VDF - A File Format for the Interchange of Virtual Worlds

Bernie Roehl and Kerry Bonin
Version 1.00
November 1994

Premise

As the Virtual Reality industry matures, there is an increasing need for standardization. In particular, there is a need for a standard file format for storing descriptions of both individual objects and entire worlds.

This document proposes such a file format, called VDF (Virtual world Description Format). It addresses such issues as the description of object geometry, object hierarchy, material properties, and various other elements that are typically used to describe a virtual environment. The proposed format is not intended to replace the native formats used by the various VR systems; instead, it is designed to be a kind of "lingua franca" for the exchange of world descriptions. In this sense, it's similar to the Rich Text Format often used for the interchange of word processing documents.

The primary purpose of the format is to enable third party developers to create and distribute objects and worlds that can be used by the entire VR community. Object creation is one of the more tedious aspects of world-building; having a standard format allows large databases of virtual objects to be created more easily than they can be at present. A standard format not only simplifies the work of the world-builder, it also enhances the usefulness of all VR systems since their users gain access to a larger collection of resources.

Some Background

The specifications for this format grew out of an informal discussion held at the Meckler Virtual Reality Conference in May of 1994. Representatives from VREAM, Superscape, Straylight, Virtek and others were present; the discussion was lively and open, and there was general agreement on the need for a standard file format.

Why Not DXF?

The only format that all vendors currently support is DXF. However, DXF was never intended for this purpose; it was originally designed to meet

the needs of CAD users. It therefore has many features which are of no use to VR developers, and lacks many features that VR developers would find useful. Many of the existing DXF translators and input routines support only a subset of the full DXF specification, often forcing world-builders to "tweak" the files after they are imported. DXF also makes no provision for storing world layout or object hierarchy descriptions.

Basic Syntax

A VDF file is considered to be a stream of printable ascii characters. The advantage of an ascii format is that it's human-readable; it can be printed on paper, created or modified using a text editor, sent via email, and so forth. It also has the advantage of platform-independence; concerns about word lengths, byte-ordering, floating-point formats and so on are eliminated.

The file is free-format; line boundaries are ignored. Spaces, tabs, carriage returns and line feeds are all treated simply as whitespace. The use of whitespace to enhance readability is strongly encouraged; in particular, indentation should be used to make clear the structure and organization of the data.

The file consists simply of a series of tagged items of the format

tagname { ... }

The { } characters enclose data that is specific to the tag. The '{' and '}' characters must be surrounded by whitespace on either side, to simplify parsing. Each tagged item can contain other tagged items, nested to any level; the resulting file looks somewhat like a C program:

```
top-level-tagname
        {
        second-level-tagname
                {
                third-level-tagname
                        {
                        ...
                        }
                another-third-level-tagname
                        {
                        }
                }
        another-second-level tagname
                {
```

```
        ...
        }
}
```

Any unrecognized tags are simply ignored, along with any items contained within them; this ensures that the format is extensible and that old parsers will still be able to deal with newer versions of the format. Ignoring a tag consists of simply skipping everything within the matching { } markers; however, see the description of the STRING data type later in this document for an important exception to this. The ordering of tags is generally unimportant, except as specifically noted below.

Comments use the C++ "//" convention; that is, anything on the current line after a pair of slashes is ignored. More specifically, everything from the slashes through the first carriage return (0x0D) or linefeed (0x0A) is ignored.

The syntax is designed to be very easy to parse; in any case, sample parsers will be freely distributed to any interested parties. Also provided will be a small "test suite" of objects and worlds that will allow anyone developing a parser to verify consistency with the standard.

The vast majority of tags are optional; they can be omitted by world-builders and ignored by import and translate functions. It is important to note that importing an object into a VR system and re-exporting it may involve a loss of information, since not all VR systems support all the features specified in the format. For example, some VR systems only support triangles; when importing a file containing facets (polygons) with more than three sides, the facets are decomposed into triangles. If the file is re-exported again, the original *n*-sided facet information is no longer available.

Fundamental Data Types

There are several data types that are used throughout this document. In addition to the types listed below, some tags may use additional types that are tag-specific.

REAL
: A signed, single-precision floating-point number. For example: -137.1294

NUM
: An unsigned decimal integer with 32-bit precision, typically used as an array index. For example: 15

STRING

A character string. Strings are always surrounded by double-quote (") characters. If a double-quote or a backslash must be included in a string, it should be preceeded by a backslash. For example, "This string contains a \"doublequoted\" string and a backslash (\\) character".

ID

An unsigned 32-bit integer, specified in either decimal or hex (hex uses the 0x prefix). IDs are only meaningful within a single file; they should not be used internally within an application. IDs are generated when the file is created, and are discarded after parsing. For example: 0x9348736

HANDLE

An unsigned 32-bit integer, specified in either decimal or hex (hex uses the 0x prefix). HANDLEs differ from IDs in that they are used by the application, not the parser. For example, if it's anticipated that the application may need to reference a particular facet within an object, the facet can be provided with a HANDLE.

DATE

To avoid the ambiguity associated with the mm/dd/yy vs dd/mm/yy conventions, a DATE is specified as YYYYMMDD where YYYY is the year, MM is the month (January == 01), and DD is the date. This is followed by the time in HHMMSS format, with the hours ranging from 0 to 23 and the minutes and seconds ranging from 0 to 59. For example: 19941104 130502.

BOOLEAN

Either of the values TRUE or FALSE. For example: TRUE

COLOR

An RGB triplet, where each of R, G and B is a REAL in the range 0.0 through 1.0 inclusive. For example: 0.2 0.3 0.2

FNAME

A lowercase STRING consisting of no more than 12 characters in the format ????????.???; all the characters (except the '.') should be alphanumeric, and the first should be alphabetic. This syntax is chosen for reasons of DOS compatability.

ANGLE

A REAL value giving a rotation angle in degrees.

UNAME

A STRING value identifying a specific user. The recommended format for this is the full name of the user, followed by his or her electronic mail address enclosed in "<>" characters. For example: John Q. Public <jqpublic@someplace.com>

Conventions

The definition of all object geometry and location/orientation information is done using a left-handed coordinate system, oriented so that if X points to the right and Y points up, then Z points forwards. All positive rotations around an axis are clockwise when viewed from the positive end of the axis looking towards the origin.

Wherever applicable, rotations are carried out in the order yaw, pitch, roll; in other words, YXZ.

Tags

Tag names consist of alphanumeric characters, plus '_'; the first character must be alphabetic. No specific length limit is imposed on tag names, but they ought to be unique in the first 32 characters. Tag names are not case-sensitive; "thing", "THING" and "Thing" are all equivalent.

There are several tags which are widely used throughout this specification; their use is described here:

Identifier { ID }
Application_handle { HANDLE }
Name { STRING }
Count { NUM }

An Identifier allows an entity to subsequently be referenced; it can be thought of as an "address". Only entities that will be referenced elsewhere in the file should be given an Identifier. Identifiers are only meaningful within a single file, and are discarded after parsing.

The Application_handle allows an entity to be assigned a handle by which it can be referenced by the application software. Any entity can also be assigned a Name; its use is application-dependent.

The Count is used to specify things like array sizes; if a Count field is omitted, the value defaults to 1. Wherever a Count is used, it must appear

before any of the elements of the array whose size it specifies.

File Inclusion

There is one tag that may appear at any point in the file where a tag is valid:

```
Include { FILENAME }
```

Files that are Included should be treated as if the entire contents of the file had been inserted in place of the Include. Included files can include other files, but there may be some limits imposed by the underlying operating system as to the number of simultaneously open files.

Overall File Structure

The file is made up of a series of tagged entities. Additional entity tags may be defined later, but those that are currently defined are World_information, Palettes, Maps, Materials, Material_tables, Shapes, Objects, Lights, Cameras, Sounds, and World_attributes; they appear in that order in the file. In other words, the World_informaton should appear first, followed by the Palette, followed by all the Maps, followed by Materials, Material_tables, Shapes, and so on.

World Information

This optional entity contains textual information about the world described in a VDF file.

```
World_information
        {
        Created_by
                {
                Person { UNAME }
                Date { DATE }
                Comment { STRING }
                }
        Modified_by
                {
                Person { UNAME }
                Date { DATE }
                Comment { STRING }
                }
        Copyright_message { STRING }
        Usage_restrictions { STRING }
```

```
Title { STRING }
Comment { STRING }
}
```

The Modified_by tag can appear more than once.

Palettes

A palette has two parts to it: an array of RGB triplets that give the colors for individual hardware palette entries, and a hue map that relates a hue and brightness to a palette entry.

```
Palette
        {
        Color_table
                {
                Count { NUM }
                Color_entry { COLOR }
                Color_entry { COLOR }
                ...
                }
        Hue_table
                {
                Count { NUM }
                Hue_entry { NUM NUM }
                Hue_entry { NUM NUM }
                ...
                }
        }
```

The use of palettes is entirely optional, and on non-paletted systems they will be ignored. However, on paletted systems, a material will have a hue value associated with it which is used to index the Hue_table. The Hue_table in turn gives the range of palette entries associated with that hue; the first NUM is the index into the Color_table of the darkest color of the hue, and the second is the index of the lightest.

Maps

A map is mechanism for relating a string to a filename. For example, a particular Material may reference an image map called "wood grain", which could be associated (through a Map) with the file "wgrain01.pcx". A Sound may refer to a sample called "door chime", which might be Mapped to "dingdong.wav".

A Map entry has the following format:

```
Map
    {
    Name { STRING }
    Filename { FNAME }
    }
```

Both the Name and Filename tags are required, since the only purpose of a Map entity is to provide the correspondance between the two. Map entities only provide a means of connecting a platform-independent map name with a local filename; typically, an Include tag is used to provide a set of local Maps for a world.

Materials

A Material is a definition of what a surface looks like; it can contain a great deal of information (needed on some systems), but the only required tags are the Identifier and either the Diffuse_color or the Hue. All other tags can be omitted when creating the file, and ignored during parsing. The format of a Material is as follows:

```
Material
    {
    Identifier { ID }
    Name { STRING }
    Rendering_mode { type }
    Hue { NUM }
    Albedo { REAL }
    Diffuse_color { COLOR }
    Ambient_color { COLOR }
    Transparency_color { COLOR }
    Specular_color { COLOR }
    Specular_exponent { REAL }
    Refractive_index { REAL }
    Texture_map { Name { STRING } }
    Bump_map { Name { STRING } }
    Opacity_map { Name { STRING } }
    Reflection_map { Name { STRING } }
    Reflection_blur { REAL }
    Transparency_falloff { REAL }
    }
```

The Rendering_mode type can be one of WIREFRAME, UNLIT, FLAT,

GOURAUD or PHONG. The default should be FLAT if it's available, otherwise UNLIT. UNLIT Materials always have the same color, regardless of lighting; FLAT Materials have a constant color across their surface, the brightness of which varies depending on lighting. Many of the parameters are included just for the sake of completeness, and for VR systems that pre-compute much of the surface property information. The Hue is only used in paletted systems, to select an entry from the Hue_table (and ultimately, a color from the palette). The Albedo is the overall reflectivity coefficient of the surface.

The various maps (texture, bump, etc) are identified by name; for example:

```
Texture_map { Name { "wood grain" } }
```

The idea is to allow a single map to be used repeatedly, and in a number of ways by different Materials. There should be a Map entry earlier in the file that relates "wood grain" to the name of a file from which the texture map can be loaded.

Texture should be stored in external files as rectangular arrays of numeric values that can be used as a regular texture (by systems which support texture mapping) or as a bump or opacity map (for those few systems which support such things). Image maps in a variety of formats can be supported; the type of map, as well as its dimensions and "depth", are all determined from the external file in which the map is stored. Recommended map formats are PCX (for 8-bit-deep maps), TGA (for 24-bit-deep maps) and FLC (for animated maps).

Note that on some systems, there may be support for "procedural" maps that are defined by code rather than data. For those maps, no Map entities are required.

There is some potential for confusion here, since the word "map" is being used to refer to both a mapping from a string to a filename, and to refer to an image map. It should be clear from context which is the intended use.

Note that a Material can be very simple:

```
Material
        {
        Identifier { 0x782938 }
        Diffuse_color { .5 .3 .1 }
        }
```

When converting any COLOR value to a fixed palette, the palette entry with the minimum "distance" from the specified RGB color should be used. This mapping can be done when the Material is processed. Better results may be obtained by "weighting" the colors; the expression would be something like

```
distance    = 0.3 * (red - palette[i].red)
            + 0.6 * (green - palette[i].green)
            + 0.1 * (blue - palette[i].blue)
```

Needless to say, there's no need to compute the square root since we're just choosing a palette entry which gives the minimum distance.

Material_tables

References to Materials are kept in a Material_table, which in turn is referenced by any number of different Shapes or Objects. A Material_table has the following format:

```
Material_table
        {
        Identifier { ID }
        Application_handle { HANDLE }
        Count { NUM }
        Material_reference { ID }
        Material_reference { ID }
        ...
        }
```

Every Material_table should have an Identifier. Each Material_reference tag contains the ID of the corresponding Material.

Shapes

A Shape is a geometric description, consisting of a collection of vertices and facets. It is not the same as an Object, which stores location, orientation and attachment information.

The format of a Shape is as follows:

```
Shape
        {
        LOD_size { NUM }
        LOD_replaces { ID }
```

```
Identifier { ID }
Name { STRING }
Is_convex { BOOLEAN }
Application_handle { HANDLE }
Bounding_box { REAL REAL REAL REAL REAL REAL }
Uses_material_table { ID }
Vertex_list
        {
        Count { n }
        Vertex { ... }
        Vertex { ... }
        ...
        }
Facet_list
        {
        Count { n }
        Facet { ... }
        Facet { ... }
        ...
        }
}
```

The Uses_material_table tag is optional; if present, it specifies the ID of a Material_table used by this Shape. Note that the Material_table for a Shape is a kind of default value, which is only used if an Object doesn't specify one. The conversion to an internal format can, of course, be done once at load time.

The Bounding_box is entirely optional, and is intended only to give a rough idea of how the vertex values should be scaled. It specifies the minimum X, Y, and Z values followed by the maximum X, Y, and Z values; the actual bounding information can (and should) be derived from the coordinates of the vertices comprising the Shape.

The Is_convex tag is optional; the default value is FALSE. Some systems take advantage of object convexity for hidden-surface removal or collision detection; if an object is known to be convex, it should be flagged as such.

The LOD_size and LOD_replaces tags have to do with automatic level-of-detail switching. Systems that don't support automatic LOD should just ignore any Shapes that have an LOD_replaces field set; to simplify processing, the LOD tags (if used) should be the first tags in a Shape. The LOD_replaces tag gives the ID of the Shape that this Shape is a replacement for when the Object is greater than a given on-screen size. The LOD_size is

the approximate on-screen size of the Object in pixels at which this version of the Shape should become the replacement. Replacement Shapes should be specified *after* the Shapes they're replacing; in other words, the Shapes should be given in increasing order of detail level.

Vertices

A Vertex always contains an X, Y, Z triple; it may also have additional information associated with it. The format for a Vertex is:

```
Vertex
        {
        Point3D { REAL REAL REAL }
        Normal3D { REAL REAL REAL }
        Color { COLOR }
        Application_handle { HANDLE }
        }
```

The Point3D contains the X, Y and Z values of the vertex coordinates. The Normal3D tag contains a vertex normal (used for Gouraud and Phong shading); this normal need not be of unit magnitude. The Color tag is used for systems that do a kind of Gouraud shading without the lighting calculation.

A very simple vertex list might look like this:

```
Vertex_list
        {
        Count { 3 }
        Vertex { Point3D { 15 12 17 } }
        Vertex { Point3D { 27  5 18.6 } }
        Vertex { Point3D { 2 129 27.9 } }
        }
```

Facets

A Facet (also called a "face" or a "polygon") is defined to be a flat, convex shape with an arbitrary number of vertices. "Flat" means that all the vertices in the Facet are co-planar; "convex" means the corners of the Shape all "point" outwards. Facets are not permitted to have holes in them.

The format for a Facet is as follows:

```
Facet
        {
        Vertex_data
                {
                Count { n }
                Vertex_info { Index { NUM } }
                Vertex_info { Index { NUM } }
                ...
                }
        Is_doublesided { BOOLEAN }
        Is_interior { BOOLEAN }
        Base_facet { ID }
        Front_material { NUM }
        Back_material { NUM }
        Normal3D { REAL REAL REAL }
        Identifier { ID }
        Application_handle { HANDLE }
        }
```

Only the Vertex_data tag is required. Each of the Index values references an element of the Shape's Vertex array. The vertices are numbered starting from zero, and are listed in a clockwise order as seen from the "front" of the Facet. If the Count is 1, a point should be created rather than a Facet; similarly, a Count of 2 defines a line.

If Is_doublesided is TRUE (the default value is FALSE), then the Facet can be seen from both sides; the Back_material is used to paint the back side of the Facet. The Front_material and Back_material are both indices into the Materials table for the Object or Shape. If the Back_material is omitted, it defaults to the value of the Front_material.

A Facet flagged as being Is_interior is on the inside of a concave Shape; for example, a cup would have all the inside surfaces (the ones in contact with liquid if the cup were full) flagged as interior. Not all systems will make use of this flag, but object designers would do well to include it since it's easy to set and is potentially of great benefit to those VR systems which do make use of it.

If a Base_facet tag is specified, then the current Facet is a kind of "decal" that appears only on the surface of the Base_facet. If the Base_facet is not visible (i.e., is backfacing or completely clipped) then no processing needs to be done on the current Facet. The current Facet is always drawn immediately after its Base_facet; no additional sorting is required. The Base_facet must be defined prior to being referenced.

The Normal3D tag, if specified, contains a vector perpendicular to the plane of the Facet and pointing outward from the "front" of the Facet. It need not be a unit vector.

A minimal Facet might look like this:

```
Facet
        {
        Vertex_data
                {
                Count { 5 }
                Vertex_info { Index { 3 } }
                Vertex_info { Index { 2 } }
                Vertex_info { Index { 8 } }
                Vertex_info { Index { 7 } }
                Vertex_info { Index { 5 } }
                }
        Front_material { 7 }
        }
```

This would define a 5-sided Facet, using vertices 3, 2, 8, 7 and 5. It would use Material number 7 in an Object's Material_table; if the Object does not have a Material_table, the Material_table for the Shape is used. If the Shape has no Material_table either, then the Object is invisible.

In a very simple system (fixed 256-color palette, no lighting, no Material_tables at runtime, etc) then the Front_material would be used to index an array of bytes that contain the offsets into the palette of the various colors.

Note that edge information is not explicitly stored in the file, but can easily be derived for those systems that require it.

Objects

An Object consists of a pointer to a Shape, a specification of location and orientation, an optional Material_table and other information. The complete format is:

```
Object
        {
        Name { STRING }
        Identifier { ID }
        Instance_of_shape { ID }
        Scaled_by { REAL REAL REAL }
```

```
Uses_material_table { ID }
Location { REAL REAL REAL }
Rotation { REAL REAL REAL }
Attached_to { ID }
Contained_within { ID }
Is_invisible { BOOLEAN }
Layer { NUM }
Text { STRING }
Facet_behind { ID }
Application_handle { HANDLE }
}
```

The Instance_of_shape tag specifies the ID of the Shape this Object uses for its geometry information. The Scaled_by tag specifies the scaling factors to be applied to the basic shape for this object in each of the X, Y and Z axes; if omitted, the values are 1,1,1. The Location tag gives the location in X, Y and Z of the origin of the Object in world coordinates, or in the coordinate system of the Object it's Attached_to (if any); the default value is [0,0,0]. The Rotation tag gives the angles of rotation around X, Y and Z (although they are not performed in that order); the default is [0,0,0].

The Attached_to tag gives the ID of the Object to which this Object is attached, to establish a hierarchy. The parent Object should be defined before its children. The Contained_within tag gives the ID of the Object inside which this Object is contained; the container should be specified before the contents.

The Uses_material_table tag references a Material_table to be used by this Object; this tag is optional, and if it's omitted then the material table from the Shape is used instead. If the Shape has no material table either, the Object should be invisible.

The Is_invisible tag indicates whether the Object can be seen or not. By default, all Objects are visible. Another way to make an Object invisible is to omit its Instance_of_shape tag; this is useful for grouping Objects together, or for providing location and orientation information for Cameras and Lights that are not associated with a visible Object.

The Layer is simply a number, and the Text is simply a string; their use is application-dependent. The Facet_behind tag is used as a hint for sorting; it specifies the ID of an already-specified Facet on another Object, to which this Object should be attached for purposes of visibility determination. If the Facet is visible (i.e., not backfacing), this Object is drawn after the one the Facet belongs to; otherwise, this Object is drawn first.

Lights

A world can contain any number of Lights; however, they should be listed in order of importance since most systems impose some kind of limit. The format of a Light is:

```
Light
        {
        Name { STRING }
        Application_handle { HANDLE }
        Type { TYPE }
        Associated_with { ID }
        Color { COLOR }
        Hotspot { ANGLE }
        Falloff { ANGLE }
        Is_on { BOOLEAN }
        Casts_shadows { BOOLEAN }
        }
```

The Type is one of DIRECTIONAL, POINT or SPOT; the default is DIRECTIONAL. The Associated_with tag contains the ID of the Object that this Light uses for position and orientation information. Remember that Objects can be flagged as being invisible, and need not specify a Shape. Lights default to being "on", and their color defaults to 1, 1, 1.

The default value for the Is_on tag is TRUE; it need only be specified if the Light should be off initially. Most systems can ignore most of the information, including anything related to shadows; it's included for completeness, and for renderers that precompute much of the surface material information.

Cameras

A world can contain any number of Cameras; the first Camera specified in the file should be active when the world is initialized. The format of a Camera is:

```
Camera
        {
        Name { STRING }
        Application_handle { HANDLE }
        Field_of_view { ANGLE }
        Aspect_ratio { REAL }
        Associated_with { ID }
```

```
Projection_type { TYPE }
}
```

The Field_of_view is measured horizontally; the Aspect_ratio can be used to compute the vertical field of view. The default Field_of_view is 45 degrees, and the default Aspect_ratio is 1.33. The Associated_with tag specifies which Object this Camera is associated with; the Camera uses that Object's location and orientation information. Remember that Objects can be flagged as being invisible, and need not specify a Shape. A Camera without a visible Object should still use an invisible Object to get its location/orientation/attachment information from. The Projection_type is one of PERSPECTIVE or PARALLEL, the default being PERSPECTIVE.

Sounds

A world can contain any number of sound sources, some of which may be associated with specific Objects.

```
Sound
    {
    Name { STRING }
    Application_handle { HANDLE }
    Associated_with { ID }
    Is_on { BOOLEAN }
    Volume { REAL }
    Sample_name { STRING }
    }
```

The Associated_with tag is optional; if present, the Sound should appear to come from the Object with the specified ID. The STRING in the Sample_name tag is mapped by a Map entry to a filename which sound data should be loaded from; that file will also contain type and other information (such as sampling rate and bits per sample).

World Attributes

In addition to Objects, Cameras, Lights and Sounds, a virtual world may have a large number of properties. The following list provides a starting point:

```
World_attributes
    {
    Gravity_vector { REAL REAL REAL }
    Ambient_light { COLOR }
```

```
Has_horizon { BOOLEAN }
Sky_color { COLOR }
Ground_color { COLOR }
Fog_color { COLOR }
Scale { REAL }
}
```

The Scale is the number of millimeters per "unit" of distance in this world, and defaults to 1. Eventually, a Local_attributes tag may be defined in order to allow the various attributes to vary from place to place within the virtual world.

An Example

```
// Three cubes, a camera and a light

Material { Identifier { 0x3A97 } Diffuse_color { 1 0 0 } }  // red
Material { Identifier { 0x4873 } Diffuse_color { 0 1 0 } }  // green
Material { Identifier { 0x9798 } Diffuse_color { 0 0 1 } }  // blue

Material_table
    {
    Identifier { 0x1C756 }
    Count { 3 }
    Material_reference { 0x3A97 }
    Material_reference { 0x4873 }
    Material_reference { 0x9798 }
    }

Shape
    {
    // two red faces, two green faces, two blue faces
    Identifier { 0x1234 }
    Uses_material_table { 0x1C756 }
    Is_convex { TRUE }
    Vertex_list
        {
        Count { 8 }
        Vertex { Point3d { 100 200 300 } }
        Vertex { Point3d { 700 200 300 } }
        Vertex { Point3d { 700 800 300 } }
        Vertex { Point3d { 100 800 300 } }
        Vertex { Point3d { 100 200 900 } }
        Vertex { Point3d { 700 200 900 } }
```

```
    Vertex { Point3d { 700 800 900 } }
    Vertex { Point3d { 100 800 900 } }
    }
Facet_list
    {
    Count { 6 }
    Facet
        {
        Front_material { 0 }
        Vertex_data
            {
            Count { 4 }
            Vertex_info { Index { 3 } }
            Vertex_info { Index { 2 } }
            Vertex_info { Index { 1 } }
            Vertex_info { Index { 0 } }
            }
        }
    Facet
        {
        Front_material { 1 }
        Vertex_data
            {
            Count { 4 }
            Vertex_info { Index { 2 } }
            Vertex_info { Index { 6 } }
            Vertex_info { Index { 5 } }
            Vertex_info { Index { 1 } }
            }
        }
    Facet
        {
        Front_material { 1 }
        Vertex_data
            {
            Count { 4 }
            Vertex_info { Index { 6 } }
            Vertex_info { Index { 7 } }
            Vertex_info { Index { 4 } }
            Vertex_info { Index { 5 } }
            }
        }
    Facet
        {
        Front_material { 2 }
```

```
            Vertex_data
                {
                Count { 4 }
                Vertex_info { Index { 4 } }
                Vertex_info { Index { 7 } }
                Vertex_info { Index { 3 } }
                Vertex_info { Index { 0 } }
                }
            }
        Facet
            {
            Front_material { 2 }
            Vertex_data
                {
                Count { 4 }
                Vertex_info { Index { 7 } }
                Vertex_info { Index { 6 } }
                Vertex_info { Index { 2 } }
                Vertex_info { Index { 3 } }
                }
            }
        Facet
            {
            Front_material { 0 }
            Vertex_data
                {
                Count { 4 }
                Vertex_info { Index { 5 } }
                Vertex_info { Index { 4 } }
                Vertex_info { Index { 0 } }
                Vertex_info { Index { 1 } }
                }
            }
        }
    }

// Now that we've defined the shape, let's instance it three times

Object { Instance_of_shape { 0x1234 } Location { 0 0 0 } }

Object { Instance_of_shape { 0x1234 } Location { 1000, 0, 2000 } }

Object { Instance_of_shape { 0x1234 } Location { 1000, 1000, 3000 } }

// And finally, here are the light and the camera
```

Object { Name { "lightsource" } Identifier { 0x9012 } Location { 0 0 0 } }

Object { Identifier { 0x5678 } Location { -1000 -1000 -1000 } Rotation { 0.25 0.25 0 } }

Light { Associated_with { 0x9012 } }

Camera { Associated_with { 0x5678 } }

Some Closing Notes

This is a late-stage draft of the standard, and is not expected to change significantly. Any comments regarding this document should be sent to Bernie Roehl (broehl@uwaterloo.ca).