal values are `rwPOINTCLOUD`, `rwWIREFRAME` and `rwSOLID`.

E_RW_INVHINT
An invalid clump hint was specified. The legal values are `rwCONTAINER`, `rwHS` and `rwEDITABLE`.

E_RW_INVIMAGEFILE
An image file (MS Windows bitmap file or Sun Rasterfile) being read by **RwReadRaster**(), **RwReadMaskRaster**(), **RwReadTexture**() or **RwReadNamedTexture**() was not valid. This code indicates an error in the image data in the file.

E_RW_INVLIGHT
An invalid light type was specified. The legal values are `rwDIRECTIONAL`, `rwPOINT` and `rwCONICAL`.

E_RW_INVLIGHTSAMPLING
An invalid light sampling type was specified. The legal values are `rwFACET and rwVERTEX`.

E_RW_INVMATERIAL
An attempt was made to destroy a material whose handle was not obtained by calling **RwCreateMaterial**(), e.g., a material obtained by calling **RwGetPolygonMaterial**()

E_RW_INVOPENOPTION
An invalid option was specified in a call to **RwOpenExt**(). For a description of legal options see Appendix B.

E_RW_INVPROTOTYPE
A prototype attempted to create an instance of itself.

E_RW_INVRASTEROPTIONS
An invalid raster processing option (or combination of options) was specified. The valid options are `rwGAMMARASTER`, `rwDITHERRASTER` and `rwFITRASTER`.

E_RW_RASTERSIZE
A raster of an invalid size was specified. If the raster was being selected into a texture with **RwSetTextureRaster**() the raster must have a width of 128 and a height of `n` * 128 where `n` is the number of frames in a multi-frame texture. If the raster is receiving a copy of a cameras viewport via **RwGetCameraViewportRaster**() the raster must be the same size as the cameras viewport.

E_RW_INVBUFFERSIZE
A buffer passed in to a RenderWare function was not large enough

E_RW_INVSEARCHMODE
An invalid search mode was specified. The legal values are `rwLOCAL` and `rwGLOBAL`.

E_RW_INVSPP
An invalid spline path type was specified. The legal values are `rwSMOOTH` and `rwNICEENDS`.

E_RW_INVSPT
An invalid spline type was specified. The legal values are `rwOPENLOOP` and `rwCLOSEDLOOP`.

E_RW_INVSTATE
An invalid state was specified. The legal values are `rwON` and `rwOFF`.

E_RW_INVSYSTEMINFO
An invalid system parameter was specified. The legal values are `rwVERSIONSTRING`, `rwVERSIONMAJOR`, `rwVERSIONMINOR`, `rwVERSIONRELEASE`, `rwFIXEDPOINTLIB` and `rwDEBUGGINGLIB`.

E_RW_INVTEXTUREDITHERMODE
An invalid texture dithering mode was specified. The legal values are `rwAUTODITHER`, `rwDITHERON` and `rwDITHEROFF`.

E_RW_INVTEXTUREHEIGHT
A texture with an invalid height was specified.

E_RW_INVTEXTUREMODE
An invalid texture mode was specified. The legal values are `rwLIT`, `rwFORESHORTEN` and `rwFILTER`.

E_RW_INVTEXTURENAME
An invalid texture name was specified.

E_RW_INVTEXTUREWIDTH
A texture with an invalid width was specified.

E_RW_INVUSERDRAWALIGN
An invalid user-draw alignment type was specified. The legal values are `rwALIGNTOP`, `rwALIGNBOTTOM`, `rwALIGNLEFT` and `rwALIGNRIGHT`. For convenience `rwALIGNTOPLEFT` and `rwALIGNBOTTOMRIGHT` are also defined.

E_RW_INVUSERDRAWTYPE
An invalid user-draw type was specified. The legal values are `rwCLUMPALIGN`, `rwVERTEXALIGN`, `rwBBOXALIGN` and `rwVPALIGN`.

E_RW_INVVERTEXINDEX
An invalid vertex index was found. A valid index for a vertex is greater than or equal to one and less than or equal to the number of vertices in the clump to which it belongs.

E_RW_NESTEDMODEL
A nested **RwModelBegin**() was found in an **RwModelBegin**() … **RwModelEnd**() block or an **RwClumpBegin**() … **RwClumpEnd**() or **RwProtoBegin**() … **RwProtoEnd**() block. Nested modeling contexts are not allowed.

E_RW_NESTEDPROTOTYPE
A nested **RwProtoBegin**() was found in an **RwProtoBegin**() … **RwProtoEnd**() block or an **RwClumpBegin**() … **RwClumpEnd**() block. Nested prototype declarations are not allowed.

E_RW_NOCLUMP
No clump is currently under construction. Note that the current clump is an implicit argument of some Object Builder functions, e.g., **RwSphere**().

E_RW_NOCLUMPBUILT
The parsing of a script file resulted in no clump being created, i.e., the top-level **ClumpBegin** … **ClumpEnd** was missing from the script.

E_RW_NOERROR
No error has been set. This code indicates that no error has been detected.

E_RW_NOFILE
The specified file does not exist.

E_RW_NOMATCHBEGIN
No matching Begin was found for an End. For example, this error would occur if there was no matching **RwTransformBegin**() for an **RwTransformEnd**().

E_RW_NOMATCHEND
No matching End was found for a Begin. For example, this error would occur if there was no matching **RwModelEnd**() for an **RwModelBegin**().

E_RW_NOMEM
The library was unable to perform the specified operation due to insufficient memory.

E_RW_NOMODELBEGIN
An attempt was made to declare a prototype outside of an **RwModelBegin**() … **RwModelEnd**() block.

E_RW_NOPROTOTYPEFOUND
No prototype with the specified name was found.

E_RW_NOTROOT
An attempt was made to add to a scene or remove from a scene a clump that is not the root of its hierarchy.

E_RW_NULLP
A NULL pointer was used as an argument to a library function where a non-NULL object pointer was expected.

E_RW_RANGE
A numeric range error occurred.

E_RW_RASTERINUSE
An attempt was made to select a raster already owned by a texture into a different texture or to destroy a raster owned by a texture.

E_RW_READ
An error occurred while reading from an input stream.

E_RW_RSINVAXISALIGNMENT
An invalid clump axis alignment type was specified in a script file. The legal values are **None**, **ZOrientX**, **ZOrientY** and **XYZ**.

E_RW_RSINVDITHERMODE
An invalid texture dithering mode was specified in a script file. The legal values are **On**, **Off** and **Auto**.

E_RW_RSINVGAMMAMODE
An invalid texture gamma correction mode was specified in a script file. The legal values are **`On`** and **`Off`**.

E_RW_RSINVGEOMETRYSAMPLING
An invalid geometry sampling type was specified in a script file. The legal values are **`PointCloud`**, **`Solid`** and **`WireFrame`**.

E_RW_RSINVHINT
An invalid clump hint was specified in a script file. The legal values are **`NULL`**, **`Container`**, **`HS`** and **`Editable`**.

E_RW_RSINVLIGHTSAMPLING
An invalid light sampling type was specified in a script file. The legal values are **`Facet`** and **`Vertex`**.

E_RW_RSINVTEXTUREMODE
An invalid texture mode was specified in a script file. The legal values are **`NULL`**, **`Lit`**, **`Foreshorten`** and **`Filter`**.

E_RW_RSINVTRACESTATE
A invalid tracing state was specified in a script file. The legal values are **`On`** and **`Off`**.

E_RW_RSNOHINTS
No hints were specified as arguments for a scripting command that adds, removes, or sets hints. The legal values are **`NULL`**, **`Container`**, **`HS`** and **`Editable`**.

E_RW_RSNOTEXTUREMODES
No texture modes were specified as arguments for a scripting command that adds, removes or sets texture modes. The legal values are **`Null`**, **`Lit`**, **`Foreshorten`** and **`Filter`**.

E_RW_RSPARSE
An invalid keyword was found in a script file.

E_RW_RSREAD
An I/O error occurred while reading from a script file.

E_RW_SHPPATH
The shape path is too long (greater than 1024 characters).

E_RW_TEXTURENOTFOUND
The specified texture was not found.

E_RW_USER
A call to **`RwSetUserError`**`()` generated this error.

E_RW_WRITE
An error occurred while writing to an output stream.

E_RW_WSWRITE
An error occurred while writing a script to an output stream.

E_RW_ZEROVEC
A zero length vector was specified.

Error Identifiers and Codes

Errors Sorted Alphabetically By Identifier

Identifier	Numeric Code
E_RW_BADOPEN	14
E_RW_COMPLEXPOLYGON	41
E_RW_DEFSCENE	25
E_RW_DEGEN	7
E_RW_DEGENPOLYGON	40
E_RW_INTERNAL	71
E_RW_INVAXISALIGNMENT	49
E_RW_INVBUFFERSIZE	70
E_RW_INVCAMERAPROJECTION	44
E_RW_INVCOP	2
E_RW_INVDEVICE	18
E_RW_INVDEVICEACTION	64
E_RW_INVDEVICEINFO	63
E_RW_INVFRAME	20
E_RW_INVFRAMESTEP	21
E_RW_INVGEOMETRYSAMPLING	26
E_RW_INVHINT	47
E_RW_INVIMAGEFILE	69
E_RW_INVLIGHT	8
E_RW_INVLIGHTSAMPLING	28
E_RW_INVMATERIAL	19
E_RW_INVOPENOPTION	65
E_RW_INVPROTOTYPE	37
E_RW_INVRASTEROPTIONS	60
E_RW_INVRASTERSIZE	62
E_RW_INVSEARCHMODE	46
E_RW_INVSPP	16
E_RW_INVSPT	17
E_RW_INVSTATE	45
E_RW_INVSYSTEMINFO	55
E_RW_INVTEXTUREDITHERMODE	61
E_RW_INVTEXTUREHEIGHT	23
E_RW_INVTEXTUREMODE	56
E_RW_INVTEXTURENAME	43
E_RW_INVTEXTUREWIDTH	22
E_RW_INVUSERDRAWALIGN	51
E_RW_INVUSERDRAWTYPE	50
E_RW_INVVERTEXINDEX	24
E_RW_NESTEDMODEL	32
E_RW_NESTEDPROTOTYPE	35
E_RW_NOCLUMP	38
E_RW_NOCLUMPBUILT	39
E_RW_NOERROR	0
E_RW_NOFILE	13
E_RW_NOMATCHBEGIN	33
E_RW_NOMATCHEND	34
E_RW_NOMEM	3
E_RW_NOMODELBEGIN	36
E_RW_NOPROTOTYPEFOUND	30
E_RW_NOTROOT	15
E_RW_NULLP	1
E_RW_RANGE	11

Errors Sorted Numerically By Code

Numeric Code	Identifier
0	E_RW_NOERROR
1	E_RW_NULLP
2	E_RW_INVCOP
3	E_RW_NOMEM
4	E_RW_RSPARSE
5	E_RW_RSREAD
6	E_RW_WSWRITE
7	E_RW_DEGEN
8	E_RW_INVLIGHT
9	E_RW_SHPPATH
10	E_RW_READ
11	E_RW_RANGE
12	E_RW_WRITE
13	E_RW_NOFILE
14	E_RW_BADOPEN
15	E_RW_NOTROOT
16	E_RW_INVSPP
17	E_RW_INVSPT
18	E_RW_INVDEVICE
19	E_RW_INVMATERIAL
20	E_RW_INVFRAME
21	E_RW_INVFRAMESTEP
22	E_RW_INVTEXTUREWIDTH
23	E_RW_INVTEXTUREHEIGHT
24	E_RW_INVVERTEXINDEX
25	E_RW_DEFSCENE
26	E_RW_INVGEOMETRYSAMPLING
27	E_RW_RSINVEGEOMETRYSAMPLING
28	E_RW_INVLIGHTSAMPLING
29	E_RW_RSINVLIGHTSAMPLING
30	E_RW_NOPROTOTYPEFOUND
31	E_RW_ZEROVEC
32	E_RW_NESTEDMODEL
33	E_RW_NOMATCHBEGIN
34	E_RW_NOMATCHEND
35	E_RW_NESTEDPROTOTYPE
36	E_RW_NOMODELBEGIN
37	E_RW_INVPROTOTYPE
38	E_RW_NOCLUMP
39	E_RW_NOCLUMPBUILT
40	E_RW_DEGENPOLYGON
41	E_RW_COMPLEXPOLYGON
42	E_RW_TEXTURENOTFOUND
43	E_RW_INVTEXTURENAME
44	E_RW_INVCAMERAPROJECTION
45	E_RW_INVSTATE
46	E_RW_INVSEARCHMODE
47	E_RW_INVHINT
48	E_RW_USER
49	E_RW_INVAXISALIGNMENT
50	E_RW_INVUSERDRAWTYPE
51	E_RW_INVUSERDRAWALIGN

52	E_RW_RSINVTRACESTATE
53	E_RW_RSINVHINT
54	E_RW_RSINVAXISALIGNMENT
55	E_RW_INVSYSTEMINFO
56	E_RW_INVTEXTUREMODE
57	E_RW_RSINVTEXTUREMODE
58	E_RW_RSNOHINTS
59	E_RW_RSNOTEXTUREMODES
60	E_RW_INVRASTEROPTIONS
61	E_RW_INVTEXTUREDITHERMODE
62	E_RW_INVRASTERSIZE
63	E_RW_INVDEVICEINFO
64	E_RW_INVDEVICEACTION
65	E_RW_INVOPENOPTION
66	E_RW_RASTERINUSE
67	E_RW_RSINVDITHERMODE
68	E_RW_RSINVGAMMAMODE
69	E_RW_INVIMAGEFILE
70	E_RW_INVBUFFERSIZE
71	E_RW_INTERNAL

The Texture File Formats

Textures in RenderWare V1.4 are 128 x 128, 8 or 16 bit deep bitmaps. However, RenderWare can read bitmaps of sizes other than 128 x 128 pixels and depths other than 8 or 16 bits. Such bitmaps are converted automatically to RenderWares internal raster format. Furthermore, RenderWare can read textures from MS Windows and OS/2 bitmap (`.bmp`) files in addition to RenderWares own texture file format (`.ras`).

In the case of RenderWares own texture file format (`.ras`) the following are supported:

- Bitmaps with depths of 1, 8, 24 or 32 bits.
- Uncompressed or run length encoded bitmaps.

For MS Windows and OS/2 bitmap (`.bmp`) files the following are supported:

- Bitmaps with depths of 1, 4, 8 or 24 bits.
- Uncompressed or run length encoded bitmaps.

Bitmaps of any width and height ... than 8 bits will be filtered ... banding

- ... pleasing ... larger, deeper textures which are filtered and quantized to size. Visually
- Texture movies (multi-frame textures) must be stored as bitmaps which are 128 pixels wide and a whole multiple of 128 pixels high.

RenderWares own texture file format is based on the format used for Sun Microsystemss rasterfiles. RenderWare will read any legal Sun Rasterfile. For a description of Suns Rasterfile format see:

[1] Rasterfile (5) in Sun Microsystems, **Sun OS 4.0 Programmers Manual**, 1990

[2] Pat McGee, Format for byte encoded rasterfiles in **Sun-Spots Digest**, Volume 6, Issue 84.

[3] David. C. Kay and John R. Levine, **Graphics File Formats**, Windcrest/McGraw-Hill, 1992.

For a description of MS Windows bitmap (.bmp) file format see:

[4] Bitmap-File Formats, **Microsoft Windows 3.1 Programmers Reference Volume 4**: Resources, Microsoft, 1987-1992. Part Number: 30211.

[5] David. C. Kay and John R. Levine, **Graphics File Formats**, Windcrest/McGraw-Hill, 1992.

Textures are, in fact, based on RenderWares own bitmap object type, RwRaster. The depth of RwRaster objects is always equal to the current RenderWare rendering depth. Therefore, textures will be either 8 or 16 bits deep, the actual depth being decided at run-time.

Currently RenderWare does not support any 16 bit texture file formats. Therefore, when rendering at 16 bits all texture files undergo color conversion.

Library Defaults

This appendix details the various default values found in the RenderWare library. This includes the default values of global library parameters and the default values of RenderWare objects when first created.

Related Topics

Camera Object Defaults
Clump Object Defaults
Debugging Defaults
Device Information Defaults
Error Status Defaults
Library Global Defaults
Light Object Defaults
Material Object Defaults
Matrix Object Defaults
Polygon Object Defaults
Raster Object Defaults
Scene Object Defaults
Spline Object Defaults
System Information Defaults
Texture Object Defaults
Texture Dictionary Defaults
UserDraw Object Defaults
Vertex Defaults

Camera Object Defaults

Attribute	Default
Position	[CREAL(0.0), CREAL(0.0), CREAL(0.0)] in world coordinates
Look At	[CREAL(0.0), CREAL(0.0), CREAL(-1.0)]
Look Right	[CREAL(1.0), CREAL(0.0), CREAL(0.0)]
Look Up	[CREAL(0.0), CREAL(1.0), CREAL(0.0)]
Viewport	Position = [0, 0] Size = [0, 0]
Viewwindow	[CREAL(1.0), CREAL(1.0)]
View Offset	[CREAL(0.0), CREAL(0.0)]
Near Clipping	CREAL(0.05)
Projection	rwPERSPECTIVE
Back Color	[CREAL(0.0), CREAL(0.0), CREAL(0.0)] (Black)
Backdrop	NULL
Backdrop Offset	[0, 0]
Backdrop Viewport Rectangle	Position = [0, 0] Size = [0, 0]
Data	NULL
Image	Device dependent

Clump Object Defaults

Attribute	Default
Origin	[CREAL(0.0), CREAL(0.0), CREAL(0.0)] in world coordinates
Matrix	Identity
Joint Matrix	Identity
LTM	Identity
Owner	The default scene (as returned by **RwDefaultScene**())
Parent	NULL
Root	The clump itself
Data	NULL
First Child	NULL
Next Clump	NULL
Number of Vertices	Defined at creation time
Number of Polygons	Defined at creation time
Number of Children	0
Number of UserDraws	0
Tag	0
Hints	rwHS
Axis Alignment	rwNOAXISALIGNMENT
State	rwON
Bounding Box	Computed
Viewport Rectangle	Computed

Debugging Defaults

Attribute	Default
Stream	`stderr` on Unix `\rw.log` on DOS and MS Windows `stderr` on the Macintosh
Assertion State	`rwON`
Message State	`rwON`
Script State	`rwOFF`
Severity	`rwINFORM`
Trace State	`rwOFF`

Device Information Defaults

Attribute	Default
`rwRENDERDEPTH`	8 or 16. Actual value is device specific
`rwINDEXEDRENDERING`	Non-zero if the current render depth is 8 Zero if the current render depth is 16
`rwPALETTEBASED`	Non-zero if the output device is palette based Zero if the output device is not palette based
`rwPALETTE`	Device dependent
`rwPALETTESIZE`	Device dependent
`rwFIRSTPALETTEENTRY`	Device dependent
`rwLASTPALETTEENTRY`	Device dependent

Error Status Defaults

Attribute	Default
Status	`E_RW_NOERROR`
Internal Error	Undefined

Library Global Defaults

Attribute	Default
Current Camera	`NULL`
Shape Path	Value of environment variable `RWSHAPEPATH` or if the environment variable is not set on DOS, MS Windows and Unix on the Macintosh
Scenes	One. The default scene (as returned by **`RwDefaultScene`**`()`)
Lights	None
Clumps	None
Cameras	None
Textures	None
Texture Dictionaries	One on the Texture Dictionary stack.
Rasters	None
Splines	None
UserDraws	None
Matrices	One on the Scratch matrix stack, one on the Current Matrix stack and one on the Joint Matrix stack
Materials	One on the Material stack
Texture Dithering	`rwAUTODITHER`
Texture Gamma Correction	`rwON`

Light Object Defaults

Attribute	Default
Type	Defined at creation time
Brightness	Defined at creation time
Color	Defined at creation time (each of the red, green and blue components of the color are equal to the specified brightness)
Position	Defined at creation time for point and conical lights
Vector	Defined at creation time for directional lights. `[CREAL(0.0), CREAL(-1.0), CREAL(0.0)]` for conical lights
Owner	The default scene (as returned by **`RwDefaultScene`**`()`)
Cone Angle	`CREAL(30.0)` for conical lights `CREAL(180.0)` for point lights
State	`rwON`
Data	`NULL`

Material Object Defaults

Attribute	Default
Ambient	`CREAL(0.0)`
Diffuse	`CREAL(0.0)`
Specular	`CREAL(0.0)`
Light Sampling	`rwFACET`
Geometry Sampling	`rwSOLID`
Color	`[CREAL(0.0), CREAL(0.0), CREAL(0.0)]` (Black)
Opacity	`CREAL(1.0)`
Texture	`NULL`
Texture Modes	`rwLIT`

Matrix Object Defaults

Attribute	Default
Elements	Identity

Polygon Object Defaults

Attribute	Default
Material	Defined at creation time by the Current Material (as returned by **`RwCurrentMaterial`**`()`)
Vertices	Defined at creation time
Center	Computed
Normal	Computed
Number of Sides	Defined at creation time
Owner	Defined at creation time
UV	`[CREAL(0.5), CREAL(0.5)]` for all vertices
Tag	`0`
Data	`NULL`

Raster Object Defaults

Attribute	Default
Width	Defined at creation time (either explicitly or by source image)
Height	Defined at creation time (either explicitly or by source image)
Depth	Equal to the current RenderWare render depth
Stride	Computed from the width
Pixel Pointer	Defined at creation time
Pixel Values	Undefined if created with **`RwCreateRaster`**`()` Derived from the source image if created by **`RwBitmapRaster`**`()`, **`RwReadRaster`**`()` or **`RwReadMaskRaster`**`()`
Data	`NULL`

Scene Object Defaults

Attribute	Default
Number of Clumps	`0`
Number of Lights	`0`
Data	`NULL`

Spline Object Defaults

Attribute	Default
Type	Defined at creation time
Points	Defined at creation time
Number of Points	Defined at creation time
Data	`NULL`

System Information Defaults

Attribute	Default
`rwVERSIONSTRING`	Library dependent
`rwVERSIONMAJOR`	Library dependent
`rwVERSIONMINOR`	Library dependent
`rwVERSIONRELEASE`	Library dependent
`rwFIXEDPOINTLIB`	Non-zero for fixed point library Zero for floating point library
`rwDEBUGGINGLIB`	Non-zero for debugging library Zero for production library

Texture Object Defaults

Attribute	Default
Name	Computed for named textures `NULL` for other textures
Frame	`0`
Frame Step	`+1`
Number of Frames	Defined at creation time
Raster	Defined at creation time
Mask	Unmasked
Dithering	Defined by the global dithering mode for a texture read by **RwReadTexture**() or **RwReadNamedTexture**() Defined by the specified raster options for a texture created from a raster created by **RwReadRaster**() or **RwBitmapRaster**().
Gamma Correction	Defined by the global gamma correction mode for a texture read by **RwReadTexture**() or **RwReadNamedTexture**() Defined by the specified raster options for a texture created from a raster created by **RwReadRaster**() or **RwBitmapRaster**()
Data	`NULL`

Texture Dictionary Defaults

Attribute	Default
Textures	None
Search Mode	`rwGLOBAL`

UserDraw Object Defaults

Attribute	Default
Alignment	Defined at creation time
Call-back	Defined at creation time
Offset	Defined at creation time
Size	Defined at creation time
Type	Defined at creation time
Owner	`NULL`
Data	`NULL`
Vertex Index	Undefined
Parent Alignment	Undefined

Vertex Defaults

Attribute	Default
Coordinates	Defined at creation time
Texture coordinates	`[CREAL(0.5), CREAL(0.5)]`
Normal	Computed
Viewport Position	Computed