Formula Graphics Multimedia System

16 bit version 5.7
32 bit version 5.7

System Instructions

Multimedia Instructions

Connectivity Instructions

Examples

Harrow Software Pty Ltd

Email: formula@magna.com.au

World wide web:
http://www.harrow.com.au/formula
http://www.magna.com.au/~formula

Australia phone: (02) 9967 4033
Australia fax: (02) 9810 2077

International phone: (612) 9967 4033
International fax: (612) 9810 2077

Surface mail:
GPO Box 1722
Sydney NSW 2001
Australia

Formula Graphics Multimedia System

Multimedia, game and internet development system for Windows 3.1, 95 and NT.

Introduction

Formula Graphics can be used to develop interactive multimedia slideshows and presentations. It has an easy to use graphical interface and there are no limits to the sounds, images, animations or interactivity that can be produced.

Formula Graphics can be used to develop business applications that open databases and download files and other information from the internet. Information can be displayed using both hypertext and graphics. Formula Graphics has a powerful object oriented language with over 500 operators, functions and instructions.

Formula Graphics has programmable 3D graphics and arcade quality sprites which can be used to develop highly animated action and adventure games.

Presentations can be played from floppy disk, or cdrom, or played directly through the internet, or an intranet, or embedded in a web page or other document. They can be installed as screen savers, or even screen savers that play through the internet !.

Formula Graphics has dynamic palette management and supports 8, 16, 24 or 32 bit color.
It supports many bitmap and animation file formats including JPG, FIF, FLC, AVI and internet GIF animations. It can do sophisticated batch conversions between these formats.

Formula Graphics can be downloaded and used for free.

Features

Formula Graphics has plain, tiled or stretched background bitmaps, stretched and rotated pictures with transparency, colored text with drop shadows, Help compatible hypertext, HTML, 256 color or 16 bit color synchronous or asynchronous animations with transparency, AVI or MOV video, 3D key framed animations, 3D key framed animations, animated 3D truetype text with bevels and texture mapping, windows sound, midi and CD audio, text and picture buttons, hot areas and hot colors with unlimited interactivity. Edit boxes, list boxes, combo boxes, menus, progess bars, 2D and 3D graphs.

Another key feature is the high performance object oriented multimedia language. The language has over 500 different instructions, functions and operators. It features bitmap manipulation, sprite collisions, database connectivity, 3D modeling and internet programming and can be used to give unlimited amounts of interactivity to your
games and presentations.

Formula Graphics is the only multimedia system that has dynamic palette management for 256 color presentations. As the presentation plays, palette colors are allocated and deallocated and images are mapped to the changing palette. Any number of bitmaps and animations with different palettes can be displayed on the same screen.

Animation is the strongest feature of Formula Graphics. The system handles five animation file formats and four styles of playback, as well as Video for Windows. Animations can be made to blend subtly into the background as complex shapes rather than appearing as boring square boxes.

Using Formula Graphics powerful 2D and 3D animation tools you can include 3D Studio mesh files directly into your presentation. You can also animate 3D beveled title text with texture mapping for real graphical impact.

Formula Graphics games and presentations can be played as standalone executables. They can be played directly across a network or the internet. They can be installed to play as screen savers. Formula Graphics includes a customizable setup utility.

Formula Graphics provides the quickest and easiest way of utilizing all of the multimedia power that the Windows operating system has to offer. It is so simple to use and yet it places no limits on the author's creativity. No other package has found such an effective solution to multimedia authoring.

Bitmaps and palettes

The way a computer works is by using bits. A bit can be either zero or one. Groups of bits can be used to represent numbers. Bits are usually grouped together in multiples of 8. With all the possible combinations of zeros and ones, 8 bits can be used to represent any number between 0 and 255.

Bitmaps

A bitmap is a way of storing an image. The image is divided up into tiny units of color called pixels. The size of a bitmap can be given as X pixels wide and Y pixels high. The color resolution of the bitmap refers to the number of bits used to store each pixel color.

A 256 color bitmap uses 8 bits per pixel and has a corresponding table of colors called a palette. Each pixel can have a value between 0 and 255, and each value refers to the position of a color in the palette. Each color in the palette has 8 bits of red, 8 bits of green and 8 bits of blue.

A 16 bit color bitmap does not use a palette, the red, green and blue color components of each pixel are stored using 16 bits. There are two variations: RGB555 uses 5 bits of red, 5 bits of green and 5 bits of blue (32768 colors). RGB565 uses 5 bits of red, 6 bits of green and 5 bits of blue (65536 colors).

A 24 bit color bitmap has 8 bits of red, 8 bits of green and 8 bits of blue for each pixel. There are 16 million possible color variations and the smallest differences between them can barely be distinguished by the eye.

A 32 bit color bitmap has 8 bits of red, 8 bits of green, 8 bits of blue, and 8 bits of alpha channel for each pixel. The alpha channel decides the level of transparency for each pixel in the image. An alpha value of 0 is totally transparent, and a value of 255 is totally opaque.

Loading a bitmap

After selecting **Load Bitmap** from the Graphics menu, a Load Bitmap dialog box will open. A bitmap file can be chosen. This bitmap will be displayed in the graphics window. If the color resolution of the bitmap differs from the color resolution of the graphics window, then the bitmap will be converted before it is displayed.

The **Subtract Bitmap** function can be used to remove a background image from a bitmap image. First load the bitmap file into the graphics window. Then select Subtract Bitmap from the Graphics menu and the background file can be selected from the Subtract Bitmap dialog box. The two images will be compared and all corresponding pixels with the same color will be set to the specified subtract color.

An area of the graphics window can be selected with a capture rectangle before choosing **Save Bitmap** from the Graphics menu. After choosing a bitmap file name, the selected area will be captured (converted into a bitmap) and saved. If no area was selected, then the entire contents of the graphics window will be captured and saved. If the color resolution of the chosen file format differs from the color resolution of the graphics window, then the image will be converted after it is captured.

Bitmap file formats

Bitmap files come in a variety of formats. Some formats store the image as raw data, and other formats use compression techniques to reduce the amount of space required to store the image.

The **BMP** format can either be 256 colors, 16 bit color (RGB 555) or 24 bit color. This format does not use compression. It is a convenient format for exchanging bitmaps between different applications.

GIF is a 256 color file format. It uses a technique called LZ compression which has an excellent

compression ratio and reasonable decompression speed. This format is very popular on the internet. A built in transparency color can be chosen.

JPG is a 24 bit color format. The JPEG compression technique can achieve a high compression ratio at the cost of a loss in image quality. The format is slow to decompress. The compression ratio can be adjusted in the bitmap options.

FIF is a 24 bit color format. It's fractal compression technology can achieve a high compression ratio at the cost of a slight loss in image quality. The format is very slow to compress but reasonable to decompress. The compression ratio can be adjusted in the bitmap options. This format requires the deco_16.dll or deco_32.dll in the project directory. Iterated Systems allows free distribution of these DLLs.

The **TGA** format can either be 256 colors, 16 bit color (RGB 555), 24 bit color, or 32 bit color (with an alpha channel). It can be uncompressed or it can use run length compression. It is a popular format for exchanging bitmaps between different platforms.

VDO is a custom 256 color bitmap file format. This format uses run length compression. It has a good compression ratio and fast decompression.

WDO is a custom 16 bit color (RGB555) bitmap file format. This format uses run length compression. It has a good compression ratio and fast decompression.

For 256 color presentations the GIF format is a good choice, although for large areas of the same color the VDO format may be faster. For 16 or 24 color presentations the JPG format offers excellent compression although the WDO is a better choice for speed.

Palettes

A palette is a table of colors. There are 256 colors in a palette, and each color has 8 bits of red, 8 bits of green and 8 bits of blue. Palettes are only used with 256 color bitmaps or 256 color video modes. 256 color bitmaps are the most common types of bitmaps, and most video cards have 256 color modes.

Microsoft Windows maintains a palette called the system palette. When Windows is running in a 256 color video mode, the system palette is used by the video card. Windows reserves 20 colors in the system palette for visual elements such as windows and icons, this leaves only 236 colors which can be used for presentations.

When Formula Graphics is the active application, it takes control of the system palette. When a 256 color bitmap is displayed, first the colors in the bitmap's palette will be copied to the system palette, then the bitmap will be displayed using those colors.

Optimizing a palette

An area can be selected with a capture rectangle before choosing **Optimize Palette** from the Graphics menu. After choosing the appropriate options and pressing the Optimize button, the selected area of the graphics window will be analyzed and a new palette will be constructed using the most popular colors. If no area was selected, then the entire contents of the graphics window will be analyzed.

The number of colors to be used in the new palette can be specified, all other colors will be set to black. If the Foreground bias option is set, then the new palette will be made up of 3/4 popular colors and 1/4 wide spectrum of colors. This wide spectrum will include all used color groups regardless of how seldom they were used.

A bias color can be given. Preference will be given to colors that are nearest to the bias color. Biasing towards light colors generally produces a palette of more useful and visually appealing colors. The default

settings in the color optimization dialog box will usually produce the best results.

Converting bitmaps

After selecting **Bitmap Convert** from the Graphics menu, the graphics window will open with the size and color resolution given in the graphics options, and a Convert Bitmap dialog box will open.

The source file(s) can be an animation file, or any number of bitmap files. Each frame of an animation will become a separate bitmap. Multiple bitmaps can be selected by holding down the shift or control keys.

If the **Display Source** button is pressed, then the first source image will be loaded and displayed.

If the **Quick Palette** button is pressed, then the first source image will be displayed and analyzed and a new palette will be created.

If the **Optimize Palette** button is pressed then each source image will be displayed and analyzed, and a new palette will be created using the best range of colors across all images. You can choose to analyze every Xth image to speed up the process.

A destination directory and file format must be chosen. When the **Convert Bitmaps** button is pressed, the source images will be converted into destination bitmaps. Depending on the destination format another dialog box may appear with additional conversion options.

Options

The **direct copy** mode will bypass the graphics window. This mode can be used when you do not want the images to be effected by the graphics window. The source and destination file formats must have the same color resolution.

The **capture source image area** mode will display each image in the graphics window and then capture the image using its original size. This mode can be used for converting to a different color resolution.

The **capture window area** mode will display each image in the graphics window and then capture the image using the specified capture area. This mode can be used for cropping the size of the images. An area can be selected with a capture rectangle before pressing the Set button.

The **key color background** option can be used in cases where the source images have a transparency or alpha channel component. If these images were displayed directly over each other then the resulting source image may become distorted. This option will fill the graphics window with the key color given in the bitmap options before displaying each image.

The **remap to current palette** option will remap any 256 color images to the palette currently displayed in the graphics window before saving them. If this option is not set then the destination bitmap or animation will have the same palette as the first image.

Animations

An animation is a series of bitmaps. When the bitmaps of an animation are displayed one after the other, they give the appearance of movement. Each bitmap in an animation is called a frame, and a typical frame rate for an animation is 12 frames per second. Any slower and the motion becomes jerky, any faster and the system resources will be strained for no reason.

File formats

The **AVI** format can have any color resolution, can use any compression technique, and it can also contain a sound track. This format is supported by the MCI (Media Control Interface) component of the Windows operating system and is the recommended video file format.

FLC is a 256 color animation format. It uses run length compression which gives moderate compression ratios and good playback speed. This is a popular format and is convenient for exchanging 256 color animations between different applications.

GIF is a 256 color animation format. It uses a technique called LZ compression which has high compression for complex images but slow playback speed. This format is very popular on the internet.

VDO is a custom 256 color animation file format. This format uses run length compression. This format is optimized for this presentation system and is the recommended file format for 256 color animations.

WDO is a custom 16 bit color (RGB555) animation file format. This format uses run length compression. This format is also optimized for this presentation system and is the recommended file format for 16 and 24 bit color animations.

Animation Styles

Formula Graphics uses four styles of animation:

The first of these are **Simple** animations. Simple animations are stored with one complete image for each frame.

Delta animations have one complete image for the first frame. Then for each frame that follows, only the changes between this frame and the last frame are saved. This is the typical method of storing animations.

An **Overlay** animation is the same as a Delta animation except that the first frame of the animation contains only the differences between the first frame and some specified background image. This method can be used to save disk space by not storing the unused areas of background.

In a **Sprite** animation, each frame contains only the animated image. As each frame of the animation is displayed, the previous frame will be removed in such a way that the movement of the animation appears continuous. A sprite animation can have transparent areas and can be played over any background.

The performance of an animation will be a compromise between image quality and playback speed. The maximum playback rate of an animation will depend on the compression ratio of the file format. The main performance bottleneck will usually be the rate at which the animation can be read from the disk.

Converting an animation

After selecting **Animation Convert** from the Graphics menu, the graphics window will open with the size and color resolution given in the graphics options, and a Convert Animation dialog box will open.

The source file(s) can be an animation file, or any number of bitmap files. Multiple bitmaps can be selected by holding down the shift or control keys. The bitmaps will be combined together to make the final animation.

If the **Display Source** button is pressed, then the first frame of the source animation will be loaded and displayed.

If the **Quick Palette** button is pressed, then the first frame of the source animation will be displayed and analyzed and a new palette will be created.

If the **Optimize Palette** button is pressed then each frame of the source animation will be displayed and analyzed, and a new palette will be created using the best range of colors across the entire source animation. You can choose to analyze every Xth frame to speed up the process.

A destination animation file must be chosen. This file can have any one of the supported formats. Each different format has it own set of conversion options. The destination format will be decided by the extension on the animation filename.

When the **Convert Animation** button is pressed, the source file(s) will be converted into the destination animation. Depending on the destination format another dialog box may appear with additional conversion options.

Options

The **direct copy** mode will bypass the graphics window. This mode can be used when you do not want the images to be effected by the graphics window. The source and destination file formats must have the same color resolution.

The **capture source image area** mode will display each image in the graphics window and then capture the image using its original size. This mode can be used for converting to a different color resolution.

The **capture window area** mode will display each image in the graphics window and then capture the image using the specified capture area. This mode can be used for cropping the size of the images. An area can be selected with a capture rectangle before pressing the Set button.

The **key color background** option can be used in cases where the source images have a transparency or alpha channel component. If these images were displayed directly over each other then the resulting source image may become distorted. This option will fill the graphics window with the key color given in the bitmap options before displaying each image.

The **remap to current palette** option will remap any 256 color images to the palette currently displayed in the graphics window before saving them. If this option is not set then the destination bitmap or animation will have the same palette as the first image.

Formats

Converting to FLC, VDO and WDO animations

The style of the destination animation can be chosen. If this style is set to Overlay then the current image in the graphics window will be used as the background. The first frame of the source animation will be compared with this background and any pixels that are the same will become transparent.

If the style is set to Sprite then the destination animation will have transparent areas and will be able to be played over any background. There are two ways to select transparent areas of the animation.

One way is to use the currently displayed image as the background. Each source frame will be compared with this background image and any pixels that are the same will become transparent. Alternatively a transparency color may be chosen. Any source pixels that are the same as the transparency color will become transparent.

An antidither filter can be used to increase the length of horizontal runs and thereby increase the file compression and playback speed. Antidither will distort the image. Values below 50 have little visible effect but can significantly improve performance. The maximum value is 10000.

Converting to AVI video

If a sound file is specified then this sound will be saved synchronously with the animation to create a video. If the sound file is smaller than the video then the last segment of sound will be repeated. The default animation playback speed will be 12 frames per second.

You should also select your video compression scheme. The **Choose Compression** button will list all of the compressors that are currently installed on your computer. The default compression is "Microsoft Video 1".

Video for Windows and Windows 95 are distributed with a number of compression schemes. "Intel Indeo" compression usually offers the best performance. If you are using an exotic compression scheme which is not distributed with your operating system then you must distribute the decompressor with your presentation otherwise Windows will give an error.

Converting to GIF animation

GIF animations are played by displaying one frame after the other. Each frame of the animation can be saved with transparency so that it builds upon the previous frame. If the delta style is chosen then each source frame will be compared with the last frame and only the changes between each frame will be saved.

When the Repeat extension block is included in the file, and when the animation is shown as a resource in an HTML page on the world wide web, the GIF animation will be played over and over again repeatedly. The playback rate can also be chosen. A playback rate of zero will play the animation at the maximum possible rate.

Preparing to convert an animation

It is recommended that the size of the graphics window be set to the size of the animation and the color resolution be set to 24 bit color before converting any type of animation.

If the destination animation requires a palette then it will use the currently displayed palette. You should either load an image that uses the desired palette, or you should optimize the source animation to create one.

Converting to a sprite animation

A sprite animation can be played transparently over any background. A sprite animation should be converted to either the VDO format for 256 colors, or WDO for 16 or 24 bit colors. The GIF animation format can also used for 256 color sprite animations.

When you convert to a sprite animation you can choose to use the current background image. As the animation is being converted, each source frame will be compared with this background image and any pixels which are the same will become transparent.

Care must be taken to ensure that the background of each source frame uses exactly the same colors as

the current background image. Some animation programs such as 3D studio do not always produce identical backgrounds. Small variations can sometimes be ironed out of 256 color animations by using palette optimization. For best results you should use a 24 bit graphics window.

An alternative method is to choose a transparency color. If each source frame has a single color as its background then that color can be chosen as the transparency color. As the animation is being converted, each pixel in each source frame will be compared with this transparency color. Any pixels which are the same color will become transparent.

Care must be taken to ensure that the transparency color is different from any other color in the image. Care must also be taken with 256 color animations so that the chosen transparency color is only specified by one palette position. Optimizing the palette is recommended because after a palette has been optimized it will use only one position for each color.

Animation playback

After selecting **Animation Play** from the Graphics menu, a Load Animation dialog box will open and an animation file can be chosen. Then an Animation Playback window will open.

There are buttons to play and stop the animation, and to step forwards or backwards one frame, and to jump to the first frame or the last frame. The current frame number will be displayed.

Any frame rate can be given for the playback. A frame rate of zero will make the animation play at the maximum possible rate. The maximum rate will be the most number of frames that can be read from disk, decompressed, and displayed per second.

Screen capture

The Formula Graphics screen capture utility can be used to record screen images from the window with the current focus, the currently active application window, or the entire desktop.

Once the screen capture utility has been started, it will continue to operate until the <Break> key is pressed (or any system key combination is used). If Formula Graphics is in the way of the application you wish to capture then simply minimize the Formula window.

Graphics window

The screen capture utility requires the graphics window. Each captured image will first be copied from the desired area of the screen into the graphics window. The image will be cut to the size of the graphics window. The color resolution of the image will be converted to the resolution of the graphics window.

For the best results the graphics window should be set to the same color resolution as the video mode and the same color resolution as the destination bitmap or animation. For more details see the graphics window.

Capture mode

The "single frames to graphics window" will capture the screen but will not save it to disk. Each time the <SysRq> key is pressed, the desired image will be captured and copied to the graphics window.

The "single frames to numbered files" mode will save single bitmap images to disk. Each time the <SysRq> key is pressed, the desired image will be captured, copied to the graphics window, and saved to disk using the specified bitmap filename. The name of the bitmap file will then be changed to indicate a numerical increment.

The "Record all activity to animation file" mode will capture an image each Windows message that is generated. The captured image will then be saved to the end of the specified animation file. For every mouse movement or click another frame will be saved. Recording will begin when the <SysRq> key is pressed and continue until the <Break> key is pressed.

The "Record animation" mode will capture an image and save it to the animation file the specified number of times per second. Recording will begin when the <SysRq> key is pressed and continue until the <Break> key is pressed.

Capturing screen animation

For the best performance you should set the resolution of the graphics window to be the same as the resolution of the format you are saving to. The most common usage would be to capture to a 256 color animation format with a 256 color graphics window under a 65000 color video mode.

As the screen capture is being recorded to an animation, the animation will be optimized for speed of compression, rather than conservation of disk space. After saving to an FLC, VDO or WDO file you should reprocess the animation using the animation converter with the Delta format option set.

The advantage of saving an animation using the frames per second option is that any pauses in activity will also be recorded. Unfortunately some applications do not record properly in this mode.

If you choose the "record activity" option then only when the mouse moves or a button is clicked will a frame of animation be recorded. Using this mode you can synthesize delays by pressing the Scroll Lock button to capture one or more additional frames.

It is not possible to screen capture from a 16 color video mode.

Graphics window

Formula Graphics has a window called the graphics window, which can be used to display bitmaps and palettes. The size and color resolution of the graphics window can be set by choosing **Options** from the Graphics menu. The size can be specified in horizontal and vertical pixels. The color resolution can be 256 colors, 16 bits or 24 bits.

The graphics window uses the computer's memory to store the bits of color for all of its pixels. The amount of memory required depends on the width, height and color resolution of the window. A window of any size may be opened, but the operating system must have enough memory for the size you specify.

When a bitmap is displayed, its bits of color will first be copied to the memory of the graphics window, then the video display driver will be used to copy these bits of color to the video card. By using the graphics window memory as a buffer, presentations can be developed independently of the video hardware they run on.

See also Bitmaps and palettes.

Types of window

After choosing a suitable resolution from the Graphics Options, select **Open Window** from the Graphics menu and the graphics window will appear. There are a several different types of Graphics Window. Formula Graphics will choose the one which gives the best performance for the selected color resolution and video mode:

WinG: If the Microsoft WinG drivers are installed, then this type of 256 color window will give high performance on 256 color video modes under Windows 3.1 and 95.

CDS: This type of window is only available in the 32 bit version and will give the best possible performance on the Windows 95 and Windows NT operating systems.

VFW 256, 16 and 24: If Microsoft Video for Windows is available, then this type of window will provide color reduction and dithering for 256 colors on a 16 color video mode, or 16 bit (RGB555) or 24 bit colors on a 16 or 256 color video mode.

DIB 256 and 24: If no other alternatives are available, then this type of window will be used.

Recommendations

If you are producing a presentation for a wide consumer market then you must expect that many of the target machines will be using 256 color video display modes. You must also expect that a large number of those machines will be running Window 3.1.

In this case you should you should use a 256 color graphics window with 256 color artwork. 256 color presentations are relatively quick and light on storage space. The main disadvantage is the limited number of colors that can be displayed at one time.

You should use a 16 or 24 bit color graphics window when you can assume that the target machines will be using 16 or 24 bit color video display modes. In this case you can use 256 color artwork for speed and storage, or 24 bit artwork for luxury of colors and ease of use. 16 bit color is generally faster and uses less memory than 24 bit color.

When a 16 or 24 bit color presentation is played on a 256 color computer, the artwork will be dithered to a standard palette. Because of the slowness and limitations of dithering, all transitions will be disabled. Video for Windows can be used to improve the speed and quality of this dithering.

When you use a 24 bit picture format such as JPEG in a 256 color presentation, the bitmap takes significant time to be converted to 256 colors and if the current palette does not contain sufficient colors then the results will be undesirable.

Problems with Windows display drivers

WinG delivers the best performance for 256 color video modes. But WinG is not reliable on all video cards or computers, particularly cards with early S3 video drivers. The user will be prompted before WinG is used. WinG can also be enabled by setting [Graphics] FastMode = 1 and disabled by setting [Graphics] FastMode = -1 in the "formula.ini" file or "*project*.ini" file in the Windows directory.

WinG is recommended for a 256 color presentation running on a 256 color video mode under Windows 3.1. If WinG is not installed under these circumstances, then all fades and wipes will be disabled. If wipes were not disabled then the time taken to display some bitmaps may become unacceptable.

Video for Windows can give improved performance for all video modes. It allows a 256 color project to be shown on a 16 color video card, or a 24 bit color project to be shown on a 16 or 256 color video card. Video for Windows support can be provided on Windows 3.1 and Window NT by including "msvideo.dll" version 1.1a in the project directory. This DLL must not be used with Windows 95. A later version of Video for Windows is built into Windows 95.

Video for Windows causes problems on Windows NT and will be disabled for the 16 bit version of Formula Graphics. Video for Windows can be enabled by setting the [Graphics] VFWMode = 1 and disabled by setting [Graphics] VFWMode = -1 in the "formula.ini" file or "*project*.ini" file in the Windows directory.

When a project is being run using the "formula.exe" file, the project settings will be saved in the "formula.ini" file in the Windows directory. This file can be changed using Formula script or by any other application. This file can also be created with desired settings before Formula is ever executed. If the project is being run using a renamed "*project*.exe" file then the settings will be saved in "*project*.ini".

Mouse cursor

The position of any pixel in the graphics window can be given by the term **X,Y**. The **X** value is the pixel position across the screen and the **Y** value is the pixel position down the screen. The position of the pixel in the top left corner on the graphics window is 0,0.

As the mouse cursor moves across the graphics window, its position will be displayed on the status bar. If the left mouse button is clicked on a pixel, then the red, green, and blue components of the selected pixel color will be displayed on the status bar.

If the left mouse button is held down while the cursor is dragged across the graphics window, then a capture rectangle will appear over the selected area. The capture rectangle can be resized or moved around the graphics window. The capture rectangle will disappear when the left or right mouse button is clicked again.

A capture rectangle can also be moved around the screen by pressing the up, down, left and right arrow cursor keys.

Printing

An area can be selected with a capture rectangle before choosing **Print Window** from the Graphics menu. If no area was selected, then the entire contents of the graphics window will be sent to the printer.

A printer options dialog box will appear and you can choose the most suitable options before printing. If the printer is set to portrait mode then the image will appear at the top of the page with proportional dimensions. If the printer is set to landscape mode then the image will be stretched to fit the entire page.

The image in the graphics window can also be printed out during a running presentation by pressing **Ctrl+P** on the keyboard. The presentation must be waiting for input from the keyboard.

Frequently asked questions

Q1) Why are all of the elements disabled in the menu ?

Before you can add an element to a presentation, you must first open a screen. Select New Screen from the Project menu.

Q2) Does the order of the elements on the screen make a difference ?

When the screen is played, the elements will be displayed in the order they are listed.

Q3) How do I change the order of the elements ?

Click on the desired element name in the element list in the project window. Then press the up or down cursor keys to move the selected element up or down. More than one element can be selected. To delete an element, click the element name and press the delete key.

Q4) When I try to add a Picture element I get asked if I want to save over the top of my existing picture ?

If you want to add an existing picture then make sure that you do not have any areas of the screen selected. If you want to cut out an area of the screen and save it as a picture then select the area before choosing Picture from the element menu.

Q5) How do I move or resize a Picture element ?

You must first select the picture by double clicking on it. When an element is selected it will have a rectangle surrounding it. As the mouse moves over this rectangle the cursor will change. You can then drag the picture to a new position or resize it by dragging the edges.

Q6) I do not understand "Activate hot elements". How can I finish a screen?

When you create a new screen, the "Activate hot elements" option will be enabled in the screen properties window. When the screen is played, after all the elements have been displayed, the screen will wait for you to click the right mouse button before proceeding to the next screen. You can disable this option and replace it with an input element at the bottom of the element list. The input element can be set to "Key press or mouse click" so that any response from the user will proceed to the next screen.

Q7) How do I execute another application by pressing a button ?

The button will have an "If button clicked …" option. In this option type 'execute' and then follow with the name of the other program inside inverted commas. The option would look like: execute "*other program*".

Q8) When I try to load a picture element, it tells me "File not found" !

The 16 bit version of Formula Graphics Multimedia System does not support long file names, or long names for directories.

Q9) Why won't my transitions work ?

If you are running a 256 color presentation on a 256 color video mode under Windows 3.1 without WinG then transitions will be disabled because they may be too slow. If you run a 256 color presentation on a 16 color computer or a 24 bit color presentation on a 256 color computer then transitions will be disabled. If you are running 16 bit Formula Graphics under Windows NT then transitions will be disabled.

Q10) Why are my bitmaps so slow to display ?

JPEG images are always slow. If you are trying to display 24 bit artwork in a 256 color window then converting each pixel will be very slow. If you have resized a bitmap which is not a multiple of 4 pixels wide then it will display slower. Compressed hard drives are always slow. Networks are slow. Old computers are slow.

Q11) Why do I see nasty side effects when I try to undisplay an element ?

If you do not undisplay elements in the reverse order that you displayed them then you will see undesirable images. These were the images that were beneath each element before they were displayed.

Q12) Why can't I use the Format function on the editor menu ?

Formatting will only work if you are editing a formatted text file such as RTF or HTM.

Q13) Why can't I change the color of a hypertext element ?

The color information of a hypertext document is contained inside the RTF format. You need to change the hypertext color using the Formula Graphics editor or a word processor such as Word for Windows.

Q14) When I try to run a script I get an error. How can I quickly jump to the line with the error ?

If you double click on the error in the result window, the script with the error will be displayed and the line with the error will be highlighted.

Q15) How many screens can I have ? How many elements ?

There are no limits to the number of screens or elements you can use. The 16 bit version only allows 65000 characters in a script file. This limit can be overcome by using separate scripts.

Q16) When I run my presentation on a certain computer, the graphics window goes all over the place or the program crashes with a General Protection Fault !

There are some low cost video cards that have bugs in their video drivers. During normal operation of a computer you would not find these bugs. But multimedia presentations can place huge demands on video drivers and this can expose hidden bugs. You will need to obtain up to date video drivers for your video card.

Video bugs can also occur because of old or unreliable versions of Video for Windows or because WinG is not compatible with your video card or operating system.

Formula Graphics is a complex application which may have some unknown bugs. If you find a particular element or operation which causes a crash or other undesirable behavior, and if this problem is repeatable then please document the problem and contact Harrow Software. We can provide you with a bug fix within days.

Q17) I just accidentally destroyed my project FGX file. What will I do ?

In the project directory you will find a project backup file with the extension FBK. This will be a copy of the last good project file. Rename this file to FGX.

** And remember to save your project regularly and back it up to a separate disk as often as possible.*

Changes and new features

Version 5.6

Internet FTP
Active-X control
Improved printing
Other fixes

Version 5.7

Improved HTML element
Animation+Sound element
Soft properties in the language
Improved bitmap and animation conversion
16 FIF format support

Multimedia System

Overview

The following steps will give you some guidelines on using Formula Graphics Multimedia System to create a multimedia presentation.

Producing the artwork

1. Design the presentation. A storyboard should be developed which outlines the contents of the presentation on a screen by screen basis. Each screen would usually contain a heading, some text, a picture or two and maybe an animation.

2. Use a graphics package such as Photoshop to create the artwork. Start by designing some backgrounds which suit the theme of the presentation. Then design each screen of the presentation by laying out pictures and text on the backgrounds.

3. An animation package can be used to create 2D and 3D animations for the presentation. Animations are a great way to enhance a presentation and are particularly good for product demonstrations.

4. Sound and motion video can be recorded or captured from tape. Voice overs, background music and video sequences are an important part of any presentation.

Authoring with Formula Graphics

1. You can start a new project by selecting New Project from the Project menu. As it is opened you can choose the options for the project. These options include the resolution of your presentation window. The most presentations are 640 x 480 with 256 colors. If you can afford the luxury then you should choose 16 or 24 bits of color.

2. A project can be made up of any number of screens. After opening your first screen you can begin adding elements. The first element on any screen is usually a background. If you want to use a plain color or a color gradient then you can use a Rectangle as a background.

3. You can add a picture to the screen by selecting Picture from the Element menu and choosing a filename. You can also add a picture by loading the artwork using Load Artwork and selecting an area of the graphics window before choosing Picture. The selected area will be cut out and saved as the new picture. An optimum palette can be determined automatically.

The most efficient way to build a screen is to load the screen layout, subtract the background, and then cut out the remaining images as picture elements. As each picture is added to the presentation its subtracted area can be chosen to become transparent.

4. One advantage of using Formula Graphics is that each picture and animation can have a different palette. The dynamic palette management system continuously adjusts the palette as the presentation is playing so that every element has the colors it needs. You should try to make sure that each element uses only the necessary number of colors so that no more than 236 different colors are displayed at the same time.

6. AVI and FLC animations can be added directly to the screen as Video and Animation elements. By converting your animations to the VDO and WDO formats you can improve the performance and take advantage of the additional features these formats provide.

7. You can add interactive elements such as buttons to the screen. When the presentation is playing and after all the elements on the screen have been displayed, the system can wait for some response from the user before continuing. If the user clicks on a hot element such as a button, then that hot element might carry out any number of possible actions.

These actions may be to display or undisplay another element, to jump to another screen or exit the presentation. They may also be to execute a procedure in the project script.

8. A script written in Formula Graphics multimedia language can be included with a project. A script can be used to control almost any part of the project or operating system. Some elements need to use a script. Graph elements use a script to get the data they are supposed to display. Dialog elements use the script to set and get the contents of the control.

9. Any screen in the project can be opened by double clicking its name in the project window. Any element can be opened by double clicking its name. Any number of elements can be selected by shift dragging, control clicking or double clicking their positions in the graphics window. Selected elements can be dragged to new positions or fine tuned with the cursor keys.

10. A final presentation can then be prepared by archiving all of the artwork into one easily distributable file and a password can be given to secure the work. An installation utility can be written using the multimedia language. The "formula.exe" file can be renamed to become a standalone runtime player.

Projects

A new project can be opened by selecting **New** from the Project menu. A New Project dialog box will open and you can select the name of the new project and the directory that the new project will use for all its files. Several configuration windows will then be opened for you to select the options for your project.

Projects are divided into screens. The names of all the screens in a project will be listed in the project window. A new screen can be added to the project by selecting **New Screen** from the Project menu. As each new screen is added to the project, its name will be added to the screen list.

Screens are made up of elements. When a screen is open, its list of elements will appear in the project window. Elements are things like pictures, animations and sounds. As each new element is added to the screen, its name will appear on the element list.

A project can be played by selecting **Play Project** from the Project menu. When a project is being played, the screens in the screen list will be played one after the other until they are finished. As each screen is played, its elements will be displayed one at a time in the order they are listed.

The order of the screens and elements can changed by clicking the desired screen name in the screen list or element name in the element list and pressing the up or down cursor keys to move the item. A screen or element can be deleted from the list by clicking its name and pressing the delete key.

Starting a new project

To begin a project select **New** from the Project menu. After choosing some options you will be asked to open your first screen. A screen properties window will open and the screen name will be added to the project window's screen list. The graphics window will open with the size and color resolution given in the project graphics options.

As our first element, choose **Background** from the Element menu. A Load Background dialog box will open. The background bitmap we select will be displayed in the graphics window. A Background element window will open, and the background name will be added to the project window's element list.

Now we can add a picture by choosing **Picture** from the Element menu. A Load Picture dialog box will open. The picture bitmap we select will be displayed in the graphics window. A picture element window will open, and the picture name will be added to the project window's element list. The picture can be selected by double clicking on it in the graphics window. The selected picture can then be moved to a new position by dragging it with the mouse.

We can add a sound by choosing **Sound** from the Element menu. After selecting a sound from the Load Sound dialog box, a sound element window will open and the sound name will be added to the project window's element list. We can test the sound by pressing the Play button in the element window.

Finally, we can add an input by choosing **Input** from the Element menu. After entering an element name, an input element window will open and the input name will be added to the project window's element list. Select the "key press or mouse click" option in the input element window. Because we have added our own input to the end of the screen we should disable the "Activate hot elements" option in the screen properties window.

Now that we have put together our first screen, we can test it by choosing **Play Screen** from the Project menu. First the background will be displayed, then the picture, and then the sound will play. When the project gets to the input, it will stop and wait for the user to either click on a mouse button or press any key on the keyboard.

Project options

When **Options** is selected from the Project menu a project options window will open.

If a **password** is given in the project options then the project will be protected by that password. Only those who know the password will be able to edit the project file. The contents of the project file will be scrambled so that they cannot be viewed by any editor. A password will not stop you playing a project. You will always be able to play the project.

Some elements such as animations or sounds can take a long time to finish. You may not wish to wait that long. If the **Spacebar or Enter to stop a time consuming element** option is on then any element can be stopped by pressing the space bar or Enter key. The project will then continue with the next element.

if something goes wrong when a project is playing, for example, if a file cannot be found, then an error message will be shown and the project will stop playing. If the **Continue playing after an error has occurred** option is set then the project will continue playing after any error. It should be noted that some errors are unrecoverable and the player may crash if it continues.

If the **Show project screen by screen** option is set then the project will begin playing at the first screen. The project will then continue to play each screen one by one in the order they are listed. You also have the option of using the left and right cursor keys to step forward or backward through the screens. This is the normal way to play a presentation.

If the **Control the project from the script** option is set then the project script will be responsible for controlling the flow of the project. Screens and elements can be displayed and undisplayed using the script language. A wide range of instructions are available.

If you are using Formula Graphics as a helper application to a web browser, then you will need to set the **Remote directory** option. This should not be done for normal presentations because it prevents the presentation from being copied to another directory. This may be a local directory or an internet address.

For information on archiving and preloading, see Archiving and preloading.

Using a script

Any script written in Formula Graphics multimedia language can be specified as the project script. This script will use the same variable space as the screens and element in the project, so any variables that are used by elements for conditions can also be used by the script.

There are two options for using a script with a project. The easy option is just to specify a script name in the project options. When the project begins playing, the script will open. At any time during the playing of the project any procedure in the script can be called.

A procedure in the script might need to be called to carry out some complex set of instructions when a button is clicked. Or it might also be called by an Action element at the beginning of a screen to perform some initialization of variables.

The harder option is to specify that the script will take control of the project. In this case the script will be opened and begin executing at the first line. It will be the responsibility of the script to display and undisplay screens and elements.

When you use a script to display and undisplay elements, make sure that the conditions of those elements are set to **STATIC** so they will not be affected by changing conditions or by the updating of the screen. For more information, see Conditions.

For a full list of all the instructions which can be used in a script, select Language from the Window menu.

For an introduction to the language see Language Basics. For more information on controlling the project from the script, see Projects, screens and elements.

Graphics Options

When **Graphics options** is selected from the Project menu, a project graphics options window will open.

The title of the project can be given in the Window Title option. This title will be shown in the title bar of the project window. If the size of the project window is the same size as the video screen, then the project will be played on a full screen window with no title bar.

The size and color resolution of the project graphics window can be chosen. The default is a 640 x 480 project in 256 colors. These options also specify the resolution of the window which is opened when a screen name is double clicked in the project window.

For more details on size and color resolution, see Graphics Window.

The "Always on top" option will prevent any other application from being used while the project is playing. This feature is useful for multimedia kiosks and screen savers.

The "Cover desktop" option will cover any unused area of the desktop with a black border while playing the project in the centre of the screen.

A presentation mouse cursor can be selected from the list of cursors. Using the 32 bit version (under Windows 95 or NT 4.0) you can include a cursor file (*.cur) or an animated cursor file (*.ani) in the same directory as the Formula Graphics executable. You can then specify this filename as the cursor for the presentation.

Screens and elements

A screen or element can be selected by clicking its name in the project window. More than one screen or element can be selected by holding down the shift or control keys.

Screens and elements can be arranged in any order by selecting them and moving them up or down with the cursor keys. The order in which they are listed will be the order in which they are played. Selected screens and elements can also be copied and pasted using the Project menu.

If you double click on a screen name, the selected screen window will open and its elements will be listed in the project window and displayed in the graphics window. If you double click on an element name, the selected element properties dialog will open.

Most elements can also be selected by double clicking the element's position on the graphics window. More than one element can be selected by holding down the control key while clicking once on each element's position or by holding down the shift key and dragging a capture rectangle over an area of elements.

Screens and elements can be deleted from the project by selecting them and pressing the delete key.

Displaying and undisplaying

Elements can be displayed and undisplayed. When a picture element is displayed, the area of the screen under the picture is preserved. When the picture is undisplayed, the area under that picture is restored. It is important that elements are undisplayed in the reverse order that they were displayed, otherwise they may leave artifacts.

Elements can also be removed. When a picture element is removed, the preserved area under the picture will be discarded without being restored. When a background element is displayed over the top of other elements, these elements are removed rather than undisplayed so that the new background is not disturbed.

After a screen is finished, the elements on the screen may be undisplayed by choosing the "Undisplay screen elements when finished" option in the screen properties dialog. The current background will always remain displayed until it is replaced with another background.

Element names

When a new element is created the element must be given a name. In some cases the name will be taken from the file name related to the element. In other cases the name may be typed in as one or more words. In the second case, the default name is usually the type of element.

An element name can have any number of characters and any type of character can be used. Names are generally case insensitive. Lower case names are easier to read in the element list of the project window.

An element name can be changed at any time. If the name contains a file name then this should be the name of an existing file in the project directory. The element file name can be changed to specify an alternative path, but this is not recommended because that path may not exist on another computer.

You can add additional words to the end of an element name which is a file name. These words must be separated from the file name by a space. If there is more than one element on a screen which uses the same file then it may be necessary to add a number or an extra word to the end of the element name to highlight the difference between different elements.

Element positions

When most elements are double clicked in the graphics window, a capture rectangle will appear around them. They can then be resized or moved by dragging them into a new position. As an element is being dragged, its position or size will be displayed on the status bar. The position of an element can be fine tuned by pressing the up, down, left or right cursor keys.

Choosing **Position** from the Element menu will open a position dialog box. The position of the currently selected element will be displayed. The element can be moved by changing its position values and pressing the Move button. A new position can also be entered by selecting an area with the capture rectangle before choosing Position from the Element menu.

Any changes to the position of an element can be undone by choosing **Undo** from the Element menu.

Palette management

Formula Graphics dynamically manages the system palette while playing a 256 color project. Before an element with a palette is displayed, any colors used by that element are allocated to free color positions in the system palette. If two elements use the same color, they will share that palette position. If no free positions are available, the nearest color will be found.

As each element is displayed, its colors will be remapped to the system palette. In the case of bitmaps, only colors that are actually used in the image are allocated positions. In the case of animations, any colors that are not used in the animation should be set to black so they won't be allocated. After an element is undisplayed or removed, its palette positions will be freed.

Care should be taken to ensure that there are no more than 236 colors being used on the screen at one time. The other twenty colors are reserved by Windows. Formula Graphics provides excellent color optimization to reduce the number of colors in each element to the minimum possible number.

Screen options

A screen should have a name. A screen condition can be specified in the "If this is true …" box. As the project is playing, the screen will only be displayed if its condition is true.

A screen can have a function key associated with it. As the project is playing, if this function key is pressed at any time then the project will jump to this screen. The project can then jump back to the previous screen using a "backscreen" command.

Another screen name can be given in the "Next screen" option. After the current screen is finished the project will jump to this next screen. As an example the last screen of a project may be set to jump back to the first screen when it is finished.

If the "Activate hot elements" option is set, then after all the elements of a screen have been displayed the screen will wait for some response from the user. Buttons and other hot elements will become active and the only way to proceed to the next screen will be to click the right mouse button (if the option is enabled) or to click on a button with a "break" or "continue" command.

If the "Undisplay screen elements" option is set, then every element displayed by this screen except the current background will be undisplayed before continuing to the next screen. This option is not needed if the first element of the next screen is a background. Displaying a background on the next screen will usually give a smoother transition.

Advanced Screen options

Each screen can have its own script. Before the screen begins playing, the specified script will be loaded and compiled. While the screen is playing, any procedure in the script can be called. Any variable used by the elements on the screen can also be used by the script. Until this screen is finished, the project script

will not be available. When the screen is finished, the script will be closed and its variables will be forgotten.

Each screen can have its own archive file. Before the screen begins the specified archive file will be opened. While playing the screen, artwork will be searched for first in this archive file. If it is not found in this archive then the project archive will be searched. If the artwork is not found in any archive then the project directory will be searched.

Each screen can preload its own artwork. Before the screen begins the specified files will be loaded into memory. When the screens finishes, the screen's preloaded files will be removed from memory, and its archive file will be closed.

For more information on archiving and preloading, see Archiving and preloading.

Conditions

Each screen and element can have a condition. The condition will be specified in the "If this is true…" option of the properties window. The default condition is "TRUE". As the project is playing, each screen will only be displayed if its condition is true and each element on a screen will only be displayed if its condition is true.

A condition is just a value. If this value is equal to zero then the condition will be false. If this value is not equal to zero then the condition will be true. So a screen or element will only be displayed if the value specified in its "If this is true…" box is not equal to zero.

A condition value may be given in a variety of ways. It may be a number, it may be a constant (such as "TRUE" for 1 or "FALSE" for zero), or it may be a variable representing a number. In fact, any numerical expression can be given for a condition. The expression can be any combination of numbers and variables with mathematical functions and operators, numerical and string comparisons, and boolean operators.

Variables

A variable is created when a name is assigned with a value. From then on, that name can be used to represent that value, and at any time the variable's name can be reassigned with a new value. For example, "let x = 1" will assigned the value 1 to the variable "x".

A variable can be assigned with a value in response to the clicking of a button or a hot area, or after a certain element has been shown, or some other event. After a variable has been assigned a value, then that variable can be used to represent that value in the condition for a screen or element.

As an example, if the action given in the "If button clicked then …" box of a button element is "let x = 1", and the condition of some other element is "x" (if x is not equal to zero), then the second element will only be displayed after the button has been pressed.

If the variable name used in a condition has not been previously assigned a value, then it will be assumed to be equal to zero.

The value of a variable can be compared with any other value using the equals operator '=='. For example the expression "x == 1" will return 1 (or true) if "x" is equal to one, or 0 (or false) if "x" is not equal to 1. Other types of comparison can be used such as not equals '!=', greater than '>', less than '<', etc.

String variables

A string is a collection of characters that can form a word or a sentence. A string is usually given inside quotation marks, for example, "this is a string". Strings are most often used to indicate the names of things.

If a variable name is preceded by $, then it will be a string variable. A string variable is created when a variable name is assigned with a string, for example, let $a = "this is a string". The string variable $a can then be used to represent that string in any expression. If a string variable has not previously been assigned a string, then it will be taken to be an empty string "".

As an example, if the action given by an action element is: let $a = "some name", and the condition of some other element is: $a == "some other name", then the second element will only be displayed if the two names are the same.

Other operators can also be used to compare two strings together such as not equals '!=', case insensitive equals "~=", and case insensitive not equals "!~". Case insensitive means that it doesn't matter whether the string has upper case or lower case letters.

Global variables

Global variable names are preceded by a '%' sign. A global variable can be assigned a value or a string. For example, let %b = 10, or let %b = "some option". The global variable %b can then be used to represent that value or string in any expression.

When a global variable has been assigned a value or string, the assignment becomes permanent. It will be stored in a file and will be restored the next time the project is played. The variable may be used throughout the project in any screen, element, or script. Global variables are ideal for setting options.

If a global variable has not previously been assigned a value or a string, then it will be taken to be 0 or "".

Hot elements

After the elements of a screen have been displayed, the system would usually wait for some kind of response from the user. The system will wait for a response if the "Activate hot elements" option is set in the screen options, or if an Input element is found in the list of elements.

When a project is in the "Activate hot elements" mode, it tests to see if any buttons, hot areas or any other type of hot element are being clicked or activated in any other way. When a hot element is activated then some action can be carried out. This action might be to carry out the commands given in an "If button clicked then …" box.

The action taken by a hot element might be to change the value of a variable. If some other element uses that variable in its condition, then the condition of that element will suddenly change. So when the system is waiting for a response, and some hot element is activated, the system must check again the conditions of every element on the screen.

First each element which is currently visible will be checked, from the topmost visible element to the bottom. Any visible element whose condition is no longer true will be undisplayed. Then each element on the list of the current screen will be tested from the top of the list to the bottom. Any element whose condition has just become true will be displayed.

Using a script

If a project is using a script then the variables used in that script will be available to all the screens or elements to use in their conditions. The screens and elements will share a common data space with the script.

If you are using a script to display and undisplay elements, and you do not want to have these elements automatically being displayed and undisplayed as their conditions become true or false, then you can set the conditions of those elements to blank. Elements with blank conditions will not be effected by changing conditions.

Static conditions

If the condition of an element is set to "STATIC", then the element will not be controlled by the presentation system. The element will not be displayed when the screen is displayed. The element will not be undisplayed or redisplayed by the activation of any hot element.

An element with a static condition can only be displayed and undisplayed using an action command or by using an instruction in the project script. This can be useful for when you want to control an element from the script. If you are wondering why the element you want to display from the script keeps appearing or disappearing, try setting its condition to "STATIC".

If no condition is specified for an element, that is if the "If this is true …" option is empty, the condition will also be static.

Condition codes

Condition codes are a redundant feature. They are available only for backward compatibility.

There are 1000 condition codes and each one can be represented by a number between 1 and 1000. Each condition code has only two states, they can either be true or false. By setting condition codes to true or false, and by testing these codes, elements in a project can be displayed and undisplayed.

A condition code can be used in the condition of an element or screen using the code function "code (condition code)". If the specified condition code is true, then the element or screen will be displayed. If the condition code is given as a negative number, then the screen or element will only be displayed if the condition code is false.

For example, if an element has "code (20)" specified as its condition, then the element will only be displayed if the condition code 20 is true. If "code (-20)" is specified, then it will only be displayed if condition code 20 is false.

A condition code can be set to true by using a "set" instruction as an action. For example, "set 20" will set condition code 20 to true. Two or more condition codes can be set by separating the codes with commas. For example, "set 20,30" will set both 20 and 30 to true. A range of condition codes can be set to true by specifying the lower and upper codes separated by a forward slash. For example, "set 20/30" will set all codes between 20 and 30 inclusive to true.

A condition code can be set to false using a "set" instruction with a negative code value. For example, "set -20" will set 20 to false. Two or more condition codes can be set to false by separating the negative codes with commas. A range of condition codes can be set to false by specifying a negative value for the first code. For example, "set -20/30" will set all codes between 20 and 30 to false.

Action commands

Some elements have actions which will be carried out when the element is clicked or activated in some other way. For example a button element has an "If button clicked then …" box in which a set of commands can be given. When the user clicks on the button these commands will be carried out.

More than one command can be given. If there is more than one command then each one must be separated by semicolons and spaces. For example "command; command; command; …".

Variables

The **let** command may be used to assign a new value to a variable. Assigning a value to a variable can be useful for changing the condition of an element or passing a parameter to the project script. If a variable name hasn't been assigned before then a new variable will be created.

To assign a value to a variable, the command would be of the form "let variable = numerical expression", where the numerical expression may consist of a number, a variable, or any combination of numbers, variables, functions and operators.

Some examples would be "let x = 1" or "let x = x + y / 2".

The **let** command may also be used to assign a string to a variable. The command would be of the form "let $variable = string expression", where the string expression could be a quote, a string variable, a number (which will automatically be converted to a string), or a collection of quotes, string variables and numbers separated by commas which will be joined together to form a single string.

Some examples are "let $a = "some name" or "let $a = "Mr ", $name, " at location ", x".

The **let** command may also be used to assign a global variable. The command would be of the form "let %variable = numerical expression" or "let %variable = string expression".

See also Conditions.

Procedures

The **call** command can be used to execute any procedure in the project script. The command would be of the form "call *procedure_name*". One optional parameter can be passed to the procedure, this can be any numerical or string expression. The command would then be of the form "call *procedure_name*: *parameter*".

If you need to pass more than one parameter then use "let" statements to set the values of variables before you call the script. After the procedure has been executed the system will return to the element.

The **script** command can be used to execute any procedure in a script for which there is a handle in the project script. The command would be of the form "script *script_variable* call *procedure_name*". One optional parameter can be passed to the procedure, this can be any numerical or string expression. The command would then be of the form "script *script_variable* call *procedure_name*: *parameter*".

Screens

The **select** command can be used to select another screen. Once a screen has been selected, its elements can be displayed using a "display" command. The selected screen may be different from the currently displayed screen.

The **screen** command can be used to jump to another screen. If the screen name is more than one word then the screen name should be enclosed by inverted commas. As an example: screen "screen 2".

The **overlay** command can be used to display another screen over the top of the current one. After all the elements on the given screen has been displayed then the current action will be continued.

The **next** command can be used to jump to the next screen.

The **back** command can be used to jump back to the previous screen.

The **break** command can be used to end the current screen.

Elements

The **display** command can be used to display an element on the screen. If the element is already displayed, then it will be redisplayed. If the element name has an extension or uses more than one word then it should be enclosed by inverted commas. As an example: display "picture.gif". The name can also contain a wildcard '*' character.

The **undisplay** command can be used to undisplay an element from the screen. If the element name has an extension or uses more than one word then it should be enclosed by inverted commas. The name can also contain a wildcard '*' character. As an example the command undisplay "pict*" will undisplay all elements whose names begin with "pict".

The **undisplay all** command will undisplay all elements on the screen.

The **remove** command will remove an element from the screen without undisplaying it.

The **play** command can be used to continue playing an element that has been stopped or is ready to play. If the element name has an extension or uses more than one word then it should be enclosed by inverted commas.

The **stop** command can be used to stop an animation or other element from playing. If the element is not currently playing then the command will be ignored.

The **stop all** command will stop all elements from playing.

The **goto** command can be used to jump to another element on the current screen. If the element name has an extension or uses more than one word then it should be enclosed by inverted commas.

Control commands

The **click** command can be used to click a button element on. If the button has an 'on' state then it will be switched into the 'on' state.

The **unclick** command can be used to click a button element off. The button will be switched into the 'off' state.

The **continue** command can be used to continue past an "activate hot elements" input.

The **exit** command can be used to stop the project playing.

Special commands

The **popup** command can be used to display a short string in a small white window with a black border. The window will open near the position of the mouse cursor. When a mouse button or keyboard key is pressed, the window will disappear. As an example: popup "This will appear".

The **sound** command can be used to directly play a sound file. The sound file name should be enclosed by inverted commas. As an example: sound "example**.**wav". If the sound command is followed by a **wait** specifier then the system will wait until the sound is finished before executing the next command. As an example: sound "example**.**wav" wait.

The **execute** command can be used to execute any other application. The other program's file name should be enclosed by inverted commas. As an example: execute "example**.**exe". See also Operating system.

Condition codes

The **set** command may be used to set a single condition code, a number of condition codes, or a range of condition codes. As an example, "set 20" will set condition code 20 to true.

The **hit** command may be used to toggle a condition code. The code will be switched on, the screen will be updated, then the code will be switch off.

See also Conditions.

Using action commands

As an example, lets say you wanted to use a Button to display a picture, when the button is clicked, the picture will be displayed. First you will need to create a Button element and a Picture element. Then you will need to have an "Activate hot elements" input at the bottom of the screen.

The "Activate hot elements" option is on by default in the screen properties window. If this option is set, then after all the elements on the screen have been displayed, the system will wait for a response from the user before moving on to the next screen. While the system is waiting, the button will be active.

When the button is clicked, the commands given in the "If button clicked then …" option will be carried out. In this example we wish to display a picture element, so the command would be 'display "*picture name*"'.

Because we will be displaying the Picture element using commands, the "If this is true …" option of the picture element must be set to 'STATIC'. The picture can then be included anywhere in the element list because it will be ignored when all the elements are first displayed.

Using variables

An alternative method of displaying the picture would be to use variables. These have the advantage of allowing much more complex levels of interactivity. In this case the button command would be something like 'let show_picture = TRUE'. Where "show_picture" is just a normal variable name. And the "If this is true …" option of the picture element would be set to 'show_picture'.

When the button is pressed, the value of the variable 'show_picture' will become TRUE. After any event occurs, such as the pressing of a button, the condition of each element will be tested. The condition of the Picture element will have changed from false (zero) to true (non-zero) so the picture will be displayed.

When any event occurs, after the designated actions have been carried out, the currently visible elements are first checked to see if any of the conditions are no longer true. If a condition is not true then that element will be undisplayed. Then the elements on the currently selected screen are tested to see if any have just become true. Then these elements are displayed.

So if any later event causes the value of the variable 'show_picture' to become false, then the picture will be undisplayed.

Files and directories

The directory which contains the project file (*project*.fgx) and all the content files is called the project directory. When a presentation is running, unless a path is specified for the filename, the file will be expected to be found in the project directory.

Using filenames in a script

When a file name is given in an instruction it can be given as any string expression. If no path is specified then the file will be assumed to be in the project directory. Paths should be given in the form "*drive***:**\\ *directory**filename*" where a backslash is written as '\\' instead of '\'.

You can get the current project directory or you can set it to a new location using one of the following instructions.

$*directory* **= project directory**
set project directory "*directory*"

You can also get the directory of the Formula Graphics executable file.

$*directory* **= formula directory**

The current working directory is the directory currently opened for use by the operating system. Formula Graphics does not use the current working directory but it may be used by other applications.

$*directory* **= current directory**
set current directory "*directory*"

For more details see Files and Directories in the language section.

Archiving and preloading

Archiving

Choosing **Archives** from the Project menu will open the Project Archives dialog box. This box contains a list of file names. Any files in the project directory whose names appear on this list will be compiled into an archive file. Compiling all your artwork and scripts into archive files helps make distribution easier and more secure.

If the "Add all files from project" button is pressed, then every file name used in every element of the project will be added to the list. This usually accounts for every file in a project. Some files such as AVI which require operating system assistance to play will not be copied to the list. Any files whose names are referred to only in scripts will not be copied to the list.

The "Add files from existing archive" button can be used to copy all the file names from an existing archive onto the list.

Once some screens have been copied to the clipboard using "Copy screens" from the Project menu, every file name from those screens can then be added to the list using the "Paste screens from clipboard" button. This can be handy if you are using multiple archives. These archives can be divided between groups of screens.

Individual files can be selected from the project directory and added to the list of file names.

When the "Build archive" button is pressed, an archive file with the extension "fga" will be saved to the project directory. A project can have one primary archive file which is specified in the project options. This archive file will be opened when the project begins playing.

When the project is playing and it needs to load a file, it will first search through the archives for that file. If it is found in the archives it will be loaded from the archives, otherwise the file will be found in the project directory.

File types such as AVI and MIDI which require MCI playback should not be archived. These files will need to be distributed separately.

Scripts and text files are copied directly into an archive file. When an archive file is loaded into a binary editor, the scripts and other text files in it can be viewed. If you do not want your code to be seen then use the SXM mangled text file format for your scripts and text files.

Additional archive files can be opened and closed while the project is playing. Any number of archive files can be opened. The following instructions can be used in any script.

open archive "*archive_file*"
close archive "*archive_file*"

If the presentation is running from an archive, and if another archive file is opened, and if a particular file is located in both archives, then only the file in the second archive will be used.

Preloading

The speed of a presentation can be greatly increased by preloading some of the artwork. Preloading can take place before the start of the presentation, or before the start of any screen. Preloaded artwork is read from disk and kept in memory for the duration of the presentation or screen.

Preloading can use up significant amounts of memory, so it is important that you be very selective about which files you choose to preload. If the same background is used on every screen then the background

bitmap should be preloaded. If there are any other pictures or sounds that are used regularly throughout a presentation, then they should be preloaded.

Preloading options are found both in the Project control options and in the advanced options of the screen properties window.

The following instruction in the project script will also preload a file into memory.

preload "*file name*"

Once a file has been preloaded by this instruction it will remain in memory until the end of the presentation, or until it is unloaded by the following instruction. The path does not need to be specified.

unload "*file name*"

Distribution

This section covers the following topics:

* Running your project as a standalone application.
* Distributing your project - the list of files that should be included.
* Distributing an **unregistered** project.
* Storing your artwork in an archive file - to make distribution easier and more secure.
* Changing the project icon and user cursors.

Distribution

A project can be played as a standalone presentation in a window of its own. You can specify the name of any project on the Formula Graphics command line. If the project files are in a different directory then the full path of the project file must be given.

formula.exe *project.*fgx

If the "formula.exe" (16 bit) or "fgx32.exe" (32 bit) file is renamed to "*project*.exe", and if the project files are in the same directory, then the project can be played by simply executing the "*project*.exe" file with no other parameters.

For example, the "formula.exe" file can be renamed to "example.exe" and when this file is executed, the project called "example.fgx" will be played as a stand alone presentation. Only when Formula Graphics is called "formula.exe" or "fgx32.exe" will it be a project editor.

A final project can be distributed by including the following files:
project.exe - Formula Graphics (renamed from formula.exe or fgx32.exe)
formsync.dll - used for timing (only required for 16 bit formula.exe)
project.fgx - project file (remember to use a password)
project.fga - optional archive file containing all artwork
other files

All of the bitmaps, animations, scripts and other files used by your project can be compiled into archive files which will be included with your project. This can make distribution easier and more secure. Otherwise these files must be included individually. Files such as AVI, or MIDI which require MCI playback must always be included with the project.

It is highly recommended that Video for Windows be installed on a Windows 3.1 computer before running a presentation. A distributable Video for Windows installation kit is available from Microsoft. Windows 95 has Video for Windows already built in.

Registered projects do not show any banner or any other obvious sign that they were authored with Formula Graphics. For more information on registering Formula Graphics, choose Registration from the Help menu.

Unregistered distribution

Some features of Formula Graphics are not available to unregistered users. These disabled features are particularly related to distribution.

. Protecting your project with a password.
. Copying all your artwork into one or more archive files.

Unregistered projects must be distributed with the Formula Graphics help file (formula.hlp). Before the unregistered project starts playing, a Formula Graphics banner will appear for a few seconds.

User icons and cursors

The setup application provided with Formula Graphics allows a separate icon file to be distributed with the presentation. This icon can be assigned to play the presentation.

The main icon of Formula Graphics can be changed by editing the executable with a dialog editor such as the Microsoft Windows SDK dialog editor or Borland's resource workshop. The main icon is called "USER".

Using the 16 bit or 32 bit version of Formula Graphics, you can change the user cursors using a dialog editor. The user cursors are called "CUR_USER_1", " CUR_USER_2", etc.

Using the 32 bit version (under Windows 95 or NT 4.0) you can include a cursor file (*.cur) or an animated cursor file (*.ani) in the same directory as the Formula Graphics executable. You can then specify a cursor filename for the entire presentation or any of the hot elements.

Changing any other undocumented resources is in breach of the license agreement and is against the law.

Internationalization

There are other resources used by Formula Graphics presentations that can be changed to suit international distribution. These resources include the English language words contained in dialog boxes used for hypertext searching, "HYPERTEXT_SEARCH", "HYPERTEXT_TOPICS" and "HYPERTEXT_NOT_FOUND".

Installation

Installation utility

The example version of Formula Graphics includes a setup utility called "setup.exe". This utility can be tailored to set up any presentation. There are three text files included with the utility: "files.lst", "dialogs.lst" and "icons.lst". You can edit these three files using any text editor.

The "files.lst" file contains a list of all the files you wish to install. These files can be compressed using the Microsoft compression utility. Four fields are given for each file: source name, destination name, destination file size and the source disk label. The source and destination names may be different if the file is compressed.

```
source name, destination name, size in bytes, disk label
source name, destination name, size in bytes, disk label
...
```

The "dialogs.lst" file contains all of the messages given to the user. Each message uses one line. These messages must be in the correct order. This list includes the default destination directory of the installation.

The "icons.lst" file contains the name of each executable file to have an icon, its corresponding icon title and an optional icon file. The first line of this file specifies the application group name. The "USER" icon of "formula.exe" or "example.exe" can be changed using a dialog editor.

```
program group name
executable name, name to appear beneath icon {, optional icon file }
executable name, name to appear beneath icon {, optional icon file }
...
```

Using Formula Graphics for installation

Formula Graphics can be used as an installation program. The language has all the features needed to create a directory, copy and decompress files with version checking and to install an icon. The following example shows you how to do this.

```
files = new byte[5][16]                    // files to be copied
$files[0] = "example.exe"
$files[1] = "formsync.dll"
$files[2] = "example.fgx"
$files[3] = "example.fga"
$files[4] = "example.ico"
files_max = 5

$destination = getedit "destination"       // set the destination directory
new directory $destination
if !dir_exist $destination
    error "Could not create directory: ", $destination

total = 0                                  // check for enough disk space
for n = 0 to files_max
    if !file_exist $files[n]               // does file exist ?
        error "Could not find file: ", $files[n]
    total = total + file_length $files[n]
if total > disk_space $destination
    error "Not enough space on drive: ", $destination strcnt 2
```

```
for n = 0 to files_max                                  // copy files to destination directory
     $srce = project directory
     $srce = $srce,"\\",$files[n]
     $dest = $destination
     $dest = $dest,"\\",$files[n]
     if !file_exist $dest
          expand $srce to $dest                         // copy or expand files
     else if file_date $dest < file_date $srce    // check version
          expand $srce to $dest

$iconfile = project directory
$iconfile = $iconfile,"\\example.ico"
program manager "[CreateGroup(Example)]"            // create program group and icon
program manager "[ShowGroup(1)]"
program manager "[AddItem(",$destination,"\\example.exe,Example,",$iconfile,")]"
```

Screen savers

The 16 bit version of Formula Graphics can be used as a screen saver by simply renaming the "formula.exe" file to **formula.scr** and copying it into the Windows directory. After "formula.scr" has been copied to the Windows directory, the "Formula Graphics" screen saver will be available in your list of Windows screen savers.

The 32 bit version of Formula Graphics can be used as a screen saver by renaming the "fgx32.exe" file to **my screen saver name.scr** and copying it into the Windows directory. After the screen saver has been copied to the Windows directory, your chosen screen saver name will be available in your list of Windows screen savers.

You can select the Formula Graphics screen saver from your list of screen savers and press the configure button. You can then choose any Formula Graphics project to be shown by the screen saver. You can choose to end the project when the mouse is moved or a key is pressed, or when the end of the project is reached.

You can also choose a password. Password protection applies only to projects which can be ended by a mouse move or key press. If the project has a finish or if the Escape key is enabled then no password will be required to end the presentation.

If you are distributing your screen saver and you want the installation to be configured by the installation program, then you must add the following entry to the Windows "control.ini" file in the Windows directory.

```
[Screen Saver.Formula Graphics]
Project=drive:\path\project.fgx
Interrupt=1
```

An interrupt value of 1 will stop the screen saver if the mouse is moved or a key is pressed. An interrupt value of 0 will wait until the project has finished playing. The following lines of script can be executed in a Formula Graphics script to create this entry.

```
$scr_file = "control.ini"
$scr_section = "Screen Saver.Formula Graphics"
set profile string $scr_file; $scr_section; "Project" = "drive:\path\project.fgx"
set profile string $scr_file; $scr_section; "Interrupt" = 1
```

If you are using the 16 bit version of "formula.exe" then you will also need to copy "formsync.dll" into the Windows directory as well. If you have selected an unregistered project, then a Formula Graphics banner will appear before the screen saver begins.

Internet support

This section covers the following topics:

* Playing a project directly through the internet
* Internet optimization - caching, archiving and preloading
* Helper application, Netscape plugin, and Active-X control
* Security and other internet issues
* Programming the world wide web
* Programming email.

Direct internet playback

A Formula Graphics project can be played directly over the internet. A project on a remote server can be played from the editor by choosing 'Internet player' from the Project menu. Otherwise the project can be played alone by specifying the full internet path on the command line. The specified path must begin with "http:". An example of the command line to play a project across the internet would be:

formula.exe http://www.harrow.com.au/netplay.fgx

A copy of the Formula Graphics executable file must already be on a local disk drive. The project file "fgx" will be downloaded first, then the graphics window will open and the presentation will begin to play. Other files will be downloaded as they are needed. The presentation will play the same as it would from a local disk drive, but using a standard modem connection it will play slower.

Internet cache

After Formula Graphics downloads a file from a remote server, it saves a copy of the file to the local disk drive. A subdirectory called 'netcache' will be created from the directory which contains the Formula Graphics executable. The downloaded files will be saved as "c#.tmp".

If the same file is requested again in a later session then the date of the file in the cache will be compared to the date of the file on the server. If the dates are the same then the file will be loaded directly from the local cache.

If the file is not used again for five active sessions then it will be removed from the cache. An active session is when any project is run from a remote server and at least one file is downloaded.

If the cache does not have enough disk space then all cached files from previous sessions will be removed from the cache to make more space. If the cache directory cannot be opened or the disk has no space then the cache will be disabled.

The cache files can be deleted at any time to free up more hard drive space.

Cache instructions

The following instructions can be used to control the cache. The specified filenames do not need to contain paths. The first instruction can be used to make sure that the file with the specified name is never automatically removed from the cache, unless a new file with the same name is available on the server.

netcache retain "*filename*"

The following instruction will compare the date of the local file with the file on the server and if the dates are different then the file will be downloaded again and saved in the cache.

netcache update "*filename*"

The following instruction will remove the specified file from the cache.

netcache remove "*filename*"

The following instruction will clear the entire cache.

netcache remove all

Archives

The performance of a Formula Graphics presentation playing over the internet can be greatly improved using archive files. A single archive file can contain all of the artwork used in the presentation. The overhead involved in a single download is much less than for downloading each individual artwork file. If a project archive file is used, the entire file will be downloaded into the cache before the project begins playing.

A large presentation may contain many megabytes of material. A single archive file can take a long time to download. The person using the presentation may not see all the material they have downloaded. It is much more efficient to use a separate archive file for each screen. Screen archive files will be downloaded into the cache before each screen begins playing.

For more information on archiving, see Archiving and preloading.

Helper application

Formula Graphics can be set up as a helper application to a web browser such as Netscape or Internet Explorer. When a Formula Graphics project filename file is found embedded in a web page, or referenced with a link, then Formula Graphics will be loaded and will begin playing the presentation.

A project can be embedded using the tag:

<embed src="*project*.fgx">

or referenced with a link using the tags:

<a href="*project*.fgx"> click here <a>

You will need to set the project directory in the project options dialog to point to the remote location of the project. This is because after the browser has downloaded the project file, Formula Graphics will need to know where the content can be found.

The browser then needs to be set up to understand Formula Graphics file type. For example, to set up the Netscape browser, go to the Helpers section of the General Preferences dialog. Press the 'Create New Type' button. For the Mime Type enter 'application/formula'. Set the extension to 'fgx' and choose 'Launch the application'. Then locate Formula Graphics on your hard drive and press OK.

See also the section on server configuration.

Netscape plugin

Formula Graphics presentations can be embedded in a world wide web page played using Netscape Navigator. To play the presentation, the client must first download the Formula Graphics Netscape plugin. The plugin has all of the features available in a normal project. The presentation will play the same as it would if it was running from a local hard drive, except on a standard modem, it will play slower.

To install the Netscape plugin, all you need to do is to copy the file called NP32FG**.DLL into the Netscape plugin directory. This subdirectory can be found off the Netscape program directory. When Netscape starts up, it will search the plugins directory and install the Formula Graphics plugin. You can check this by selecting "About plugins" from the Help menu.

A presentation can be embedded in a HTML page using the following tag. If no height or width are specified then the presentation will use the full Netscape window client area.

<embed src="*project*.fgx" width="*width*" height="*height*">

You may also need to set the project directory in the project options dialog to point to the location of the project. It is important to note that only one presentation can be embedded on each page.

See also the section on server configuration.

Instructions

The following instruction can be used to change the current Netscape page. This instruction will direct Netscape to close the current page and open a new page at the given location.

netscape url "*location*"

This instruction is available to both the helper application and the plugin.

Server configuration

Before a Formula Graphics presentation can be played as a helper application or embedded in a web page, the internet web server must be configured to understand the Formula Graphics file type. Any type of server can be easily configured and your internet provider should be happy to do this for you.

As an example, the Netscape HTTP server needs to have the following line included in "ServerRoot/config/mime.types".

applications/formula fgx

As another example, the Microsoft Internet Server needs to have the following value included in the registry at "`HKEY_LOCAL_MACHINE \SYSTEM \CurrentControlSet \Services \InetInfo \ Parameters \MimeMap`"

application/formula **,** fgx **, ,** `5:REG_SZ:`

Active-X control

Formula Graphics presentations can be embedded in any OLE enabled document. This includes a world wide web page played using Microsoft Internet Explorer. To play the presentation, the client must use the Formula Graphics Active-X control. The control has all of the features available in a normal project. The presentation will play the same as it would if it was running from a local hard drive, except on a standard modem, it will play slower.

The Active-X control is called "axfgx.ocx".

A presentation object can be embedded in a HTML page using the following tags.

<OBJECT ID="FormulaGraphics1"
TYPE="application/x-oleobject"

CLASSID="clsid:1EBCC564-F2F9-11cf-940C-444553540000"
CODEBASE="ftp://www.harrow.com.au/pub/axfgx.cab#version=5,7,0,0"
HEIGHT=*height* WIDTH=*width*>
<PARAM NAME="Project" VALUE="*projectname*.fgx"></OBJECT>

The number 1EBCC564-F2F9-11cf-940C-444553540000 is a unique identifier for the Formula Graphics Active-X control. The version number should indicate the version used to develop the presentation. The height, width and name of the project will need to be specified in the required places.

The address given by the CODE tag refers to an address from which the plugin can be downloaded. It is preferred that you provide a download from your own site. The CAB file is Microsoft's answer to a ZIP file which will automatically be decompressed and installed.

Active-X controls need to be registered before they can be used. Internet Explorer will automatically register the control after it has been downloaded and decompressed. The control can also be registered manually using "regsvr32 /s /c *drive*:*path*\axfgx.ocx".

You may also need to set the project directory in the project options dialog to point to the location of the project. It is important to note that only one presentation can be embedded on each page.

Internet playback issues

If a presentation is playing from a remote server, and it attempts to copy, rename or delete any file on the local drive, or tries to send information back to the server, the user will be prompted for permission. If the user presses the "yes to all" button then they will not be prompted again during that session.

A presentation playing over the internet should have the "Continue playing after an error has occurred" option enabled in the project options. The player will report any communication problems but will continue to play the project regardless. It will also continue to play after the cancel button is pressed on a download.

Files such as AVI, or MIDI which require MCI playback cannot be played through the internet and must first be downloaded to a local drive.

Animation files such as VDO, WDO and FLC do not need to be download before they are played. They can be streamed through the internet while the presentation is playing. For the best performance, large animation files should not be included in an archive.

Programming HTTP, FTP and Email

Almost any instruction in the Formula Graphics language which takes a filename as a parameter will also take an internet path name. If a file name begins with "http://" then it will be expected to be found on the world wide web. If it begins with "ftp://" then it will be expected to be found on an ftp server.

A number of special instructions are available for dealing with the World wide web, FTP and Email. For more information on these instructions, see Internet programming.

Internet bitmaps and animations

Formula Graphics can be used to make transparent GIF's for world wide web pages. Enable the "GIF interlace" and "Choose transparency on every save" options in the Graphics menu Save options. Then simply load an image, save the image, and click on the area you wish to be transparent.

Keeping bitmap file sizes down to a minimum is an important part of HTML authoring. Using Formula Graphics you can minimize the size of bitmaps by reducing the number of colors to a minimum and filtering with the antidither filter to increase the compression ratio.

Formula Graphics can be used to convert to transparent GIF animations for World Wide Web pages. Use the animation conversion to utility to convert any animation format into a GIF animation file. Set the Netscape extension option to repeat the animation continuously on a Netscape browser. Choose delta compression to generate a performance optimized GIF animation.

When converting a GIF file, use a 65000 color video mode and a 256 color video window so that only the palette colors that are used are saved.

For more details on animation conversion, see Animations.

Multimedia Elements

Background

After selecting Background from the element menu, a Load Background properties window will open and a bitmap file can be selected as a background. The background will appear in the graphics window and a background element window will open with some options.

When a background is displayed, in most cases it will be displayed over the top of all other elements on the screen. These elements are no longer visible and will be removed after the background is displayed.

To prepare a background

If the number of colors used by the background needs to be reduced, then display the background in the graphics window using Load Bitmap, reduce the number colors using Optimize Palette, and save the bitmap using Save Bitmap. Then choose the optimized bitmap as a new background element.

Using backgrounds

Most presentations will have a background. The first element of the first screen will usually be a background. Once a background has been displayed it will stay displayed until the end of the presentation or until another background element replaces it. If you would like to use a plain color background instead of a bitmap then you should use a Rectangle element.

When one screen finishes and the next screen begins, you would normally want to remove all the old elements from the window before displaying any new elements. The previous screen options may specify that all elements are to undisplayed when the screen finishes.

Otherwise the first element of the next screen may be a background element. After the new background has covered the whole screen, the old elements will be quietly removed from the system. The only disadvantage to this method is with 256 color presentations, while the new background is being displayed, more colors will need be available at the same time.

See also Bitmaps and Palettes and Graphics Window.

Rectangle

An area can be selected with a capture rectangle before choosing Rectangle from the element menu. After specifying an element name, a Rectangle element window will open with some options.

If the Filled rectangle option is set, then the area of the rectangle will be filled with color. If the Horizontal gradient option is set, then the rectangle will be filled with a range of colors from the "top color" to the "bottom color". If the 50% translucent option is set, then only every second pixel of the rectangle's area will be filled and the rectangle will appear translucent.

If the Border option is set, then the rectangle will have a 3D border. This border can be used with a filled rectangle or it can be used as a border for another element. The border can have any pixel width and can appear as a raised bevel or a sunk bevel.

Using a rectangle as a background

If no area was selected before choosing Rectangle, then the Background option will be set. When this option is set, the rectangle will behave like a background element. When the rectangle is displayed, any other elements that were on the screen before it will be removed including the previous background.

Picture

After selecting Picture from the element menu, a Load Picture dialog box will open and a bitmap file can be selected. The picture will appear in top left corner of the graphics window and a picture element window will open with some options.

If an area was selected with a capture rectangle before choosing Picture, then a Capture Picture dialog box will open. After specifying a file name, the selected area of the graphics window will be captured and saved. This new bitmap will become the picture element.

After pressing the Choose Transparency button, the left mouse button can be clicked on any part of the graphics window to select a transparency color. You can cancel your selection by clicking the right mouse button.

If the Do not undisplay option is set then the image that was in the window before the picture was displayed will not be restored after the picture has been removed.

Moving and resizing a picture

A picture can be selected by double clicking on its position in the graphics window. A capture rectangle will appear around the picture to indicate its position. This rectangle can then be dragged to another position by holding down the left mouse button over the rectangle and moving the mouse.

A picture can be resized by dragging the edges of the capture rectangle to new positions. A resized picture will take a little extra time to display. Resized pictures will be displayed faster when they are a multiple of four pixels wide.

To prepare a picture

Design a screen layout using a paint program. This layout would usually consist of pictures and text placed on a background. It is best to keep a separate copy of the blank background. Display the screen layout in the graphics window using Load Bitmap.

Then use Subtract Bitmap to subtract the background bitmap, choose a subtract color that is different to any other color in the layout. The resulting image on the screen will be just the pictures. The background will be replaced by the subtract color.

The individual images in the layout can now be added to the presentation as picture elements. Place a capture rectangle over each image and select Picture from the Element menu. The selected areas will be captured and saved as new bitmaps and new picture elements will be added to the element list.

After all the pictures have been cut out, select Display All from the Presentation menu. Then for each picture element choose the subtract color as the transparency color.

Reducing colors

During a 256 color project only a limited number of colors can be displayed at the one time. Each picture should use the minimum number of colors possible for an acceptable quality image. Formula Graphics uses palette management, so each picture can have a different palette.

You can choose Optimize Palette from the Graphics menu to reduce the number of colors for the entire layout, and then save each bitmap using that palette, or you can optimize the palette for each bitmap as it is captured and saved.

See also Bitmaps and Palettes and Graphics Window.

Free Picture

After selecting Free Picture from the element menu, a Load Picture dialog box will open and a bitmap file can be selected. The picture will appear in top left corner of the graphics window and a free picture element window will open with some options.

If an area was selected with a capture rectangle before choosing Free Picture, then a Capture Picture dialog box will open. After specifying a file name, the selected area of the graphics window will be captured and saved. This new bitmap will become the free picture element.

After pressing the Choose Transparency button, the left mouse button can be clicked on any part of the graphics window to select a transparency color. You can cancel your selection by clicking the right mouse button.

Choose one of the buttons in the "Frame position" box and then drag the mouse cursor across the graphics window to change the characteristics of the image. Right click the mouse or left click on the same spot to change back to the normal cursor.

Text

An area should be selected with a capture rectangle before choosing Text from the element menu. After specifying an element name, a Text element window will open with some options.

The text to be displayed by the element can be given in the "Text" option. Any number of characters can be specified. If you need to display a text with formatting, use a hypertext element instead.

The font type and text color can be chosen. The text can be justified to the left, right or center. The text can have a drop shadow. You can set the color of the shadow and its offset in pixels.

Advanced techniques

After a text element has been displayed, its text can be changed executing the following instruction in the project script.

textbox "*element name*" **=** "*some text*"

Any element can be created dynamically from a script. In the following example we will create a new text element, then we will set the font type, text color and position of the element before we display it.

```
my_text = new element "Text"
$my_text.font_face = "Arial"
my_text.font_height = 20
element my_text set "text" color 255,255,255
element my_text position 100,40 size 200,30
my_text.text = "Set the text property of the element"
display element my_text
```

Animation

After selecting Animation from the Element menu, a Load Animation dialog box will open and an animation file can be selected. The first frame of the animation will appear in the graphics window and an animation element window will open with some options.

The default playback rate for the animation is 12 frames per second. If the "Skip frames" option is set then the animation will skip frames if necessary to achieve the required frame rate. Frame skipping is not recommended for delta animations because it will leave artifacts on the screen..

If the playback rate is set to zero then the animation will be played as fast as possible. The fastest possible playback rate will be a determined by the disk speed, the video driver speed and the file format used.

The animation can be set to display the first frame only. After the first frame has been displayed, the animation will then be ready to begin playing at any time by carrying out a 'play' command or executing a 'play' instruction in the project script.

There are two modes for playing an animation. The typical mode is that the animation will be played until it is finished before the system moves on to the next element. If the "Space bar or Enter key" option is set in the project options then the animation can be stopped at any time by pressing one of these keys.

The other mode will play the animation in the background while the system moves on and displays other elements or waits for input. At any time the animation can be stopped using a **stop** command or instruction. The animation can then be started again using a **play** command or instruction. The **playing** function can be used in any expression to determine if the animation is still playing.

A background animation can also be set to repeat itself continuously.

The animation can be given a group number that identifies it as one of a series of animations which will be joined together. When an animation with the same group number as a previous animation is played, the previous animation will removed from the screen before the next animation is displayed. The transition will be unnoticeable.

A list of key frames can be given in the "Display next elements at frames …" option. After each key frame is played, the next element in the list will be displayed before playing the next frame of the animation.

For example, if the key frames "10,20,30,40" are specified, then after tenth frame the first element after the animation will be displayed, after the twentieth frame the second element after the animation will be displayed, and so on until the fortieth frame. If the "Continue" option is set then the animation will keep playing until it is finished.

Animation file formats

When a 256 color animation is played in a 256 color project, its palette colors will be allocated positions in the system palette. Every color in its palette will be allocated, so any color positions not used by the animation should be set to black (0,0,0).

The number of colors used by an animation can be reduced by choosing Animation Convert from the Graphics menu. An optimum palette can be generated for the entire animation and then a new animation file can be created from the old one.

For best performance a 256 color animation should be converted to the VDO format. This format has high performance as well as palette remapping and several modes of transparency.

16 and 24 bit color animations can be converted to the WDO animation format which also has the

advantage of transparency.

Although AVI files can be played using this element, for the best performance and to take advantage of sound and other features of an AVI file you should use a Video element.

For more details on animation formats and animation conversion, see Animations.

Animation+Sound

After selecting Animation+Sound from the element menu, a Load Animation dialog box will open and an animation file can be selected. Then a Load Sound dialog box will open and a sound file can be selected. The first frame of the animation will appear in the graphics window and an Animation+Sound element window will open with some options.

The playback rate of the animation can be given, the default rate is 12 frames per second. The animation will be synchronized with the sound. The Synchronization option can be used to specify a time delay between the animation and the sound.

If the synchronization value is positive, then that number of animation frames will be played before the sound begins. If the specified value is negative, then that number of sound periods will play before the animation begins.

If the Skip option is set then the video will skip frames if necessary, to achieve the specified frame rate. Frame skipping does not look good with delta animations.

2D Sprite

After selecting 2D Sprite from the element menu, a Load Sprite dialog box will open and a bitmap or animation file can be selected. The sprite will appear in the graphics window and a sprite element window will open with some options.

If an area was selected with a capture rectangle before choosing Sprite, then a Capture Sprite dialog box will open. After specifying a file name, the selected area of the graphics window will be captured and saved. The new bitmap will become the sprite element.

After pressing the Choose Transparency button, the left mouse button can be clicked on any part of the graphics window to select a transparency color. You can cancel your selection by clicking the right mouse button.

Key Frames

Any number of key frames can be positioned around the graphics window. Each key frame can be moved, resized, warped or rotated around any axis. Choose one of the buttons in the "Key frame position" box and then drag the mouse cursor across the graphics window to change the size or position of the key frame image. Click the right mouse button or click on the same spot with the left mouse button to change back to the normal cursor.

You can select any key frame by clicking on it in the key frame list. The selected key frame will be displayed. The characteristics of that frame can then be modified. A new key frame can be inserted. The new frame will be inserted after the currently selected frame. A selected key frame can be deleted by pressing the delete key.

The key frames will only be used to guide the sprite around the graphics windows. The number of key frames should be kept to a manageable level but there must be enough of them to guide the sprite along the correct path.

Playback

The default playback rate for the sprite is 12 frames per second. If the playback rate is set to zero then the sprite animation will be played as fast as possible. The total number of frames should be set to a suitable value so that the movement of the sprite appears to be smooth.

The sprite can be set to display the first frame only. After the first frame has been displayed, the sprite will then be ready to begin playing at any time by carrying out a 'play' command or executing a 'play' instruction in the project script.

There are two modes for playing a sprite. The typical mode is that the sprite will be played until it is finished before the system moves on to the next element. If the "Space bar or Enter key" option is set in the project options then the sprite can be stopped at any time by pressing one of these keys.

The other mode will play the sprite in the background while the system moves on and displays other elements or waits for input. At any time the sprite can be stopped using a **stop** command or instruction. The sprite can then be started again using a **play** command or instruction. The **playing** function can be used in any expression to determine if the sprite is still playing.

A background sprite can also be set to repeat itself continuously.

If the specified sprite file is an animation file then a number of source animation frames can be specified. As the sprite is playing the source animation will play so that each new frame of the sprite shows a new frame of animation. After the source animation reaches the given number of frames it will begin again at

the first frame.

Converting to an animation

The sprite may not be able to play fast enough to achieve the desired frame rate. To solve this problem the sprite can be converted into an animation. When the 'Play as Animation' button is pressed a dialog box will appear with some options for rendering the sprite as an animation file.

Only the VDO format for 256 colors and WDO format for 16 or 24 bit colors are capable of being used for sprite animations. There are four possible modes for storing the animation. The default is 'Sprite' mode. This means that the animation contains only the moving image. No background information is stored, so the animation can be played over any background.

When the 'Render to Animation' button is pressed, the sprite will play. As it plays it will be recorded to the given animation file. After the animation has been recorded, a check box will be switched on beside the 'Play as Animation' button to indicate that the animation will be played instead of the sprite element. The animation can be disabled by switching off the check box.

* Sprites can be displayed much faster when their bitmaps are multiples of 4 pixels wide.

3D Sprite

After selecting 3D Sprite from the element menu, a Load 3D Model dialog box will open and a 3DS file can be selected. The bounding box of the object will appear in the graphics window and a 3D Sprite element window will open with some options.

'Autodesk 3D Studio' files are the most common format for 3D images on the PC. A huge collection of these files are available in the public domain. The image that appears in the graphics window will be the camera's view of the 3D world. The object will initially be positioned so that appears directly in front of the camera.

Key Frames

Any number of key frames can be positioned around the graphics window. Each key frame can be moved and rotated around in 3D space. Choose one of the buttons in the "Key frame position" box and then drag the mouse cursor across the graphics window to move or rotate the key frame. Click the right mouse button or click on the same spot with the left mouse button to change back to the normal cursor.

You can select any key frame by clicking on it in the key frame list. The selected key frame will be displayed. The characteristics of that frame can then be modified. A new key frame can be inserted. The new frame will be inserted after the currently selected frame. A selected key frame can be deleted by pressing the delete key.

The key frames will only be used to guide the 3D object around the graphics windows. The number of key frames should be kept to a manageable level but there must be enough of them to guide the object along the correct path.

Materials and lights

Most 3DS files contain material definitions such as color and texture mapping. But for those that don't, you can set the default material properties for the object. Diffusion indicates how the surface responds to normal light. Specularity indicates how the surface responds to reflected light. Shininess indicates how sharp the specular reflection is. And translucency makes the object see through.

The default lighting for the scene will be a light of normal intensity positioned on the camera. There will also be some ambient light. Any one of a number of other lighting schemes can be chosen and the intensity of the direct and ambient lights can be changed. The intensity of a light source can be greater than 100 percent.

Rendering

The are four modes for rendering an object. 'Bounding box' will show the volume of space that the object occupies. 'Wire frame will show the lines surrounding each polygon in the object. 'Quick render' will quickly render the object with shading and unshaded texture mapping (quality will be compromised for speed). 'Photo render' will show smooth polygon surfaces with full shading, texture mapping and translucency.

The Render button can be pressed at any time to render the current image using the chosen render mode. Most rendering modes will show the rendering status and the rendering of the object can be stopped by pressing the escape key.

The play button will render each frame of the object as it travels between key frames. Usually the render mode would be set to 'bounding box' and the play button would be used to see the outline of the object moving in real time.

Considerations should be made when rendering on a 256 color graphics window. The system does not allocate any additional colors for the new image, it only uses colors that are currently displayed in the system palette. You may need to display a suitable palette before rendering in 256 colors.

Playback

The default playback rate for the sprite is 12 frames per second. If the playback rate is set to zero then the sprite animation will be played as fast as possible. The total number of frames should be set to a suitable value so that the movement of the sprite appears to be smooth.

The sprite can be set to display the first frame only. After the first frame has been displayed, the sprite will then be ready to begin playing at any time by carrying out a 'play' command or executing a 'play' instruction in the project script.

There are two modes for playing a sprite. The typical mode is that the sprite will be played until it is finished before the system moves on to the next element. If the "Space bar or Enter key" option is set in the project options then the sprite can be stopped at any time by pressing one of these keys.

The other mode will play the sprite in the background while the system moves on and displays other elements or waits for input. At any time the animation can be stopped using a **stop** command or instruction. The animation can then be started again using a **play** command or instruction. The **playing** function can be used in any expression to determine if the animation is still playing.

A background animation can also be set to repeat itself continuously.

Converting to an animation

The sprite may not be able to play fast enough to achieve the desired frame rate. To solve this problem the sprite can be converted into an animation. When the 'Play as Animation' button is pressed a dialog box will appear with some options for rendering the sprite as an animation file.

There are four possible modes for storing the animation. The default is 'Sprite' mode. This means that the animation contains only the moving image. No background information is stored, so the animation can be played over any background. Only the VDO format for 256 colors and WDO format for 16 or 24 bit colors are capable of being used for sprite animations.

When the 'Render to Animation' button is pressed, the sprite will play. As it plays it will be recorded to the given animation file. After the animation has been recorded, a check box will be switched on beside the 'Play as Animation' button to indicate that the animation will be played instead of the sprite element. The animation can be disabled by switching off the check box.

Recommended method

Render the sprite animation as a 16 bit color WDO sprite animation on a 24 bit color graphics window. Rendering in 24 bits allows for the full range of colors. It may be necessary to first display a key frame color which is totally different from any colors used in the animation (using bitmap options in the Graphics menu)

Then for 256 color presentations, convert the WDO sprite animation to a VDO sprite animation. Use the animation optimize function first to reduce the number of colors used by the animation. The transparent areas of the sprite will appear in the key frame color. Set the transparency color to the same color (default is 0, 255, 0).

3D Title

After selecting 3D Title from the element menu, an element name can be chosen, then a 3D Title element window will open with some options. The characters used in the element name will be those used to form the 3D title.

The image that appears in the graphics window will be the camera's view of the 3D world. The element name will be converted into a 3D object. The object will initially be positioned so that appears directly in front of the camera. The bounding box of the object will appear in the graphics window.

Key Frames

Any number of key frames can be positioned around the graphics window. Each key frame can be moved and rotated around in 3D space. Choose one of the buttons in the "Key frame position" box and then drag the mouse cursor across the graphics window to move or rotate the key frame. Click the right mouse button or click on the same spot with the left mouse button to change back to the normal cursor.

You can select any key frame by clicking on it in the key frame list. The selected key frame will be displayed. The characteristics of that frame can then be modified. A new key frame can be inserted. The new frame will be inserted after the currently selected frame. A selected key frame can be deleted by pressing the delete key.

The key frames will only be used to guide the 3D object around the graphics windows. The number of key frames should be kept to a manageable level but there must be enough of them to guide the object along the correct path.

Fonts, materials and lights

Only true type fonts can be used for 3D titles. Any size font can be chosen. The font depth and bevel width can also be chosen. There are three different styles of beveling available. The character delay can be used to drop each character a few frames behind the last one.

The material button lets you change the visual attributes of the title. The color of the material can be specified. If a texture map is chosen then the colors of the texture bitmap will be mapped across the full face of the title.

Diffusion indicates how the surface responds to normal light. Specularity indicates how the surface responds to reflected light. Shininess indicates how sharp the specular reflection is. And translucency makes the object see through.

The default lighting for the scene will be a light of normal intensity positioned on the camera. There will also be some ambient light. Any one of a number of other lighting schemes can be chosen and the intensity of the direct and ambient lights can be changed. The intensity of a light source can be greater than 100 percent.

Rendering

The are four modes for rendering an object. 'Bounding box' will show the volume of space that the object occupies. 'Wire frame' will show the lines surrounding each polygon in the object. 'Quick render' will quickly render the object with shading and unshaded texture mapping (quality will be compromised for speed). 'Photo render' will show smooth polygon surfaces with full shading, texture mapping and translucency.

The Render button can be pressed at any time to render the current image using the chosen render mode. Most rendering modes will show the rendering status and the rendering of the object can be

stopped by pressing the escape key.

The play button will render each frame of the object as it travels between key frames. Usually the render mode would be set to 'bounding box' and the play button would be used to see the outline of the object moving in real time.

Considerations should be made when rendering on a 256 color graphics window. The system does not allocate any additional colors for the new image, it only uses colors that are currently displayed in the system palette. You may need to display a suitable palette before rendering in 256 colors.

Playback

The default playback rate for the sprite is 12 frames per second. If the playback rate is set to zero then the animation will be played as fast as possible. The total number of frames should be set to a suitable value so that the movement of the title appears to be smooth.

The title can be set to display the first frame only. After the first frame has been displayed, the title will then be ready to begin playing at any time by carrying out a 'play' command or executing a 'play' instruction in the project script.

There are two modes for playing an animation. The typical mode is that the animation will be played until it is finished before the system moves on to the next element. If the "Space bar or Enter key" option is set in the project options then the animation can be stopped at any time by pressing one of these keys.

The other mode will play the animation in the background while the system moves on and displays other elements or waits for input. At any time the animation can be stopped using a **stop** command or instruction. The animation can then be started again using a **play** command or instruction. The **playing** function can be used in any expression to determine if the animation is still playing.

A background animation can also be set to repeat itself continuously.

Converting to an animation

The title animation may not be able to play fast enough to achieve the desired frame rate. To solve this problem the title animation can be saved as an animation file. When the 'Play as Animation' button is pressed a dialog box will appear with some options for rendering the title as an animation file.

There are four possible modes for storing the animation. The default is 'Sprite' mode. This means that the animation contains only the moving image. No background information is stored, so the animation can be played over any background. Only the VDO format for 256 colors and WDO format for 16 or 24 bit colors are capable of being used for sprite animations.

When the 'Render to Animation' button is pressed, the 3D title will play. As it plays it will be recorded to the given animation file. After the animation has been recorded, a check box will be switched on beside the 'Play as Animation' button to indicate that the animation will be played instead of the 3D title element. The animation can be disabled by switching off the check box.

Recommended method

Render the sprite animation as a 16 bit color WDO sprite animation on a 24 bit color graphics window. Rendering in 24 bits allows for the full range of colors. It may be necessary to first display a key frame color which is totally different from any colors used in the animation (using bitmap options in the Graphics menu)

Then for 256 color presentations, convert the WDO sprite animation to a VDO sprite animation. Use the animation optimize function first to reduce the number of colors used by the animation. The transparent

areas of the sprite will appear in the key frame color. Set the transparency color to the same color (default is 0, 255, 0).

Video

An area should be selected with a capture rectangle before choosing Video from the element menu. A Load Video dialog box will open and an AVI video file can be selected. The first frame of the video will appear in the graphics window and a video element window will open with some options.

The video file must be an AVI file or some other format supported by Microsoft Windows MCI (Media control interface). MCI players are also available for Apple Quicktime (MOV) and Motion picture group (MPEG) videos. Video files usually contain a sound track that will be played synchronously with the animation.

The video can be played at its normal resolution, it can be stretched across any area of the graphics window, or it can be played full screen. When the video is played full screen the video mode will be switched to low resolution while the video plays.

The video can be set to display the first frame only. After the first frame has been displayed, the video will then be ready to begin playing at any time by carrying out a 'play' command or executing a 'play' instruction in the project script.

There are two modes for playing a video. The typical mode is that the video will be played until it is finished before the system moves on to the next element. If the "Space bar or Enter key" option is set in the project options then the video can be stopped at any time by pressing one of these keys.

In the other mode the video will play in the background while the system moves on and displays other elements.

At any time the video can be paused using the **stop** command or instruction. It can be started again using the **play** command or instruction. The **playing** function can be used in any expression to determine if the video is still playing.

An action can be given in the "When finished …" option. When the video is finished the specified action will be carried out. The presentation system must be waiting in "Activate hot elements" mode for the end of the video to be detected. The "playing" function can also be used to find out if the video is currently playing.

Before you can play a video file, you must first make sure that the correct MCI drivers have been installed. If the drivers are not properly installed then the video will not play. Windows 95 already has drivers for AVI files. If you are using advanced compression for your AVI file then your decompression drivers must be installed.

Sound

After selecting Sound from the element menu, a Load Sound dialog box will open and a sound file can be selected. A sound element window will open with some options.

A sound card must be installed in the system for this element to play. If no sound player exists then the element will be ignored. The SOUND_DEVICE variable can be used to detect for a sound card.

A sound can be played in one of two ways. If the "Start sound and continue" option is set then after the sound starts to play the project will move on to the next element. The sound will continue to play in the background until it is finished.

If the "Play sound until finished" option is set then the sound will play until it is finished before advancing to the next element. When this mode is chosen you can use the "Display next elements at these times …" option. Any number of times (in seconds) can be given. As the sound is playing, after any of the specified times have elapsed, the next element in the list will be displayed.

As an example, if the times "2,4,6,8" were specified, then after two seconds of sound the first element after the sound element would be displayed, after four seconds the second element would be displayed, and so on until after the eighth second the fourth element would be displayed and the sound would continue to play until finished.

An action can be given in the "When finished …" option. When the sound is finished the specified action will be carried out. The presentation system must be waiting in "Activate hot elements" mode for the end of the sound to be detected.

The sound can be paused using the **stop** command or instruction. It can be started again using the **play** command or instruction. The **playing** function can be used in in any expression to determine if the sound is still playing.

When the "Play across screens" option is enabled, the Sound element will remain an active element until it is specifically undisplayed using an "undisplay element" command or instruction. When the "Repeat continuously" option is enabled, the sound file will be played over and over again.

Midi

After selecting Midi from the element menu, a Load Midi dialog box will open and a Midi sound file can be selected. Then a Midi element window will open with some options.

A MIDI device must be installed on the system for this element to play. Most sound cards have MIDI devices built in. If no MIDI player exists then the element will be ignored. The MIDI_DEVICE variable can be used to detect for a MIDI device.

If the Wait option is set then the system will wait for the MIDI file to finish before moving on to the next element.

An action can be given in the "When finished …" option. When the MIDI is finished the specified action will be carried out. The presentation system must be waiting in "Activate hot elements" mode for the end of the MIDI to be detected.

The music can be paused using the **stop** command or instruction. It can be started again using the **play** command or instruction. The **playing** function can be used in any expression to determine if the music is still playing.

When the "Play across screens" option is enabled, the MIDI element will remain an active element until it is specifically undisplayed using an "undisplay element" command or instruction. When the "Repeat continuously" option is enabled, the MIDI file will be played over and over again.

Advanced techniques

By enabling the "Play across screens" and "Repeat continuously" options, a MIDI file can be made to play continuously throughout the presentation. But what if you wish to play different MIDI files, one after the other, throughout the presentation? There are a number of ways you could do this, but we will show you the easiest way.

You can dynamically create a MIDI element at run time using the project script. The MIDI element must be given a filename and some options. The "no_screen" property will be the same as setting the "Play across screens" option. The "finish_action" property will be the action which is taken when the MIDI finishes playing.

```
init_midi:                                  // initialize a few variables
    midi_index = 0
    midi_files = new byte[3][16]            // allocate space for 3 file names
    $midi_files[0] = "vivaldi.mid"; "canyon.mid"; "ballade.mid"
    exist_flag = FALSE                      // no current element

play_midi:                                  // start playing a MIDI file
    if exist_flag                           // if the element exists then undisplay it
        undisplay element my_midi

    // get the new element name
    $element_name = $midi_files[midi_index]
    midi_index++
    if midi_index > 2 then midi_index = 0

    my_midi = new element "MIDI"            // allocate a new element
    $my_midi.name = $element_name
    $my_midi.finish_action = "call play_midi"
    my_midi.no_screen = TRUE
    display element my_midi                         // play the new element
```

exist_flag = TRUE

When each midi file is finished, the finish action will call the procedure again and play the next midi. The midis will be cycled around to play continuously. The sound can be stopped at any time using "undisplay my_midi". The same technique can also be applied to other sound and video elements.

CD Audio

After selecting CD Audio from the element menu and specifying a name, a CD Audio element window will open. This element will play a number of tracks from a music CD. The first and last tracks can be specified.

The music can be paused using the **stop** command or instruction. It can be started again using the **play** command or instruction. The **playing** function can be used in any expression to determine if the music is still playing.

When the "Play across screens" option is enabled, the CD Audio element will remain an active element until it is specifically undisplayed using an "undisplay element" command or instruction. When the "Repeat continuously" option is enabled, the specified audio tracks will be played over and over again.

A CDROM must be installed on the system and a music CD must be inserted for this element to play. If no playable CD exists then the element will be ignored. When the element is undisplayed, the music will stop.

Text Button

An area should be selected with a capture rectangle before choosing Text Button from the element menu. After specifying an element name a Text Button properties window will open. The text button may be resized or dragged to a new position on the screen.

There are three types of button. The first type has an up and a down state. The button will go down for as long as the left mouse button is held down over it. Then when the button is released the commands given in the "If button clicked then …" option will be carried out.

The second type of button will carry out the "If button clicked then …" commands when the button is clicked on. The button will stay on until it is clicked off again and the "If clicked off …" commands will be carried out. If the "… then button 'on'" condition is true when the button is first displayed then the button will begin in the 'on' state.

The third type of button can be one of a group of buttons. When one button in the group is switched 'on' then all others in the group will be switched 'off'. As each button is switched on and off it will carry out the respective commands. These buttons must have a group number, which can be any value. All buttons with the same group number will belong to the same group.

Normally the button will have the specified face color and display the text given by the 'Button text' option. But when the button is in the 'on' state it will have the specified 'On color' and display the text given by the 'On text' option.

Using a Text Button

As an example, lets say you wanted to use a Text Button to display a picture, when the button is clicked, the picture will be displayed. First you will need to create a Text Button element and a Picture element. Then you will need to have an "Activate hot elements" input at the bottom of the screen.

The "Activate hot elements" option is on by default in the screen options. If this option is set, then after all the elements on the screen have been displayed, the system will wait for some response from the user before moving on to the next screen. While the system is waiting, the text button will be active.

When the button is clicked, the commands given in the "If button clicked then …" option will be carried out. In this example we wish to display a picture element, so the command would be 'display "*picture name*"'.

Because we will be displaying the Picture element using commands, the "If this is true …" option of the picture element must be set to 'STATIC'. The picture can then be included anywhere in the element list because it will be ignored when all the elements are first displayed.

Using variables

An alternative method of displaying the picture would be to use variables. These have the advantage of allowing much more complex levels of interactivity. In this case the button command would be something like 'let show_picture = TRUE'. Where "show_picture" is just a plain old variable name. And the "If this is true …" option of the picture element would be set to 'show_picture'.

When the button is pressed, the value of the variable 'show_picture' will become TRUE. Then because an event has occurred, the condition of each element will be tested. The condition of the Picture element will have changed from FALSE (zero) to TRUE (non-zero) so therefore the picture will be displayed.

See also Conditions and Action commands.

Picture Button

After selecting Picture Button from the element menu, a Load Picture Button dialog box will open and a bitmap file can be selected. This bitmap will be used to display the button when it is in a normal state. After the button has been displayed, an element window will open with some options.

If an area was selected with a capture rectangle before choosing Picture Button from the element menu, then a Capture Button dialog box will open. After choosing a file name, the selected area of the graphics window will be captured and saved. This new bitmap will become the normal state of the button.

Three other bitmaps can be specified. The first bitmap will be the one that is shown when the cursor passes over the button area. The second will be shown when the left mouse button is down over the button area. The third will be shown when the button has been clicked into an 'on' state. These bitmaps may be selected from the project directory (o) or captured from the button's area in the graphics window and saved to disk (x).

Button types

There are three types of button. The first type simply has an up and a down state and will only stay down for as long as the left mouse button is down over it. Then when the button is released the commands given in the "If button clicked …" option will be carried out.

The second type of button will carry out the "If button clicked then …" commands when the button is clicked on. The button will stay on until it is clicked off again and the "If clicked off …" commands will be carried out. If the "… then button 'on'" condition is true when the button is first displayed then the button will begin in the 'on' state.

The third type of button can be one of a group of buttons. When one button in the group is switched 'on' then all others in the group will be switched 'off'. As each button is switched on and off it will carry out the respective commands. These buttons must have a group number, which can be any value. All buttons with the same group number will belong to the same group.

Preparing picture buttons

Design at least two screen layouts. The first with all the buttons in the normal state and the second with all the buttons in the down state. Display the first screen layout in the graphics window. Then one by one place capture rectangles over the individual buttons and select Picture Button.

Then display the down state layout in the graphics window. For each button element press the down state capture button (x). The down state bitmaps will be captured and saved using the same area as their normal state bitmaps.

The same process can be carried out for the 'cursor' and 'on' states of the button.

See also Bitmaps and Palettes and Graphics Window.

Using a Picture Button

As an example, lets say you wanted to use a Picture Button to display some text, when the button is clicked, the text will be displayed. First you will need to create a Picture Button element and a Text element. Then you will need to have an "Activate hot elements" input at the bottom of the screen.

The "Activate hot elements" option is on by default in the screen options. If this option is set, then after all the elements on the screen have been displayed, the system will wait for some response from the user before moving on to the next screen. While the system is waiting, our button will be active.

When the button is clicked, the commands given in the "If button clicked then …" option will be carried out. In this example we wish to display a picture element, so the command would be 'display "*text name*"'.

Because we will be displaying the Text element using commands, the "If this is true …" option of the text element must be set to 'STATIC'. The picture can then be included anywhere in the element list because it will be ignored when all the elements are first displayed.

Using variables

An alternative method of displaying the text would be to use variables. These have the advantage of allowing much more complex levels of interactivity. In this case the button command would be something like 'let show_text = TRUE'. Where "show_text" is just a plain old variable name. And the "If this is true …" option of the text element would be set to 'show_text'.

When the button is pressed, the value of the variable 'show_text' will become TRUE. Then because an event has occurred, the condition of each element will be tested. The condition of the Text element will have changed from FALSE (zero) to TRUE (non-zero) so therefore the text will be displayed.

See also Conditions and Action commands.

Mouse cursor

A different bitmap can be specified to be shown when the mouse cursor moves over the picture button. A different mouse cursor can also be specified. The new mouse cursor can be chosen from a list of cursors. A number of user cursors are also available which can be modified in the Formula Graphics executable using a dialog editor.

Using the 32 bit version (under Windows 95 or NT 4.0) you can also include a cursor file (*.cur) or an animated cursor file (*.ani) in the same directory as the Formula Graphics executable. You can then specify this filename as the cursor for the button.

Picture Slider

After selecting Picture Slider from the element menu, a Load Slider Base dialog box will open and the bitmap for the base of the slider can be selected. Then a Load Slider Knob dialog box will open and the bitmap for the sliding knob can be selected. After the slider has been displayed, an element window will open with some options.

If an area was selected with a capture rectangle before choosing Picture Slider from the element menu, then a Capture Slider Base dialog box will open. After choosing a file name, the selected area of the graphics window will be captured and saved as the base of the slider. Then a Load Slider Knob dialog box will open.

After pressing the Choose Transparency button, the left mouse button can be clicked on any part of the graphics window to select a transparency color. You can cancel your selection by clicking the right mouse button.

The slider may be horizontal or vertical. The slider knob bitmap will initially be positioned in the middle of the base bitmap at the zero position. The slider can be moved by holding the left mouse button over the knob and dragging it to a new position.

If a procedure name is given in the "Call on … change" option then the specified procedure will be called when the slider is moved. The procedure will be passed two parameters. The first will be the name of the element and the second will be the new position of the slider as a value between zero and one.

The following instruction can be used to set the position of the slider. The knob bitmap will be displayed at the new position. The specified value must be between zero and one.

slider "element name" **position** *value*

Hot Area

An area should be selected with a capture rectangle before choosing Hot Area from the element menu. After specifying an element name, a Hot Area element window will open.

If the mouse cursor goes over the hot area then the commands given in the "If cursor over" option will be carried out and the specified cursor will be shown. When the left mouse button is clicked over the hot area then the commands given in the "If area clicked" option will be carried out.

See also Action commands.

Mouse cursor

A different mouse cursor can be specified to be shown when the mouse cursor moves over the picture button. The new mouse cursor can be chosen from a list of cursors. A number of user cursors are also available which can be modified in the Formula Graphics executable using a dialog editor.

Using the 32 bit version (under Windows 95 or NT 4.0) you can also include a cursor file (*.cur) or an animated cursor file (*.ani) in the same directory as the Formula Graphics executable. You can then specify this filename as the cursor for the hot area.

Hot Color

After selecting Hot Color from the element menu and specifying a name, a Hot Color element window will open. After pressing the Choose button, any color in the graphics window may be chosen. When the left mouse button is clicked over the selected hot color then the commands given in the "If color clicked" option will be carried out.

See also Action commands.

Edit Box

An area should be selected with a capture rectangle before choosing Edit Box from the element menu. After specifying an element name, an edit box will be displayed and an Edit Box element window will open with some options.

The text in the edit box may be "numerical", "upper case" or "read only". The edit box may be used for a single line, or it can be used for multiple lines with optional word wrap and a vertical scroll bar. The font type, text color and background color can also be chosen.

To copy text into an edit box, use the following instruction in the project script. Any length of text can be sent to the edit box.

editbox "*element name*" **=** "*string expression*"

To get the text from an edit box you can use the **getedit** "*element name*" function in any string expression. This function will return the contents of the edit box.

Notification

A procedure name can be given in the "Call on …" option. This procedure will be called when it is time to verify the contents of the edit box. The procedure will be passed two parameters. The first will be the name of the element and the second will be the contents of the edit box.

```
my_procedure: my_element, my_string
    message "The name of the Edit Box is ", $my_element
    message "The contents of the Edit Box are ", $my_string
     …
```

This procedure can be called after input focus has moved away from the edit box, or if the enter key is pressed, or if the edit box is undisplayed.

Advanced techniques

The following example will show you how to load a text file from disk and copy the contents of the file into an Edit Box. We begin by loading the raw text file into a byte array.

```
load "example.txt" byte my_text
```

Then we copy the text into our edit box by treating the byte array as a string variable. The element "my element" must have been displayed already. A warning will be generated if it hasn't.

```
textbox "my element" = $my_text
```

An element can be created dynamically by a script. In the following example a new edit box element will be created. The position of the element will be set before the element is displayed.

```
my_editor = new element "Edit Box"
element my_editor position 100,40 size 200,30
display element my_editor
```

Edit boxes have other properties which are not available in the properties window. These are "right" (right align) "lower_case", "password" (asterisks) and "limit" (limit the number of characters that can be entered). These values can be permanently set by editing the project file in a text editor.

Note: The colors of an edit box will always be black and white when using the 32 bit version of Formula Graphics on a 256 color

video mode. Blame Microsoft.

List Box

An area should be selected with a capture rectangle before choosing List Box from the element menu. After specifying an element name, a list box will be displayed and a list box element window will open with some options.

The font type, text color and background color can be chosen. The list box can have a thin black border. The scrollbar can be hidden when the list is less than one page long. The list box can have multiple columns and can take multiple selections. The contents of the list box can be sorted in alphabetical order.

If a procedure name is given in the "Call …on …" option, then when an item is clicked or double clicked the procedure will be called in the project script. The procedure will be passed three parameters, the first will be the name of the element, the second will be the string that was selected and the third will be its position in the list.

```
my_procedure: my_element, my_string, my_position
    message "The name of the List Box is ", $my_element
    message "The selection from the List Box is ", $my_string
    message "The position of the selection is ",my_position
     …
```

Advanced techniques

A number of instructions can be used with list boxes.

The following instructions can be used to add new items to the list, to set the selection state of any item, to test the state of a list box or to remove every item from the list.

The following instruction will add a new item to the bottom of a list box. If an optional position variable is given then the position of the new item in the list will be assigned to the variable.

{ *position* **=** } **listbox** *element* **add** "*item*"

The selection state of any item in the list can be changed using the following instruction. The given item name can be a whole word, or just a matching prefix. If no mode value is given or if the value is not equal to zero then the item will be selected. If the mode value is zero then the item will be deselected.

listbox *element*" **select** "*item*" (**mode** *state*)

The selection state of an item at a given position can be changed using the following instruction. If no mode value is given or if the value is not equal to zero then the item will be selected. If the mode value is zero then the item will be deselected.

listbox "*element*" **select position** *position* (**mode** *state*)

The selection state of any item can be found using the following instruction. If the item at the given position is selected then the selection state variable will be assigned with one, otherwise it will be assigned with zero.

selection state **= listbox** "*element*" **position** *position*

A list box loop can be used to find every selected item on a multiple selection list. The count variable will be incremented as the selected strings are returned.

$*item* **= listbox** "*element*" **loop** *count variable*
    …

…

Each item in a list can have more than one field by using tabs to separate the fields. The following statement can be used to set the tab alignment positions, in pixels, for list box.

listbox "*element*" **tabs** *position 1*, *position 2*, **…**

The following instruction will remove all items from a list box.

listbox "*element*" **reset**

Example

You can give a list of items in the element properties or you can fill the list with items using the script. The following example shows how to fill a list box with items. The list box must have been displayed first.

```
for n = 0 to 9
    listbox "my listbox" add $item[n]
```

Another example shows how to display more than one column of information in a list box by using tab alignments.

```
listbox "my listbox" tabs 20, 50          // each tab will be aligned to these positions.
for n = 0 to 9
    listbox "my listbox" add $field1[n], "\t", $field2[n], "\t", $field3[n]
```

Combo Box

An area should be selected with a capture rectangle before choosing Combo Box from the element menu. After specifying an element name, a combo box will be displayed and a combo box element window will open with some options.

There are three different types of combo box. The first is a "Simple" combo into which you can enter your choice and the list box is always visible. The second is a "Drop down" combo into which you can enter your choice and the list box will appear when you press the button. The third type is a "Drop down list" from which a selection can only be made by pressing the button and choosing from the list.

The font type can be chosen. The scrollbar can be hidden. The items in the combo list can be sorted in alphabetical order. You can specify a list of items in the element properties or you can fill the combo list with items from the script.

If a procedure name is given in the "Call …on …" option, then after an entry is made or an item is selected the procedure will be called in the project script. The procedure will be passed two parameters, the first will be the name of the element and the second will be the entry that was typed or selected.

```
my_procedure: my_element, my_string
    message "The name of the Combo Box is ", $my_element
    message "The entry from the Combo Box is ", $my_string
     ...
```

A number of instructions can be used with combo boxes. The following instructions can be used to set the contents of the combo edit box, to add a new item to the combo list or to remove all items from the combo list.

combobox "*element*" **=** "*entry*"
combobox "*element*" **add** "*item*"
combobox "*element*" **reset**

The function **getcombo** "*element*" can be used in any string expression to get the contents of the combo edit box.

Menu Box

An area should be selected with a capture rectangle before choosing Menu Box from the element menu. After specifying an element name, an empty menu will be displayed and a menu box element window will open with some options.

The default condition of a menu box is "STATIC". This is because the menu would normally only appear after a hot area or some other element had been clicked. The hot area would display the menu using a command, and then after a selection was made the menu would be undisplayed, so there would never be a need for conditions.

When a menu is displayed, all other hot elements will be deactivated. After the left mouse button is clicked, the menu will be automatically undisplayed and the other elements will become active again. The menu will be undisplayed regardless of whether a selection was made.

The menu can have a single black border and a 3D beveled border. The text color, hilighted text color, menu color, hilighted menu color and the font for the menu can be chosen. A cursor can be selected to be displayed when the mouse is over the menu.

Using a menu box

A list of menu items can be given in the element options. If a procedure name is given in the "Call … on display" option then this procedure will be called in the project script just before the menu is displayed. The procedure will take one parameter. This will be the name of the element.

display_procedure: *element_name*
    …

The procedure can use the following instruction to add other items to the menu.

menubox "*element name*" **add** "*item*"

If any item name is given as an empty string then this position in the menu will be used as a separator. A horizontal line will be drawn and no selection will be able to be made at this position.

When a selection is made from the menu, the procedure given by the "Call … on select" option will be called in the project script. This procedure will be passed three parameters. The first parameter will be the name of the element. The second will be the name of the item that was selected. The third will be the position of the item.

select_procedure: *element_name*, *item_name*, *position*
    …

Progress Bar

An area can be selected with a capture rectangle before choosing Progress Bar from the element menu. An empty progress bar will be displayed and a Progress Bar element dialog will open with some options.

The progress bar can be horizontal or vertical. The color of the progress bar and the color of the empty rectangle can be chosen. If the Border option is set, then the progress bar will have a 3D border. The border can have any pixel width and can appear as a raised bevel or a sunk bevel. If the 50% translucent option is set, then only every second pixel of the bars area will be filled.

The following instruction can be used to set the position of the progress bar. The position of the bar will be the proportion of the given position value to the given maximum value. These two values may be in any range.

progress "element name" **position** *position*, *maximum*

As an example if the given position value is 5 and the maximum value is 10 then the progress bar will appear at half way.

Child Window

An area can be selected with a capture rectangle before choosing Child Window from the element menu. A child graphics window will appear inside the main graphics window and a Child Window element dialog will open.

The child window will take the input focus away from the main window. Any elements that were on the main window will be disabled. All new elements will be displayed inside the child window - at a position relative to the top left corner of the child window. The child window must be undisplayed before you can use the main graphics window again.

Hypertext

An area can be selected with a capture rectangle before choosing Hypertext from the element menu. A Load Hypertext dialog box will open and a hypertext file can be selected. The formatted text file will appear in the graphics window and a hypertext element window will open.

The RTF file format is the industry standard for formatted text. Most word processors, including Microsoft Word will support the RTF file format. You can write your document in any word processor and save it as an RTF file. If you need to see the raw format you can edit your document with the Formula Graphics editor.

Most paragraph and character formatting are supported. Formula Graphics supports font type, font height, bold, italic, color, line breaks, page breaks, indentation, centering, tables, etc.

Any page number can be given in the "page" option. This will be the first page to be displayed. If the pages are longer than the window then you can use a scroll bar.

Multiple pages

One way of displaying a multiple page document is to use a different Hypertext element for each page. This way each page can have a different position on the screen and a different position in the list of elements.

Another way to display multiple pages is to use one Hypertext element and to display the different pages as required. There are several ways to display a page, one of the simplest is to use the following instruction.

display hypertext "*element name*" **page** *page*

Hot words

Hypertext can have hot words which will carry out an action when they are clicked. To specify a hot word in the Formula Graphics editor, highlight the selected words and choose Hyperlink from the Format section of the Editor menu.

To specify a hot word in a word processor such as Word for Windows, the selected hot words must first be formatted using a strikethrough font and a suitable color should be chosen to highlight them. The words will not appear as strikethrough when they are displayed by the element.

Immediately after the hot words, the action to be carried out must be written using hidden text. It is important that this hidden text is not strikethrough. The hidden text will not be displayed by the element.

The action associated with a hot word can be any command or combination of commands. For instance, the action may be to display another element, play a sound or execute a procedure in the project script. If the given action is not a known command then it will be assumed to be a hyperlink name.

Hyperlinks

At the top of each page of your document, you have the option of specifying a topic name, a hyperlink name, and some keywords. Topic names and keywords are only used when performing hypertext searches. During a search, if more than one page has a matching keyword then a list of topics will be shown.

When a hot word with a hyperlink name is clicked, then any page of the document with that hyperlink name will be displayed. Any number of hot words can point to the same page, but each page must have a unique hyperlink name. The hyperlink name must be one word only, made up of letters, numbers and

underscores.

A topic, hyperlink, and list of keywords can be added to the top of any page of the document using the Formula Graphics hypertext editor. Select Topic from the Format section of the Editor menu. After specifying a topic, hyperlink and some keywords, these will be inserted in the document.

To specify these items in a word processor, use a $ footnote for the topic name, a # footnote for the hyperlink name, and a K footnote for the keywords, in that order.

Using Word for Windows, choose Footnote from the Insert menu. Place a #, $ or K in the Custom Mark option and press OK. Then beside the mark in the footnote window, type your topic, hyperlink name or keywords.

Each keyword in a list of keywords must be separated by a semicolon.

This is exactly the same technique used to author Windows help files. Any RTF file using strikethrough, hidden text and footnotes can be used as hypertext by Formula Graphics or can be compiled directly into a Windows Help file by a help compiler. There are a number of help utilities available to assist you. There is considerable documentation written about this technique.

Hypertext procedure

You can use the project script to jump to any page with a hyperlink name. The following instruction will display the page of the document associated with the given hyperlink name.

display hypertext "*element*" **link** "*hyperlink*"

If you specify a procedure name in the "Call … on display" option then every time a new page of hypertext is to be displayed the given procedure will be called. The procedure will be passed four parameters.

The first parameter will be the element name. Second will be the topic of the current page if there is one. Third will be the hyperlink name of the current page if there is one. The fourth parameter will be the page number. The procedure should look like

```
hypertext_procedure: element_name, topic, hyperlink, page_number
    message "The element name is ", $element_name
    message "The topic name is ", $topic
    message "The hyperlink name is ", $hyperlink
    message "The page number is ", page_number
```

Searching for keywords

A word or keyword search can be performed on a hypertext document. The following instruction will open a Text Search dialog box. Every keyword in the document will be listed. If one of these keywords is selected, then the page associated with it will be displayed. If more than one page has that keyword, then another dialog box will open and the page numbers or the topics associated with those pages will be listed.

search hypertext "*element*"

If any phrase is typed into the Text Search dialog box and the Search button is pressed, then the document will be searched for any case insensitive occurrences of that phrase. If more than one page has an occurrence of the phrase, then another dialog box will open with the page numbers or topics associated with those pages.

This method of specifying and searching topics, hyperlinks and keywords is exactly the same as that used

by Windows Help. The same RTF files can be used by both Formula Graphics and the Windows Help compiler. For more information, refer to documentation for authoring Windows Help files

Converting a document

A word processor such as Microsoft Word can save any document as an RTF file. But usually the saved file will not only contain text and formatting information but will also contain a lot of unnecessary definitions and codes. And some characters such as quotation marks and extended language characters will be stored in ways that make them slow to display.

When you select Convert Document from the Editor menu, you will be asked to give source and destination files. The source file will be processed so that all unnecessary information is removed and characters such as quotation marks and international characters are properly resolved.

Keep your original files in case you need to edit them later. Save the newly processed files as FGH (lean text) files.

Caching

Formula Graphics needs to do a lot of work to find a hyperlink name in a large RTF file. This can take a significant amount of time. To solve this problem Formula builds a cache of indexes as it displays each page. Formula can then use the cache to display any page instantly. This cache will be saved to disk as a cache file (*.che). The next time the element is displayed the cache will be loaded and used.

At design time the cache file will be deleted each time the element is displayed because if the document is changed then the old cache will cause problems (the program may crash). The final cache files should be distributed with the project and can be stored in an archive file. In order to cache every page only the last page needs to be displayed.

Printing

An entire document, or a single page of the document can be printed to the default Windows printer using one of the following instructions in the script.

print hypertext "*element*"
print hypertext "*element*" **page** *page*

Using Word for Windows

You can make a hypertext element out of any document written using Word for Windows. The document can be exported using the RTF format by choosing SaveAs from the file menu and specifying "Rich text format (*.rtf) as the save file type. The document can have almost any type of formatting including different fonts, colors, and tables.

If you wish to have hyperlinks and keyword searching in the document then you will need to insert a topic name, hyperlink name and a list of keywords at the top of each page.

To insert a topic name, start by moving the cursor to the beginning of the page. Then select Footnote from the Insert menu. A 'Footnote and Endnote' dialog box will open. In the Insert option, choose Footnote. In the Numbering option, choose 'Custom mark' and type '$' in the custom mark box. Press OK and a window will open with 'all footnote' definitions. Then type your topic name beside the '$' sign.

To include a hyperlink name, repeat the above procedure but choose a '#' custom mark instead. Then in the 'all footnotes' window type your hyperlink name beside the '#' sign. To include a list a keywords use a 'K' custom mark. Separate each keyword with a semicolon.

Once you have formatted your document using this convention then you will not only be able to use it as a Formula Graphics hypertext element, but you will also be able to compile it into a Windows help file using the Microsoft help file compiler.

For the best performance use the Convert Document option from the Editor menu to convert the RTF file into a FGH file before choosing it as a hypertext element.

HTML

An area can be selected with a capture rectangle before choosing HTML from the element menu. A Load HTML dialog box will open and a hypertext markup language file can be selected. The formatted text file will appear in the graphics window and a HTML element window will open.

The HTML file format is the world wide web standard for presenting information. Many HTML editors are now available. If you wish to edit your HTML in the raw format you can use the Formula Graphics editor. Formula currently supports most HTML 3.0 specifications including tables.

World Wide Web

If a world wide web server name is given in the Server option then Formula will attempt to download the HTML file from the given server. The given server name should not contain the "http" specifier. But the server name can contain the directory where the file is kept.

example: web_server / directory / subdirectory

If the internet is not connected, or the server is not available then the HTML file will be loaded from the local project directory. If the file is not found in the local directory then an error will be generated.

Hot words

Hot words can be defined using the <A> tag. The action to be performed when the hot word is clicked can be given by the 'HREF=' attribute of the <A> tag. The following examples show the range of actions that can be made to happen by clicking on a hot word. The action will be considered in this order until a valid response is found.

<A HREF="any action command">Hot word</A>

If the attribute specifies a valid set of Formula Graphics action commands then these commands will be carried out.

<A HREF="#hyperlink name">Hot word</A>

A hyperlink name can be defined at any position in the HTML file using the <A NAME=…> tag. If the hyperlink name is defined in the file then the browser will jump to this definition.

<A HREF="another HTML file">Hot word</A>

If the specified HTML file exists, then the current file will be discarded and the specified HTML file will be loaded and displayed by the element.

Advanced features

A HTML element can be used to display a formatted list of information. The information can be read from any source, formatted into HTML, and then sent to the element to be displayed. A number of instructions are available to facilitate this.

The existing HTML document can be replaced with a new HTML document using the following instruction. The new HTML document can be given by any string expression.

html "*element name*" **display** "*html document*"

Example

```
html "my html" display "<HTML><BODY> Some HTML data </HTML></BODY>
```

The 'reset' instruction will empty the contents of the HTML element. The empty document will contain the standard header "<HTML><BODY>". The HTML element will now be ready for new information to be added.

html "*element name*" **reset**

The 'add' instruction will add more formatted text to the end of the currently displayed text. Text can be added to the element any number of times but the new contents will not be displayed until the 'update' instruction is called.

html "*element name*" **add** "*html information*"
html "*element name*" **update**

Example

The following example will display the name and address information contained in a list of structures. Each item will be made hot so that when it is clicked, a variable is set and a function is called. It would also be possible to include a bitmap with each row of data.

```
html "my html" reset
for n = 0 to my_data_list.max - 1
    data = @my_data_list[n]
    $new_info = "<A HREF = let choice = ",n,"; call selection_procedure>\
                        <B>", $data.name, "<: /B>\
                        <I>", $data.address, "</I>\
                        <BR></A>"
    html "my html" add $new_info
html "my html" update
```

Graph

An area should be selected with a capture rectangle before choosing Graph from the element menu. After specifying an element name, a Graph element window will open with some options. The graph may be a line graph, vertical bar graph, horizontal bar graph or pie graph.

Graduations can be shown beside the graph. They can be drawn in any given font or color. They will be drawn at a distance given by the margin value. Percentage differences can be shown between series on a vertical bar graph. Line and bar graphs can be drawn using stacked series. The "Line width" option will not only set the width of lines, but will also set the distance between bars.

A procedure name must be given. This procedure will be called before the graph is displayed. The procedure will be expected to return a list of parameters. The information given by these parameters will be used to draw the graph.

The first parameter may be a one dimensional floating point array of the form array[categories]. Each element of the array will contain the value of one category. The number of elements will define the number of categories.

The first parameter may also be a two dimensional array which has the form array[series][categories]. The number of elements in the first dimension will indicate the number of series. The number of elements in the second will indicate the number of categories.

The second parameter must be a two dimensional byte of array containing the red, green and blue values of the graph's colors. The array must be of the form array[series][3] where the 0, 1 and 2. The first color will be used for the first data series, the second color will be used for the second data series and so on.

The third parameter will be the minimum value of the graph scale, the fourth parameter will be the maximum value of the graph scale, and the fifth parameter will be the increment of the graduations between the two.

Your graph procedure should look similar to this

```
graph_procedure:
    data = new float[12]            // 12 months of the year
    ...fill the array with data ...

    colors = new byte[1][3]         // one series
    colors[0][0] = 128,128,192              // muddy blue

    min = 0                         // min value on the scale
    max = ...                               // max value on the scale
    inc = 10                        // 10 graduations

    return @data, @colors, min, max, inc
```

Script

After selecting Script from the element menu, a Load Script dialog box will open and a Script file can be selected. Then a Script element window will open with some options. Four procedures can be specified by the element.

The 'Display' procedure will be called when the element is supposed to be displayed. It will not be passed any parameters.

The 'Undisplay' procedure will be called when the element is supposed to be undisplayed. It will be passed one value. This value will be TRUE if the element was undisplayed, or FALSE if it was removed.

The 'Left click' procedure will be called when the left mouse button is clicked in the graphics window. It will be passed three values. The first and second values will the x and y positions of the mouse cursor in the graphics window. The third value will be TRUE if the button is clicked down or FALSE if it is clicked up.

The 'Right click' procedure will be called when the right mouse button is clicked in the graphics window. It will be passed three values. The first and second values will the x and y positions of the mouse cursor. The third value will be TRUE if the button is clicked down or FALSE if it is clicked up. If the 'Right mouse button to advance **...**' option is set in the project options then the screen will end when the right mouse button is clicked.

The 'Double click' procedure will be called when the left mouse button is double clicked in the graphics window. It will be passed two values which are the x and y positions of the mouse cursor in the graphics window.

If the mouse cursor moves over the graphics window or then the 'Mouse move' procedure will be called. It will be passed two values which are the new x and y positions of the mouse cursor.

The Script element is an advanced element for authors who can understand the basics of programming. Some of the typical uses of the script element are to replace the mouse cursor with an animated sprite, or to build complex controls.

Message

After selecting Message from the element menu and specifying an element name, a message properties window will open. Several procedure names can be given in this element. When the specified event occurs, if a procedure name has been given then this procedure will be called in the project script. The procedure will be called before any hot elements are tested, or any other event occurs.

The **Left click** procedure will be called when the left mouse button is clicked in the graphics window. It will be passed three values. The first and second values will the x and y positions of the mouse cursor in the graphics window. The third value will be TRUE if the button is clicked down or FALSE if it is clicked up.

The **Right click** procedure will be called when the right mouse button is clicked in the graphics window. It will be passed three values. The first and second values will the x and y positions of the mouse cursor. The third value will be TRUE if the button is clicked down or FALSE if it is clicked up. If the 'Right mouse button to advance **…**' option is set in the project options then the screen will end when the right mouse button is clicked.

The **Double click** procedure will be called when the left mouse button is double clicked in the graphics window. It will be passed two values which are the x and y positions of the mouse cursor in the graphics window.

If the mouse cursor moves over the graphics window or then the **Mouse move** procedure will be called. It will be passed two values which are the new x and y positions of the mouse cursor.

The **Keyboard** procedure will be called when any key on the keyboard is pressed. It will be passed three values. The first value will be the key value (same as the ANSI value). The second value will be 1 if the key was pressed down and 0 if the key was released.

The third value will be a flag indicating the states of the Alt, Control and Shift keys. The Alt key has a value of 1. The Control key has a value of 2. The Shift key has a value of 4. If more than one key is pressed then their values will be added together.

The **Message** procedure will be called when any other Windows message is generated. It will be passed three values. The first value will be the message type. The second and third values will be the wparam and the lparam values.

For more information on the mouse and keyboard see Mouse and keyboard. For more information on messages see the Microsoft Windows SDK documentation.

Input

After selecting Input from the element menu and specifying an element name, an input properties window will open. The default type of input is "Key press or mouse click".

A "Key press or mouse click" input will wait until a mouse button is clicked or a key is pressed before advancing to the next element.

An "Activate hot elements" input will wait for a response from the user. If the left mouse button is clicked over any hot element such as a button, text button, hot area, or hot color, then the commands given by those elements will be carried out. These commands may result in a change of condition for some elements, so the conditions of all elements will be updated.

First the system will check the list of visible elements, from the front of the list to the back. Any elements whose conditions are no longer true will be undisplayed. Then the system will check the elements on the current screen, from the top of the list to the bottom. Any elements whose conditions have just become true will be displayed.

If the right mouse button is clicked and the "Right mouse button to advance" option is set, then the system will advance to the next element. Any element can also make the system advance to the next element by carrying out a "continue" or "break" command.

A period of time can be given for the "Activate hot elements" option. After the period of time has elapsed without any response from the user, the system will advance to the next element.

Advanced features

If an "Active procedure" input is activated then the system will "activate hot elements" while executing a procedure in the project script. The system will continue to activate the hot elements and execute the procedure until either a right mouse click or a "continue" or "break" command or the procedure returns a zero value.

The procedure will be passed 4 parameters which contain message information given the by Windows operating system. These parameters are the message type, the wparam, the low word of the lparam and the high word of the lparam. The procedure must return a value.

Your procedure should look like

```
my_procedure: msg, wparam, loword, hiword
    switch msg
        case WM_MOUSEMOVE     // test for the mouse move
            xpos = loword
            ypos = hiword

        case WM_CHAR              // test for keyboard key
            message wparam, " key"

    return TRUE
```

During the execution of this procedure, no messages will be processed. All messages will be preserved for the activation of hot elements. Care must be taken to have no infinite loops in this procedure otherwise the system may lock up. For more details on Windows messages see Mouse and keyboard in the programming language section or read the Windows SDK documentation.

Wait

This element can be used to wait for a period of time to elapse. The "Wait for ... 1/100 seconds" option will specify the waiting time. There are four modes of waiting.

The "wait for specified time" option will simply wait for the specified time to elapse before advancing to the next element. There is a project option available to stop waiting if the Spacebar or Enter key is pressed.

The "wait for key press or mouse click" option will wait the specified time or until any key is pressed or any mouse button is clicked.

The "wait for active hot elements" option will wait the specified time. During this time all hot elements will be active. Buttons can be pressed and actions will be carried out.

If "Wait for sound to finish" is selected then the system will wait until the current sound is finished playing before advancing to the next element. The specified waiting time will be ignored.

If "Wait for all elements to finish" is selected then the system will wait until all animations, videos, sounds, midis and cd audios are finished playing before advancing to the next element.

Timer

A timer element can be used to carry out a set of commands after a period of time has elapsed. The time period must be given in hundredths of a second, although the Windows timer may not be that accurate. There are four modes of timer.

The first mode will carry out the given commands after the specified period of time has elapsed. The system does not wait for the time period to elapse. During the time period any other events may be happening.

The second mode will carry out the given commands again and again. After each period of time has elapsed the commands will be carried out and the timer will begin again.

The third mode will carry out the first set of commands after the first period of time has elapsed. It will then carry out the second set of commands after the second time period has elapsed. As an example, the first set of commands may be used to display an element, and the second set of commands may undisplay the element.

The fourth mode will carry out the first and second sets of commands again and again. After the first period the first commands will be carried out. Then after the second period the second commands will be carried out. Then the timer will begin the first period again.

The timer will stop after the timer element is undisplayed. The timer can also be stopped and started again using the "stop element" and "play element" commands or instructions.

Example

A timer can be used to make an element flash on and off. Set the element to use the fourth mode, "periods on and off continuously". If the first and second time periods are both 100 then the element will appear for 1 second and disappear for 1 second. The first action should be - display "element name" - and the second action should be - undisplay "element name".

Action

An action element has nothing more than a condition and a set of commands. If the condition is true then the commands will be carried out. Any numerical expression can be given in the "If this is true …" box and any number of commands can be given in the "… then carry out these commands" box. Commands should be separated by semicolons and spaces.

One of the most common uses of a control element is to set the value of a variable. A variable can be set using "**let** *variable* **=** *value*". A string variable can be set using "**let** $*variable* **=** "*string*".

If an action element is used to change the value of a variable, and that variable is used in the conditions of other elements, then to use the new condition to display or undisplay those elements an update command must be carried out. As an example consider the following action commands.

```
let my_condition = TRUE; update
```

Another common use is to call a procedure in the project script. A procedure can be called using a "**call** *procedure_name*". After the procedure has been executed control will return to the element and either the next command will be carried out or the next element will be displayed.

Remove

After selecting Remove from the element menu and specifying a name, a Remove Element window will open. The remove element has four different options.

The Remove option will remove all of the listed elements from the screen without undisplaying them (without restoring the old image). The Undisplay option will undisplay all of the listed elements.

Any number of element names can be listed, one element on each line. The '*' character can be used as a wildcard, for example, if the name "my*" is specified, then all elements whose names begin with the letters "my" will be removed.

The "Remove all" option will remove all elements without undisplaying them. If there is a background then this will not be removed. The "Undisplay all" option will undisplay all elements except for backgrounds.

Debug

After selecting Debug from the element menu and specifying a name, a Debug element window will open. Any numerical or string expression can be given in the "… then display this message …" option.

If a project is playing and it gets to a Debug element, the given value or string will be displayed. The message will be displayed in the result window if it is open, otherwise it will be displayed in a message box. If the project is running from the command line then this element will be ignored.

Under Construction

This element is still under construction…

Programming Language

Basics

The Formula Graphics programming language is similar to other programming languages like BASIC and C. A wide variety of instructions are available and each instruction performs some operation. A program can be written using a list of instruction statements. Only one instruction can be written on each line. The instructions will be executed one at a time in the order they are listed. After the last instruction has been executed, the program will end.

first instruction
second instruction
...
last instruction

There are a number of instructions available which can change the flow of execution. These include condition statements, which only execute a branch of the program if a certain condition is true, loop statements, which execute the same group of instructions again and again, and call statements, which branch off and execute some other group of instructions before returning.

Indentation is used in Formula Graphics programs to indicate the possible paths of execution. Indentation refers to the number of tabs or spaces at the beginning of a line. For example, an "if" statement can be used to test if a certain condition is true. If the condition is true, then all following statements with a greater level of indentation will be executed. If the condition is false, then those statements with greater indentation will be skipped.

if condition is true
    then execute these indented statements
    ...
in any case execute these
...

Comments can be added to a program. These are usually just a few words describing what the program does. Anything written after a double forward slash *//* specifier will be considered a comment and will be ignored by Formula Graphics. A comment can be alone or placed on the same line as an instruction. Blank lines and comments can be included between indented lines without effecting the levels of indentation.

// This is a comment

Some instructions may have so many words that they become too long to fit on one line. One instruction can be written across more than one line by using a backslash \ as a continuation character.

first line of the instruction\
  another line\
  and another

Example

The following is an example of a simple program.

```
message "Hello, I'm going to count to ten now"

for my_counter = 1 to 10                    // execute the following line ten times
    message "Counting... ", my_counter              // show the counter value

message "I'm finished counting now"
```

Floating point variables

Floating point numbers are used in Formula Graphics programs. A floating point number can be whole number or a fraction or it can be positive or negative. When using a floating point number, fractions less than one must have a zero before the decimal point. 0.05 and -1.414214 are examples of floating point numbers:

A floating point number can also have a power of ten exponent which is specified with a lower case "e". As an example, the value of 1e6 will be 1 times 10 to the power of 6 (or 1000000). The exponent may be positive or negative.

Variables are used extensively in programming, in fact they are one of the most important ingredients in any program. There are several different types of variables. Each variable has its own individual name. A variable name can be made up of case sensitive letters, numbers and underscores. The first character must be a letter.

The simplest form of variable is a floating point variable. A floating point variable is created when a variable name is assigned a floating point value. That variable name can then be used as a substitute for its value anywhere else is the program.

variable name **=** *floating point value*

A floating point value can be just a floating point number, or it can be an expression containing numbers and variables combined with operators like '+' and '-'. A complete set of mathematical operators and functions are available for use in floating point expressions. Spaces can be used where necessary.

As an example consider the instruction "x = 1". The variable "x" will be assigned with the value "1". The variable "x" can then be used elsewhere in the program to represent the value "1". The value of the variable "x" can be changed by reassigning it with another value. For instance "x = x + 1" will add one to the value of "x".

Any number of variables can be assigned with values in the same instruction. The variables and values must be separated by commas, as an example "x, y, z = 10, 20, 30". A variable assignment can be written in the general form:

variable 1, *variable 2*, **...** **=** *expression 1*, *expression 2*, **...**

Example

The following is an example of a program using variables.

```
my_first_variable = 10
my_variable_doubled = my_first_variable + my_first_variable
my_variable_doubled_and_halved = my_variable_doubled / 2
if my_first_variable != my_variable_doubled_and_halved
    message "if 10 is not equal to (10 + 10) / 2 then the universe is broken !"
```

Condition statements

If the condition given in an **if** statement is true, then all following statements with a greater level of indentation will be executed. If the condition is false, then those statements with greater indentation will be skipped.

if *condition*
    …
    …

If an **else** statement follows an **if** statement, and the condition is false, then all statements following the **else** statement with a greater level of indentation will be executed, otherwise they will be skipped. Any number of **else if** statements can also follow an if statement.

else if *condition*
    …
else
    …

If more than one level of indentation is used, then each **else** statement will correspond to the **if** statement above it that has exactly the same level of indentation. If you are using tabs then make sure that there are no hidden spaces.

The condition given by an **if** statement can be any floating point expression. If the value of the expression is not equal to zero, then the condition will be true. If value of the expression is equal to zero, then the condition will be false. A range of operators are available for comparing floating point values.

The **==** (equal to) operator will compare two values, and if they are equal the result will be one, otherwise the result will be zero. Using the **!=** (not equal to) operator, if the two values are not equal the result will be one, otherwise the result will be zero. The **<, >, <=** and **>=** (magnitude) operators can also be used to compare two values.

The result of two or more comparisons can be combined using boolean logic. The **&&** (and) operator will compare two values, and only if they are both not equal to zero the result will be one, otherwise the result will be zero. The **||** (or) operator will compare two values, and if either of them is not equal to zero then the result will be one, otherwise the result will be zero.

If the **!** (not) operator is applied to a zero value the result will be one, if it is applied to a non zero value the result will be zero.

If there is only one instruction to be executed by the condition then almost any type of instruction can be written on the same line.

if *condition* **then** *instruction*
else if *condition* **then** *instruction*
else *instruction*

Example

```
if x > 10
    message "x is greater than 10"
else if y > 10
    message "y is greater than 10"
else
    if z > 10 then message "z is greater than 10"
    else if z == 10 then message "z is equal to 10"
```

```
    else message "z must be less than 10"
```

Switch statements

A **switch** statement can specify one or more floating point values. This statement can be followed by a number of indented **case** statements. Each case statement must specify the same number of floating point values.

```
switch value, value, …
    case value, value, …
        …
    case value, value, …
        …
     …

    default
        …
```

When the switch statement is executed, each case statement will be checked to find one with equal values. If such a case is found then all statements with greater indentation following the case will be executed. If no such case is found and if a **default** case has been declared then all indented statements following the default will be executed. The other cases will be ignored.

Example

```
x, y, z = 1, 1, 1

switch x, y, z
    case 0, 0, 0
        message "This message will not be displayed"
    case 1, 1, 1
        message "This message will be displayed"
    default
        message "This message would be displayed if nothing else was"
```

Loops

For loops

A "for" loop can be used to increment the value of a variable from a starting value to a finishing value. All following statements with a greater level of indentation will be included in the loop.

for *variable* **=** *starting value* **to** *finishing value* { **step** *increment* }

The given variable will be assigned with the starting value, then all following statements with greater indentation will be executed. The value of the variable will then be incremented by the optional step value. If no step value is specified, then the variable will be incremented by one.

As long as the value of the variable is less than or equal to the finishing value, then the variable will continue to be incremented, and all following statements with greater indentation will continue to be executed. As an example:

```
for n = 1 to 10
    // These lines be executed 10 times
    ...
// The program will continue here
 ...
```

A negative step value may be specified, in which case the loop variable will continue to be decremented for as long as its value is greater than or equal to the finishing value.

for *var1, var2, ...* **=** *val1, val2, ...* **to** *val1, val2, ...* { **step** *inc1, inc2, ...* }

Any number of variables can be specified in a single "for" statement. The variables will be incremented in separate nested loops with the first variable being in the outer loop. Using more than one loop, an area or a volume of values can be generated.

Example

```
for n = 1 to 3
    for m = 1 to 3
        message "n = ", n, ", m = ", m
```

These instruction are the same as

```
for n, m = 1, 1 to 3, 3
    message "n = ", n, ", m = ", m
```

The result window will show

```
n = 1, m = 1
n = 1, m = 2
n = 1, m = 3
n = 2, m = 1
n = 2, m = 2
n = 2, m = 3
n = 3, m = 1
n = 3, m = 2
n = 3, m = 3
```

While loops

A "while" loop can be used to continuously execute all following statements with greater indentation for as long as the value of the specified expression is true (not equal to zero).

while *expression*

The following example shows a while loop used to count to 10:

```
n = 0
while n < 10
    n = n + 1
```

If a **break** statement is found during the execution of a loop, the loop will immediately finish, and execution will jump to the first line after the loop.

If a **continue** statement is found during the execution of a loop, the current iteration of the loop will finish, and execution will begin again at the top of the loop.

There are some other, more specialized types of loops which will be discussed later.

Arrays

An array is an object used for storing lots of values. An array can be allocated with any number of dimensions. Each dimension can have any number of elements. Each element in an array can be used to store a value.

As well as arrays of floating point numbers, Formula Graphics also handles arrays of bytes and words. A byte is an 8 bit unsigned value between 0 and 255. A word is a 16 bit signed value between -32768 and 32767.

Each of the following instructions will allocate an array, the elements of the array will be set to zero, and the array will be assigned to the variable:

variable **= new byte [** *dimension 1* **][** *dimension 2* **] ...**
variable **= new word [** *dimension 1* **][** *dimension 2* **] ...**
variable **= new float [** *dimension 1* **][** *dimension 2* **] ...**

The position of an element in an array can be given using an index value for each dimension. An element can be assigned with a value and then be used to represent that value in an expression. Byte and word values are automatically converted to and from floating point values.

variable **[** *index 1* **][** *index 2* **] ... =** *expression*

The term *variable* **[** *index 1* **][** *index 2* **] ...** can then be used in any floating point expression to get the value of the element.

Any number of consecutive array elements can be assigned with values in the same instruction. Only the index values of the first element need to be specified, all other values will be copied into consecutive positions.

variable **[** *index 1* **][** *index 2* **] ... =** *expression, expression, ...*

Example

The following example allocates a floating point array with six elements, and then assigns a value to each of the six elements:

```
my_array = new float[6]
my_array[0] = 1,2,3,4,5,6
```

The following example allocates a two dimensional floating point array. There are five elements in each dimension giving a total of 25 elements. Each element is then assigned a value:

```
array_two = new float[5][5]
for n,m = 0,0 to 4,4
    array_two[n][m] = n*5+m
```

Array properties

The term **$***variable***.type** can be used in any string expression to return the type of the array. The possible of arrays are: "Byte Array", "Word Array" and "Float Array".

The *variable***.max** property can be used in any floating point expression to return the number of dimensions in the array.

To return the size of each dimension use the function *variable* **(** *dimension* **)** in any numerical expression.

Lists

A floating point list can be used to store any number of values. Values can be added, inserted, removed, or gotten from a list. The following instruction can be used to create a new floating point list and assign it to a variable. When the list is first created it will be empty.

list variable **= new float list**

When a number is added to the list, the list will grow. When a number is removed from the list, the list will shrink. The following instructions can be used to add, get, insert and remove numbers from a list.

list variable **push** *value*

The pop instruction will remove an object from the top of the list.

variable **=** *list variable* **pop**

The value at a given position can be returned using the get instruction. A value can also be returned by using the term *list variable* **[** *position* **]** in any numerical expression.

variable **=** *list variable* **get** *position*

The list can be searched for a value. If the value is not found in the list then returned position will be -1.

position **=** *list variable* **find** *value*

A value can be inserted at any position in the list. Any values above that position will be shifted up to make a space.

list variable **insert** *value* **at** *position*

The value stored at a given position in the list can be replaced with another value.

list variable **assign** *value* **at** *position*

A value can be removed from any position in the list. Any values above that position will be shifted down to fill the space.

list variable **remove** *position*

The following instruction will remove every value from the list. The list will become empty.

list variable **remove all**

The current number of items on a list can be obtained using the *list variable***.max** property.

Example

The following example demonstrates a few manipulations using floating point lists.

```
my_float_list = new float list          // create a new float list

for n = 0 to 3                                  // add 3 values to the list
    my_float_list push n

my_float_list remove 2                  // remove the 2nd value
```

```
my_float_list insert 4 at 1                    // insert 4.0 at position 7
my_float_list[0] = 5                           // assign 5.0 to position 0

for n = 0 to my_float_list.max-1               // display the values
     message "list[",n,"] = ",my_float_list[n]
```

The output to the result window will be:

```
list[0] = 5
list[1] = 4
list[2] = 1
list[3] = 3
```

Strings

A string is a collection of characters that can form a word or a sentence. A string is usually given inside quotation marks, for example, "this is a string". Strings are most often used to give names to things.

Each character in a string is represented by a byte value. The value of each keyboard character was standardized many years ago. The byte value of a character may be used by enclosing it in single quotation marks, for example 'A' has the value 65. The following program will display a list a characters and their values.

```
for n = 0 to 255
    message n, " = ", strchr n
```

A byte array can be used to store a string of characters. A byte array becomes a string by preceding its variable name with a '**$**' sign. A string variable can be assigned with a string expression.

$*string variable* **=** "*string expression*"

A string may be a group of characters inside of quotation marks, or it may be a string variable, global string variable, floating point value, or any combination of these separated by commas. Strings separated by commas will be added together form a single string. Floating point values will be automatically converted into strings.

$*string variable* **=** "string in quotes", $string_variable, %global_variable, number

When a string variable is assigned with a string expression, a byte array is created and assigned to the variable. The size of the array will be equal to the length of the string expression plus one. The string expression will be copied into the array and a zero value placed after it to indicate the end of the string. The following are examples:

```
$string_one = "This is a string"
$string_two = $string_one, " and the first character value is ", string_one[0]
```

In the above examples, a byte array called "string_one" will be allocated with 17 byte elements, then the string will be copied into it and the 17th element will be zero. A second array called "string_two" will then be allocated with enough bytes to store the first string plus the rest of the sentence. "84" will be added to the string to represent the character value of 'T'.

Displaying strings

A **message** instruction can be used to display a string. If the program is being run from the Formula Graphics environment, then the string will be displayed in the result window. If it is run from the command line, then the message will appear in a dialog box. Each new message will be displayed on a new line.

message "*string expression*"

The following example shows the value of a floating point variable.

```
message "The value of x is ", x
```

A debug message can be displayed by the following instruction. Debug messages can be useful during the development of a script. If the program is being run from the environment then the message will be displayed. If it is run from the command line, then the message will be ignored.

debug "*string*"

An error message can be displayed by the following instruction. This instruction will stop the execution of the program. The error string will either be displayed in the result window or in a Windows message box.

error "*string*"

String arrays

If a byte array is allocated with more than one dimension then it can be used to store a number of strings. The last dimension of the array must have enough bytes to store the longest string plus one. When an array is used to store strings, no index value needs to be given for the last dimension.

In this example a two dimensional byte array is used to store 10 strings. The longest string is less than 20 characters. The array is declared as

my_string_array = new byte[10][20]

and the first string on the array can be assigned

$my_string_array[0] = "*string expression*"

Any number of strings can be copied into a string array using the same instruction. Only the index values of the first string need to be specified, all other strings will be copied into consecutive positions. Semicolons must be used as separators.

$*string variable* **[** *index* **] ... =** "*string expression*"***;*** "*string expression*"***; ...***

Example

This example allocates a string array which is used to store five strings.

```
my_string_array = new byte[5][10]
$ my_string_array[0] = "string 1"; "string 2"; "string 3"; "string 4"; "string 5"
```

String operators

A number of operators are available for working with strings. The following operators can be used in any floating point expression. The strings used by these operators can be any string expressions.

strlen "example" (gives the length of the string, in this case 7)
strval "1.41421" (gives the floating point value of the string, or 0 if not a number)
strstr $a**;** $b (gives the offset of first occurrence of $b in $a, otherwise gives ERROR)

The following operators can be used in any string expressions. The strings used by these operators can be any string expressions.

"example" **stroff** 2 (takes the string from the given offset, in this case "ample")
"example" **strcnt** 4 (takes only the given number of characters, in this case "exam")
"example" **stroff** 2 **strcnt** 3 (takes the given number from the given offset, in this case "amp")

Example

As an example consider the following expression:

"abcdexyz" stroff (strval "2") strcnt (strlen "hello")

Reduces down to

"abcdexyz" stroff 2 strcnt 5

The final result of the expression:

"cdexy"

String comparisons

There are four operators that can be used to compare two strings. These operators can be used in any numerical expressions and will return 1 if the comparison is true, or 0 if it is false. These expressions are commonly used as the conditions for elements and "if" statements.

string expression **==** *string expression* (true if the strings are the same)
string expression **!=** *string expression* (true if the strings are not the same)
string expression **~=** *string expression* (true if the strings are the same regardless of case)
string expression **!~** *string expression* (true if the strings are not the same regardless of case)

As an example consider the following

```
if $my_string_1 ~= $my_string_2 then message "Same string - case may be different"
```

Switch statements

A **switch** statement can specified with one or more strings.

```
switch "string"; "string"; …
    case "string"; "string"; …
     …
    case "string"; "string"; …
          …
     …

    default
         …
```

When the switch statement is executed, each case statement will be checked to find one with case insensitive matching strings. If such a case is found then all statements with greater indentation following the case will be executed. If no such case is found and if a **default** case has been declared then all indented statements following the default will be executed. All other cases will be ignored.

Escape characters

If you try to use quotation marks "" inside a quotation you will not be very successful. The way to include a quotation mark is to precede it with an escape character '\'. As an example "you asked me \"why can't I use quotation marks\"".

Escape characters can also be used for tabs and new lines. If the tab cannot be given literally then it can be given as "\t". A new line can be specified inside a string using the escape sequence "\n".

When a backslash character '\' needs to be used in a string it must be given as '\\' otherwise it may be confused with an escape character. This is particularly important for filenames which must be written in the form "drive:\\path\\filename".

New lines

The carriage return and line feed characters are used to specify a new line. They mark the end of the current line and the beginning of a new one. Pressing Enter in an editor will generate a carriage return and line feed. A carriage return has the value 13 and a line feed has the value 10. Almost every text file format uses carriage return, line feed pairs to specify new lines.

A carriage return and line feed can be added to a string by appending the **CRLF** string constant. The following example shows how to use carriage returns and line feeds to display one string across several lines.

```
message "One line", CRLF, "after the next", CRLF, "after the next", CRLF
```

Text files

The contents of any file can be loaded from disk and stored in a one dimensional byte array. The array will be allocated with enough space for the entire file. The contents of the file will be loaded into the array and the array will be assigned to the given variable.

load "*file name*" **byte** *array variable*

The entire contents of a byte array can be saved to a file. The file name can be given as any string expression. The length of the file will be the same as the length of the array.

save "*file name*" **byte** *array variable*

A string can be added onto the end of a text file using the following instruction. If the file does not exist it will be created. The string can be any string expression. Each time this instruction is called the file will grow in size.

save "*file name*" **string** "*string*"

Example

The following example uses a text file to keep track of the number of times an event has been carried out. It records the number of hits, and the time and date of the last hit.

```
my_hits = 0
if file_exist "log.txt"                          // if the log exists
    load "log.txt" byte my_log                       // read the log file
    my_hits = strval $my_log stroff 16               // extract the number of hits

delete "log.txt"                                     // delete the old log
save "log.txt" string "Number of hits: ", my_hits+1  // increment the number
save "log.txt" string CRLF                           // new line
save "log.txt" string "Last hit: ",TIME," on ",DATE  // save the time and date
```

Parsing a text file

A byte array can be searched for a given string. The search will begin at the specified array index and will continue until either the string is found, or the end of the array is reached. If the string is found, the given variable will be assigned with the first index of the search string in the array. If the string is not found, the variable will be assigned with the value ERROR.

index variable **= parse** *array*[*index*] **for** "*string*"

The text in a byte array can be broken down into separate terms. A term may be a floating point number, a symbol (case sensitive letters, numbers and underscores, the first character must be a letter) or a quote (any characters contained in quotation marks). Spaces and punctuation marks will be ignored.

index variable **= parse** *array*[*index*] **to** $*result*

This instruction will begin searching at the specified index of the array. If a term is found, it will be assigned to the result variable and the index variable will be assigned with the first index after the term. When there are no more terms to be found, the variable will be assigned with ERROR.

Example

A database can be given as a text file. The database text can be searched for a string matching the name

of the desired table and a string matching the name of the desired row. Then the contents of each column can be copied to another array.

```
read_database: state, account
    my_data = new float[10]
    load "textdata.txt" byte textdata                   // load the database text file
    pos = parse textdata[0] for $state                          // search for the desired table
    pos = parse textdata[pos] for $account              // search for the desired row
    pos = parse textdata[pos] to $buffer                // move to the first field
    for n = 1 to 10
        pos = parse textdata[pos] to $field             // store data in an array
        my_data[n] = strval $field
    return my_data
```

Delimiters

A table of data can be saved as a text file. The fields of data are usually separated by commas or new lines. By specifying a particular delimiter, you can easily extract any type of data from the text. For example consider a list of names separated by commas, the delimiter would be "**,**".

index variable **= parse** *array*[*index*] **delimiter** "*delimiter*" **to** $*result*

Example

Suppose you wish to load a text file containing a list of file names where each file name is on its own new line. You want to separated the file names and store them in a string array. The delimiter that marks the end of each line is a carriage return and line feed.

```
load "my_names.txt" byte my_names                   // load the text file
my_filenames = new byte[100][16]            // space for up to 100 names

pos = 0
for n = 0 to 99
    pos = parse my_names[pos] delimiter CRLF to $buffer    // new line delimiter
    if pos == ERROR then break                             // end of list
    $my_filenames[n] = $buffer
```

Database tables

A text file might contain a table of text data. Any string delimiter can be used for each column and row. The following instruction will parse through the table extracting the fields and storing the result in a three dimensional string array.

array **= parse database** *text* **delimiter** "*column delimiter*"**;** "*row delimiter*" (**fieldwidth** *width*)

The size of the first dimension of the array will refer to the number of rows. The number of rows can be found using *array variable*(0). The size of the second dimension refers to the number of columns. It can be found using *array variable*(1).

The size of the third dimension refers to the width of each field. This can be set using the optional fieldwidth specifier. The default field width is 32, any fields longer than this will be truncated.

Example

This example will load a text file, parse the text file into a table, and display the individual fields.

```
load "test.dat" byte text_array
table = parse database text_array delimiter "\t"; "\n"
message "rows ", table(0)," columns ", table(1)," field widths ", table(2)
for n,m = 0,0 to table(0)-1, table(1)-1
      message $table[n][m]
```

Global variables

Global variables can be used to store numbers as well as strings. Once a global variable has been assigned a number or a string then every script, element and screen will have access to that variable. The contents of the global variable will also be saved to disk so it can be used again the next time the program runs.

Global variable names are preceded by an '%' sign. A global variable is created when it is assigned a number or a string. The variable can then be used to represent that value or string in any other expression.

%*variable* **=** *expression*
%*variable* **=** "*string expression*"

If a global variable name has not previously been assigned anything, then it will return a zero value or an empty string.

If the project is being run from inside the Formula Graphics environment then the contents of global variables will be saved in the Formula Graphics initialization file "formula.ini" in the Windows directory. If the project is being run from the command line then the contents will be saved in the project initialization file "*project*.ini".

Objects

When a variable is assigned with an object such as an array, then that variable becomes a handle to the object. If the same object is later assigned to another variable then there will be two handles to that object. Any number of handles can exist for the one object. The easiest way to get another handle is just to assign one.

second variable **=** *first variable*

After a variable has been assigned with an object of any type then it can be assigned again with other objects, but it cannot be assigned again with a floating point number.

A variable name can be tested to see if it has been assigned to an object. The **valid** *variable* function can be used in any numerical expression and will return a value of 1 if the variable has been assigned or 0 if it has not.

Freeing an object

A variable name can be cleared from its assignment using a **free** statement. If the variable was used as a handle for an object then that object will now have one less handle pointing to it. Once a variable name has been freed it can then be used again with any type of data.

Any number of variables can be freed by the same instruction.

free *variable*, *variable*, **…**

Using one instruction every variable used by a script can be freed.

free all

If a variable was being used as a handle for an object, and it was freed, and if that object then has no other handles pointing to it (and no handles in lists or in other scripts) then that object will be removed from the system and its resources will be freed.

Object Expressions

Most instructions that take an object as a parameter will also take any object expression. The most common types of object expression are object properties and object array elements. Some instructions use the **@** operator to differentiate between the syntax of an object expression and a numerical expression. As an example consider the assignment of an object expression to a variable.

variable = **@***object expression*

Copying an object

An identical copy of an object can be made. The following instruction will copy an object and assign the copy to the given variable. After this instruction there will be two objects each assigned to its own variable.

new object **= copy** *object expression*

Comparing two objects

You can compare two variables to see if they are handles to the same object. The following comparisons can be included in any floating point expression. They will return 1 for true or 0 for false.

@*object_1* **==** **@***object_2* (equal to)
@*object_1* **!=** **@***object_2* (not equal to)

Object Properties

An object can be assigned a name. This name can be used later as a reference. For example it can be used to retrieve the object from a list of objects. You can assign a name to an object using the "name" property.

$*variable*.**name** = "*string expression*"

The object's name can be used later in a string expression by referring to it as "$*variable*.name".

Some objects, such as arrays, have their own special properties. The number of dimensions in an array can be found using "*variable*.max". Object properties can be numbers, strings or other objects. When using a string property the variable name must be preceded with an '$' sign. When using an object property the variable name must be preceded with an '@' sign.

variable **.** *property* **=** *numerical expression*
variable **.** *property* **=** **@***object expression*
$*variable* **.** *property* **=** "*string expression*"

The $*variable*.**type** property can be used to determine the type of object that has been assigned to a variable. The return string may be a "Byte Array", "Word Array", "Float Array", "Float List", "Structure", "Object Array", "Object List", "Script", "Bitmap", "Animation", "Sprite", "Sound", or some other object type.

Structures

A structure is an object which can be used to store a set of related information. Numbers, strings and objects can all be stored in a structure. The following instruction can be used to create a new structure and assign it to a variable. To begin with the structure will be empty.

structure variable **= new structure**

Once the structure has been created it can then be assigned with data. The structure will grow as each number, string and object is assigned to it.

structure variable **.** *property* **=** *expression*
structure variable **.** *property* **= @***object variable*
$*structure variable* **.** *property* **=** *string expression*

Any numerical property stored in a structure can be used in any numerical expression as

structure variable **.** *property*

Any object property can be used in any object expression as

s*tructure variable* **.** *property*

Any string property can be used in any string expression as

$*structure variable* **.** *property*

Example

As an example, suppose you need to store the contact details of a customer. You can create a structure and then assign it with all the information. The entire customer record can then be passed around as a single object.

```
customer = new struct
$customer.name = "Mr Black"
$customer.address = "123 Red St, Blueville"
customer.important = TRUE
customer.age = 42
```

Saving to disk

A structure can be saved to disk as a data file. Only the numbers and strings that were stored in the structure will be saved. The data file will be in the same format as a project file which can be viewed using any text editor.

save "*filename*" **object** *structure*

A structure can then be loaded from disk. The following instruction will create a new structure and assign it to the given variable. The data from the given data file will be used to fill the structure.

load "*filename*" **structure** *structure*

An array of structures can also be saved to disk. For more details see Object Lists.

Example

```
save_data_file:
	s = new structure
	s.testnumber = 1
	$s.teststring = "test 1"
	save "test.dat" object s

load_data_file:
	load "test.dat" structure s
	message s.testnumber
	message $s.teststring
```

Object arrays

An object array can be used to store a fixed number of objects. The following instruction can be used to create a new object array. The array can have one or more dimensions. Each dimension can have any number of items. When the object array is first created it will be filled with empty structures.

array variable **= new object [** *dimension 1* **][** *dimension 2* **] ...**

The position of an object in an array can be given using an index value for each dimension. Any position in the array can be assigned with an object. If the object that was overwritten has no other handles assigned to it then its resources will be freed.

array variable **[** *index 1* **][** *index 2* **] ... = @***object expression*

The term *array variable* **[** *index 1* **][** *index 2* **] ...** can then be used in any instruction that takes an object expression to get the object stored at that position in the array.

Array properties

The *variable***.max** property can be used in any floating point expression to return the number of dimensions in the array.

To return the size of each dimension use the function *variable* **(** *dimension* **)** in any numerical expression.

Finding an object

Objects can be assigned names. A name can be assigned to the "name" property of an object, for example: $*object***.name** = "some name". When you need to find an object in an array and you know the object has a unique name then you can use the following instruction.

object variable **=** *array variable* **find** "*object name*"

If the object array contains other object arrays or object lists then every object on every array and list will be searched for an object with the given name.

Example

The following example will create an array of 10 objects. Each position in the array will then be assigned with a structure. Then the name of each structure will be displayed in the result window.

```
my_object_array = new object[10]
for n = 0 to 9
    my_struct = new struct
    $my_struct.name = "structure ", n
    @my_object_array[n] = my_struct

for n = 0 to 9
    @my_struct = my_object_array[n]
    message $my_struct.name
```

Object lists

An object list can be used to store any number of objects. Objects can be added, inserted, removed, or gotten from a list. The following instruction can be used to create a new object list and assign it to a variable. When the list is first created it will be empty.

list variable **= new object list**

Arrays, structures, or any other type of object can be added to a list. As more objects are added to the list, the list will grow. When an object is removed from the list, the list will shrink. The following instructions can be used to add, get, insert and remove objects from a list.

list variable **push object** *object variable*

The pop instruction will remove an object from the top of the list.

object variable **=** *list variable* **pop**

The object at a given position can be returned using the get instruction. An object can also be returned by using the expression *list variable* **[** *position* **]**.

object variable **=** *list variable* **get** *position*

An object can be inserted at any position in the list. Any objects above that position will be shifted up to make a space.

list variable **insert object** *object variable* **at** *position*

The object stored at a given position in the list can be replaced with another object. If there are no other existing handles to the old object then its resources will be freed.

list variable **assign object** *object variable* **at** *position*

An object can be removed from the list. The list will be searched for occurrences of the given object. Every occurrence will be removed from the list. When an object is removed then every object above that position will be shifted down to fill the space.

list variable **remove object** *object variable*

An object can be removed from any position in the list. Any objects above that position will be shifted down to fill the space. If there are no other existing handles to the removed object then its resources will be freed.

list variable **remove** *position*

The following instruction will remove every item from the list. The list will become empty.

list variable **remove all**

The current number of items on a list can be obtained using the *list variable***.max** property.

List trees

Lists can be added to other lists to form trees of objects. As an example, if a number of object lists were placed on a main object list, then the main list could be called the root list, and the root list could be said to have a number of branches.

An object at any position on a single list can be returned by using the expression

list variable **[** *position* **]**

Any object in a tree of objects can be returned by using the expression

list variable **[** *position on root list* **] [** *position on branch* **] [** *position on branch of branch* **] …**

Finding an object

Objects can be assigned names. A name can be assigned to the "name" property of an object, for example: $*object*.**name** = "some name". When you need to find an object on a list and you know the object has a unique name then you can use the following instruction.

object variable **=** *list variable* **find** "*object name*"

If the object list contains other lists or object arrays then every object on every list and array will be searched for the given name.

List loops

A list loop can be used to count through every object in a list, returning a handle to each one as it goes. All following statements with a greater level of indentation will be included in the loop.

object variable **=** *list variable* **loop** *count variable* { **type** "*object type*" }{ **mode** *mode* }
    …

The variable will be assigned with the first object on the list and the count variable will be assigned with the value zero, then all following statements with greater indentation will be executed. The variable will then be assigned with next object on the list and the count variable will be incremented. The loop will continue until every object on the list has been counted.

If an object type is specified, then only objects of that type will be included in the loop. An optional mode value can also be given. If no mode value is given then the loop will start at the bottom and count to the top of the list, and only the items on the given list will be included.

If the mode is **LIST_TREE** and if the list contains other lists then every object in every list in the tree will also be included in the loop. If the mode is **LIST_DOWN** then the loop will start at the top of the list and count down. This mode should be chosen if you are removing objects while the list is counting. The two modes can be combined using the binary 'or' operator as LIST_TREE | LIST_DOWN.

Example

We will begin our example by creating some new lists. First we will create a list of strings.

```
my_string_list = new list                    // a new list of strings
my_string_array = new byte[10][20]           // a new array of strings
$my_string_array[0] = "zero";"one";"two";\
    "three";"four";"five";"six";"seven";\
    "eight";"nine"                           // assign 10 strings to the array

for n = 0 to 9
    $my_string = $my_string_array[n]             // add every string to the list
    my_string_list add my_string
```

Now we will create a list of structures.

```
my_struct_list = new list                    // a new list of structures
for n = 0 to 9
    my_structure = new struct                    // 10 new structures
    $my_structure.name = "struct",n
    my_structure.value = n
    my_structure.flag = TRUE
    my_struct_list add my_structure          // add each structure to the list
```

Now we will create a new list and add to it the list of strings and the list of structures.

```
my_everything_list = new list                    // make a complete list
my_everything_list add my_string_list            // add the list of strings
my_everything_list add my_struct_list            // add the list of structures
```

Now we have a couple of lists of objects to work with, and they are all stored on one list. Next we will display the names and types of the items on our main list.

```
message "Main object names and types"
item = my_everything_list loop n             // a list loop
   message $item.name,", ",$item.type            // display the name and type
```

Next we will display the names and types of the items on every list in our tree of lists.

```
message "All object names and types"
item = my_everything_list loop n mode LIST_TREE
    message $item.name,", ",$item.type
```

Next we can get a handle to an object with a particular name. In this case only the structures have been assigned names so we will grab one of them.

```
fifth_structure = my_everything_list find "struct5"
```

Finally we will get a handle to every item of a certain type on the list. In this case we will get all the strings. Strings have the type "Byte Array".

```
message "All strings"
string = my_everything_list loop n type "Byte Array" mode LIST_TREE
    message $string
```

Saving to disk

An array or list of structures can be saved to disk as a data file. Only the numbers and strings that were stored in each structure will be saved. The data file will be in the same format as a project file which can be viewed using any text editor.

save "*filename*" **object** *list handle*

A list of structures can then be loaded from disk. The following instruction will create a new object list and assign it to the given variable. The data from the given data file will be used to create structures which will be added to the list.

load "*filename*" **object list** *structure*

Example

```
save_list:
    l = new object list
    for n = 1 to 10
        s = new structure
        s.testnumber = n
        $s.teststring = "test ",n
        l push object s
    save "test.dat" object l

load_list:
    load "test.dat" list l
    for n = 0 to l.max-1
        s = l get n
        message s.testnumber
        message $s.teststring
```

Procedures

The best way to write a large program is to break it down into a number of smaller procedures.
A procedure can be defined by specifying a procedure name. All statements following that name with a greater level of indentation are included in the procedure.

procedure:
    ... the procedure starts here
    ...
    ... after executing the last line it returns to the calling instruction

A **call** statement can be used to jump to a procedure from any other place in the program. After the procedure has finished executing, the program will jump back to the line following the "call" statement.

call *procedure*

A procedure name can be made up of case sensitive letters, numbers and underscores, the first character must be a letter and the name must be followed by a colon. The procedure will begin running at the line following the procedure name. It will run until either the last indented line has been executed, or a **return** statement is found.

A procedure can be passed any number of parameters. The parameters can be listed after the "call" statement. An equal number of variable names must be listed after the procedure name to take the parameters. If a list of parameters is given then the procedure name must be followed by a colon.

call *procedure*: *parameter 1*, *parameter 2*, ...

procedure: *variable 1*, *variable 2*, ...
    ...

Parameters may be numbers, strings or objects. Parameters can usually be separated by commas, but a string parameter must be must be separated from the next parameter by a semicolon. If the parameter is an object then it must be preceded by an '**@**' sign.

call my_procedure: number, "string"; @object

The list of variables to take these parameters must all be given as variable names separated by commas.

my_procedure: my_number, my_string, my_object
    message my_number, ", ", $my_string, ", ", $my_object.name

A procedure can return any number of parameters. These parameters can be listed in a "return" statement. An equal number of variable names must be listed at the beginning of the "call" statement to take the returned parameters.

variable 1, *variable 2*, ... = **call** *procedure*

procedure:
    return *parameter 1*, *parameter 2*, ...

Return parameters may also be numbers, strings or objects and must follow the same rules as call parameters.

A **NULL** parameter can be passed to a procedure. If NULL is passed to a procedure then the receiving variable will not be assigned with anything. For example: call procedure: NULL.

Variables which are assigned with parameters are the same as any other variables which have been assigned. Once they have been assigned they are available to all procedures throughout the script.

Local procedures

Local procedures can have separate variables from the rest of the script. A local procedure can be passed parameters and return parameters, but any variables which are used inside the procedure will be completely separate from the rest of the script and will be discarded when the procedure ends. A local procedure cannot access the values of any outside variables. A local procedure can be declared using the following statement.

local *procedure*: *variable 1*, *variable 2*, **...**
...

Scripts

A script is an SXT, SXM or other text file containing a list of instruction statements. A script can be executed directly from the menu, or it can be run as part of a project. When a script is opened for execution, it is first checked for syntax errors, then it is compiled into a format which can be executed quickly by the computer.

The following instruction can be used to open an instance of a new script from the current script. The new script will be opened, checked and compiled. The given script variable will be assigned with the new script.

script variable **= new script** "*filename*"

Any number of scripts can be opened. Each script will have its own independent variables. The variables in one script will not affect the variables in another script. The variables in each script will be valid only for the life of that script.

Any procedure in any other script can be called from the current script.

script variable **call** *procedure***:**

The calling statement can pass any number of parameters and return any number of parameters.

*var 1***,** *var 2***, ... =** *script variable* **call** *procedure***:** *par 1***,** *@par 2***, ...**

You can get a handle to the current script using the following instruction.

script variable **= get script**

You can also get a handle to the main project script from any other script using the following instruction.

script variable **= project script**

Variables

The variables in a script are available to other scripts as properties of that script. You can use the following expression in any numerical expression to get the contents of a floating point variable.

*script variable***.***variable name*

You can use the following expression to get the contents of a string variable.

$*script variable***.***string variable*

You can also use the following instruction to get an object.

object variable **= @***script variable***.***object variable*

Multiple instances

The same script can be opened any number of times and assigned to different variables. Each instance of the script will have the same procedures but the variables in each one can have different values.

As an example, consider a script which is used to define a space invader. Any number of these scripts can be opened, one for each invader. Each invader will have a different set of attributes and this will be expressed by a different set values for its variables.

```
my_invaders = new list
for n = 1 to 10
    invader = new script "invader.sxt"
    my_invaders.add invader
```

Opening a second instance of the same script will be faster than the first because the instructions will already be checked and compiled.

Included scripts

A script can include any number of other scripts. These included scripts will be inserted into the current script. The procedures and variables of these other scripts will then be available to the current script. Scripts can be included using the following instruction.

include "*script filename*"

An included script will be loaded, compiled and added to the current script. This will be done while the current script is being compiled. It does not matter where this instruction is written, but it is conventional to declare included files at the top of the page.

All of an included script's procedures can be called directly by the current script. But the included script cannot directly call the current script's procedures. All common variable names will be shared by the scripts.

Inheritance

A special case arises when an included script has a procedure with the same name as a procedure in the current script. In this case, the current procedure will override the included procedure. Even if the procedure is called from the included script, the current script's procedure will be called.

An included script can include any number of other scripts and so on. If a chain of included scripts all have procedures with the same name then the procedure highest up in the chain will override them all.

The following instruction can be used to call an overridden procedure. It can be used by an overriding procedure to call the procedure it has overridden. An overridden procedure can also use this instruction to call the overridden procedure beneath it in the chain.

call include *procedure*
call include *procedure*: *parameter 1*, *parameter 2*, ...

If the current script includes more than one script which has a procedure with the same name as a procedure in the current script, and if the "call include" instruction is used then the result is undefined because there is more than one possible procedure to call. However, variable names will always be resolved correctly, regardless of where they are used..

Soft symbols

The symbols used for variable names must be single words made up of letters, numbers and underscores. Before a script is executed, each symbol used in the script is added to a symbol table. The compiled script then uses indexes to this table instead of words.

Soft symbols can be used to refer to a variable by name. A soft symbol is written as a string expression inside curly brackets {"*string expression*"}. The string is evaluated, and the variable's index is found by searching the symbol table for a matching word.

A soft symbol can be used instead of a fixed variable name in any variable assignment.

{ "*variable name*" **}** = numerical expression
${ "*variable name*" **}** = string expression
%{ "*variable name*" **}** = numerical or string expression

A soft symbol can also be used in any numerical or string expression. It is important to note that soft symbols can not be used instead of arrays because the symbol name must have already been defined in the script.

Example

```
x,y,z = 10,20,30
variable_names = new byte[3][2]
$variable_names[0] = "x"; "y"; "z"
for n = 0 to 2
    message "The value of ", $variable_names[n], " = ",   {$variable_names[n]}
```

Procedure names

An alternative way to specify a procedure name is to use a soft symbol, where the name of the procedure is given as a string.

call { "*procedure name*" **}**

The string expression is evaluated and is used to represent the symbol with the same name. If no symbols have been defined in the script with the same name then the soft symbol will generate an error.

Example

```
$my_procedures[n] = "example_procedure"
…

call { $my_procedures[n] }
…

example_procedure:
    …
```

Soft properties

The concept of soft symbols can also be applied to object properties. A soft property is when a property name is represented using a string.

*object***.{** "*property name*" **}**
$*object***.{** "*property name*" **}**

The following example can be used to enumerate all of the data in a structure using a property loop and a soft property.

```
$p = property my_structure loop n
     message n, " ", $p, " = ", $my_structure.{$p}
```

Projects, screens and elements

A script can be used to control the flow of a project. There is an option in the project options which can be used to choose the way the project will play. The default option is to show the project screen by screen. The first screen will be displayed and then when it is finished, the next screen will be displayed.

The other option is to control the project from the script. The project script will begin executing at the first line and then screens can be displayed and undisplayed as needed.

Showing the project screen by screen

If the project options are set to play a presentation screen by screen, then the following instruction can be used to jump to the screen with the given name. The screen jump will take place after the called procedure has returned. The current screen will end and then the given screen will be displayed as the next screen.

goto screen "*screen name*"

The following instruction can be used to display another screen over the top of the current screen. The first screen's elements will not be affected. Every element on the specified second screen will be displayed. If the second screen has its "Undisplay elements" option set, then its elements will be undisplayed before returning to the original screen.

overlay screen "*screen name*"

Controlling the project from the script

If the project options are set to control the presentation from the script then this instruction will display the specified screen and then return to the script. If the currently running script was called by a screen then the second screen will be displayed over the first screen.

display screen "*screen name*"

Displaying elements

Any element on the current screen can be displayed using the following instruction.

display element "*element name*"

Any element on any screen other than the current can also be displayed. First the screen must be selected. Then the desired element on that screen can be displayed.

select screen "*screen name*"
display element "*element name*"

Projects, screens and elements as objects

Projects, screens and elements are objects and they can be assigned to variables. You can get a handle to the current project, the selected screen or the element that called the current procedure using one of the following instructions.

my_project **= get project**
my_screen **= get screen**
my_element **= get element**

As an example, the following instructions will display the name of the currently displayed screen.

```
my_screen = get screen
message $my_screen.name
```

You can get the handle to any screen or element by name. The screen or element will be assigned to the given variable.

my_screen **= get screen** "*screen name*"
my_element **= get element** "*element name*"

If you need to get a handle to an element on another screen you must first select the screen, and then you can get the handle.

select screen "*screen name*"
my_element **= get element** "*element name*"

You can get a handle to any element that is currently displayed. It does not matter which screen or where the element came from. The list of visible elements will be searched and the first element with the given name will be assigned to the given variable.

my_element **= get display element** "*element name*"

You can test an element to see if it is currently visible. The function **visible** "*element name*" can be used in any floating point expression and will return 1 if an element with that name is currently visible or 0 if it is not.

A full list of instructions

The following instruction can be used to display an element with a given name. The currently selected screen will be searched for an element with this name. If the element is found it will be displayed.

display element "*element name*"

The following instruction can be used to display an element using an element variable. If an element is created using the script then this instruction should be used to display it.

display element *element variable*

The following instructions can be used to make the presentation jump to another screen.

goto screen "*screen name*"
goto screen *screen variable*

If the project options are set to play a presentation screen by screen, then the following instructions can be used to jump to the given screen. If the project options are set to control the presentation from the script then this instructions will display the specified screen and then return to the script.

display screen "*screen name*"
display screen *screen variable*

The following instructions can be used to display another screen over the top of the current screen. The overlayed screen will be displayed just like any other screen. When it is finished the presentation will go back to the other screen.

overlay screen "*screen name*"
overlay screen *screen variable*

The following instructions can be used to undisplay a given element.

undisplay element "*element name*"
undisplay element *element variable*

The following instruction can be used to undisplay all elements.

undisplay all

Every element from a particular screen can be undisplayed. Only elements that are listed on that screen's element list will be undisplayed.

undisplay screen "*screen name*"
undisplay screen *screen variable*

Buttons and dialog elements can be enabled and disabled using these instructions. If a button is disabled then it will not be clickable. If a dialog element is disabled then it will not accept any input. A state of TRUE will enable the element and FALSE will disable it.

enable element "*element name*" **mode** *state*
enable element *element variable* **mode** *state*
enable all mode *state*

Every element that was displayed from a particular screen can be enabled or disabled. Elements that were displayed from other screens will not be effected.

enable screen "*screen name*" **mode** *state*
enable screen *screen variable* **mode** *state*

If an animation or video has the "first frame only" function set then it will only start playing when the play command or instruction is executed. The following instructions can be used to play or stop playing an element.

play element "*element name*"
play element *element variable*
stop element "*element name*"
stop element *element variable*
stop all

The function **playing** "*element name*" can be used in any numerical expression and will return 1 if an element with that name is currently playing or 0 if it is not.

The dialog element that has the focus will be the one that takes input from the keyboard. The current focus can be set to the next dialog element in the list, the previous dialog element, or to any other specified dialog element. Resetting the focus will set it back to the main window.

set focus "*element name*"
next focus
previous focus
reset focus

The actions commands normally given in the element options can also be carried out by the script.

element action "*action commands*"

The following instruction can be used to update the screen. Any elements whose conditions are no longer valid will be undisplayed. Any elements whose conditions have become valid will be displayed.

update screen

Position

The position and size of an element can be changed using the following instruction.

element *element variable* **position** *left*,*top* **size** *width*,*height*

If an element has an area, then any position on the graphics window can be tested to see if it lies within the element's area. The state variable will be assigned with 1 if it is within the area, otherwise it will be assigned with 0.

state **= element** *element variable* **collision** *x*, *y*
state **= element** "*element name*" **collision** *x*, *y*

Properties

An element can be created dynamically from a script using the following instruction.

my_element = new element "*type*"

The type may be a "Background", "Rectangle", "Picture", "Text", "Animation", "Video", "Sound", "Text Button", "Picture Button", "Hot Area", "Hot Color", "Edit Box", "List Box", "Combo Box", "Input", "Wait", "Action", or any other type of element.

Once the element has been created you will need to give it some properties. The two most important properties of any object are its **name** and its **type**. The name can always be changed but the type is a permanent property which is given to it when it is created. Because these properties are strings they must be preceded with a '$' sign.

$*my_element***.**name **=** "*name*"

Other important properties are the element's position and size. There are four such properties: **xpos**, **ypos**, **xlen** and **ylen**. The position of the top left corner is xpos, ypos and the width and height are xlen, ylen. The position of an element can be set by assigning values to each of these properties or by using the following instruction.

element *my_element* **position** *left*,*top* **size** *width*,*height*

Another important property is an element's colors. Color is always given as red, green and blue values between 0 and 255. Different elements have different types of color such as text color, background color, transparency color, etc. The following instruction can be used to set any color property of an element.

element *my_element* **set** "*color_type*" **color** *red*, *green*, *blue*

If the given color type was "text" then the properties would be "text_red", "text_green" and "text_blue". Some other types of color property are "back" for background color, "hot" for hot color and "trans" for transparency color.

Some element properties are numbers and some are strings. Properties can be changed by assigning new values to them. String assignments must be preceded with a '$' sign.

my_element.*property* **=** *value*

$*my_element***.***property* **=** "*string*"

Special properties

There are some special properties which can be used to affect the behavior of an element. As an example, background elements such as Background and Rectangle elements have a **background** property which tells the system to remove all other elements when they are displayed.

Some elements, such as text and hypertext elements are drawn directly over the graphics window rather than into the graphics window memory. These elements will be redrawn after their area of the graphics window is updated for any reason. It is the **repaint** property that indicates an element should be redrawn.

The normal behavior of a Picture element is to preserve the area of the graphics window beneath it before it is displayed. Then when the Picture is undisplayed, that area of the window is restored. The undisplaying of an element can be disabled by setting the **no_undisplay** property.

Project files

Project files are saved to disk with the extension 'FGX'. These files are saved as simple text files which can be loaded into any text editor, such as the Formula Graphics text editor. A project file will have a structure which appears like

[Project]
property = *value*
...

[Screen]
property = *value*
...

[Element]
property = *value*
...

The properties of each project, screen and element will be listed in this file. You can find out the properties that are available for each type of element by examining a project file. Most of these properties can be changed while the project is playing. As an example, the picture element will have these properties.

$name = "*element name*"
$type = "Picture"
$condition = "*condition"*
xpos = *x position*
ypos = *y position*
xlen = *width*
ylen = *height*
wipe = "*transition*"
transparency = *0 or 1*
trans_red = *red transparency*
trans_green = *green transparency*
trans_blue = *blue transparency*

* If a script is used to modify the properties of any elements while a project is playing and if the project is later saved to disk then those changes to elements will be also saved.

Creating an element

An element can be created while the project is playing by using a script. In the following example we will create and display a new picture element.

```
my_picture = new element "Picture"
$my_picture.name = "example.gif"
my_picture.xpos = 100
my_picture.ypos = 80
display element my_picture
```

Care must be taken to ensure that all of the essential properties for the element have been given values. In particular, most elements require at least a position and size. Refer to the contents of an FGX file to see the essential properties for any type of element.

Screen loop

A screen loop can be used to count through every screen in a project, returning a handle to each one as it goes. All following statements with a greater level of indentation will be included in the loop.

screen **= screen loop** *count variable*
    …

The variable will be assigned with the first screen in the project and the count variable will be assigned with the value zero, then all following statements with greater indentation will be executed. The variable will then be assigned with next screen in the project and the count variable will be incremented. The loop will continue until every screen in the project has been counted.

Element loop

An element loop can be used to count through every element in a project, returning a handle to each one as it goes. All following statements with a greater level of indentation will be included in the loop.

element **= element** *screen* **loop** *count variable*
    …

The variable will be assigned with the first element on the screen and the count variable will be assigned with the value zero, then all following statements with greater indentation will be executed. The variable will then be assigned with next element on the screen and the count variable will be incremented. The loop will continue until every element on the screen has been counted.

Example

The following example will display the properties of every element on every screen.

```
s = screen loop a
    message "screen: ", $s.name
    e = element s loop b
        message "element: ", $e.name
        $p = property e loop c
            message "property: ", $p, " = ", $e.{$p}
```

Operators and functions

Floating point numbers use 32 bits: 1 for the sign, 8 for the exponent, and 23 for the numerical value. Their range is between +3.4e38 and -3.4e38 with at least 7 digits of precision.

For fractions less than one the decimal point must be preceded by a zero. For example 0.5

Operators

A wide range of arithmetic and logical operators are available for use in floating point expressions. Most of these operators are identical to those available in the C language. They are listed here in their order of precedence:

Increment **++**
Decrement **--**
Parenthesis **()**
Unary minus **-**
Boolean not **!** (returns zero if not zero, or one if zero)
Power operator **^** (value ^ power)
Round up to significant digits **~** (value ~ order of significance)
Round down to significant digits **_** (value _ order of significance)
Functions
Multiplication and Division ***** **/**
Floating point remainder **%**
Addition and subtraction **+** **-**

Binary shift **<<** **>>** (value >> shift bits)
Binary AND **&**
Binary OR **|**

Equal and not equal **==** **!=**
Magnitude operators **<** **>** **<=** **>=**
Boolean AND and OR **&&** **||**

* Care should be taken when using ++ and --. These operators will change the value of the variable before the expression is evaluated. They will always change the variable even if the instruction was conditional.

Functions

A wide range of arithmetic, trigonometric and logical functions are also available. Most of these are identical to those available in the C language:

abs positive value
mod gives 1 for +ve, 0 for 0, -1 for -ve
frac fractional component
ceil round up to an integer
floor round down to an integer
deg degrees to radians
sin sine
cos cosine
tan tangent
asin inverse sine
acos inverse cosine
atan inverse tangent
sqrt square root
log log base ten

ln natural log

Random numbers

The following function can be used in any numerical expression to generate a pseudo random number. The result will be a random floating point number between zero and the specified range value.

rnd *range*

The following instruction can be used to initialize the random number generator. To reinitialize the generator, use 1 as the seed argument. Any other value for seed sets the generator to a random starting point.

randomize *seed*

String functions

isdigit returns TRUE for characters '0' to '9'
isalpha returns TRUE for characters 'a' to 'z' or 'A' to 'Z'
tolower converts a character to lower case
toupper converts a character to upper case

strval "*string*" converts a string into a floating point value
strlen "*string*" returns the length of a string

Special functions

valid *variable name* returns TRUE if the variable name has been assigned an object.
visible "*element name*" returns TRUE if the element is currently visible or FALSE if it is not.
playing "*element name*" returns TRUE if the element is currently playing or FALSE if it is not.

Constants and system variables

These common constants have also been defined and can be used in any floating point expression:

ON 1
OFF 0
TRUE 1
FALSE 0
ERROR -1
PI 3.141592
PI_2 1.570796

TIME - returns a time string in the 24 hour format HH:MM zero padded.
DATE - returns a date string in the format DD/MM/YY zero padded.

TIMER - returns the number of hundredths of a second that have elapsed since the program began executing.

OPERATING_SYSTEM - returns a string describing the current operating system.

MIDI_DEVICE - returns TRUE if a MIDI device is available.
SOUND_DEVICE - returns TRUE if a sound device is available.
SOUND_SAMPLE - returns the current sample position of the currently playing sound, or -1 if no sound is playing.

Language Instructions

Bitmaps and palettes

A bitmap can be loaded from disk and assigned to a variable using the following instruction.

load "*bitmap name*" **bitmap** *my_bitmap*

Bitmaps have a number of properties. *my_bitmap*.**xlen** and *my_bitmap*.**ylen** can be used in any floating point expression to get the width and height of the bitmap. *my_bitmap*.**mode** can be used to get the color mode of the bitmap. This value may be **PAL**, **RGB16** or **RGB24** for 8, 16 and 24 bits of color.

A transparency color can be specified for the bitmap. This transparency will be used whenever the bitmap is displayed. Any pixels which are the same as the transparency color will not appear. If a transparency color is given to a 256 color bitmap then the palette index whose color is nearest to the given red, green and blue values will become transparent. For best results you should use a transparency color which is different from any other color in the bitmap.

bitmap *my_bitmap* **transparency** *red*, *green*, *blue*

A bitmap can be displayed in the graphics window. If the bitmap has no palette then it can be displayed directly using the following instruction. If the bitmap has a palette then the following instruction will display the palette as well.

display bitmap *my_bitmap* (**at** *x*, *y*)

If the bitmap's palette is not the same as the currently displayed palette and you want to remap the bitmaps colors to the current palette as it is displayed then you can use specify a display mode.

display bitmap *my_bitmap* (**at** *x*, *y*) (**mode** DISPLAY_REMAP)

If you need to display the bitmap many times such as in an animation and you cannot afford the delay caused by color remapping then you only need to remap the bitmap once before displaying it any numbers of times without remapping.

remap bitmap *my_bitmap*
display bitmap *my_bitmap* (**at** *x*, *y*) (**mode** DISPLAY_DIRECT)

The image in the graphics window can have a bitmap subtracted from it. All pixels in the graphics window which are the same color as the pixels in the bitmap will be set with the specified color. Only pixels which are different in color will be unchanged.

subtract bitmap *my_bitmap* **color** *red*, *green*, *blue* (**at** *x*, *y*)

The contents of the graphics window can be captured as a bitmap. If no area is specified then the whole graphics window will be captured.

capture bitmap *my_bitmap* (**area** *xpos*, *ypos*, *width*, *height*)(**mode** *mode*)

If the mode is **COPY** then the new bitmap will have the same color resolution as the graphics window. If the mode is **PAL** then the new bitmap will have a 256 color palette. This palette will be the same as the currently displayed palette. If the mode is **RGB16** or **RGB24** then the new bitmap will be 16 or 24 bit color.

A bitmap can be saved to disk with the following instruction. Any one of the documented file formats can be used.

save "*bitmap name*" **bitmap** *my_bitmap*

For more details see Bitmaps and Palettes.

Effects

A copy can be made of any object by using the following instruction. If a bitmap is copied then the new bitmap will have the same resolution, the same image and the same palette as the old bitmap.

new_bitmap **= copy** *old_bitmap*

A bitmap can be flipped along either its X or Y axis using the following instruction. The two possible modes are **FLIP_X** and **FLIP_Y**.

flip *mode* **bitmap** *bitmap variable*

Pixels

The color of any pixel in a bitmap can be found by treating the bitmap as an array of the form [xmax][ymax][3]. Regardless of the color mode of the bitmap the third dimension will have the red, green and blue values of the pixel. The following example shows how to display the color values of a pixel.

```
message "red = ", b[x][y][0], ", green = ", b[x][y][1], ", blue = ", b[x][y][2]
```

The color of a pixel can be changed with the following instruction.

my_bitmap[x][y][0] **=** *red*, *green*, *blue*

Palettes

If a bitmap has a palette then the palette can be found using the following instruction. The palette property must be preceded by a '@' sign to show that it is an object.

my_palette = @*my_bitmap*.**palette**

The palette object returned by this property will be a two dimensional byte array of the form [256][3]. The first dimension will be the 256 colors in the palette and the second dimension will be the red, green and blue values.

The palette of a bitmap can be loaded directly from disk into a byte array.

load "*bitmap name*" **palette** *my_palette*

You can then display the palette to the graphics window. If the video mode is only 256 colors then only the first 236 colors can be displayed from the palette. The other 20 colors will be reserved by Windows.

display palette *my_palette*

You can get the currently displayed palette from the graphics window. The colors will be copied into an array and the array will be assigned to the given variable.

capture palette *my_palette*

An area of the graphics window must be analyzed before an optimum palette can be found. If no area is specified then the entire window will be analyzed. A level of bias can be given towards a particular color group. The default bias is 10 towards white. The analysis data will accumulate each time this instruction is used. The data will be cleared when palette is finally optimized.

analyze { **area** *xpos*,*ypos*,*width*,*height* }{ **bias** *bias* **color** *red*,*green*,*blue* }

The following instruction will use the analysis data to find an optimum palette. The palette will only contain the given number of colors, all other colors will be set to black. The new palette can be made up entirely of the most popular colors. A optional number of foreground colors can also be specified. Foreground colors are a wide spectrum of colors to ensure that all color groups are represented.

optimize *colors* (**foreground** *foreground*)

Palette management

If a bitmap is to be displayed while a project is playing and you wish to use the palette management system to allocate colors in the system palette before the bitmap is displayed, then you can use the following instruction. This instruction will only allocate those colors actually used by the bitmap. An optional transition can also be given.

project bitmap *my_bitmap* (**trans** *r*, *g*, *b*) (**at** *x*, *y*) (**wipe** "*transition*")

If you want the palette management system to allocate the colors of a bitmap without displaying the bitmap then you can use the following instruction.

compose bitmap *my_bitmap*

The palette management system can also allocate an entire palette, or just a single color.

compose palette *my_palette*
compose color *red*, *green*, *blue*

The colors will remain allocated in the system palette until the script that composed them is freed, or until the following instruction is executed. This instruction will only free colors that were composed by this script.

decompose

Animations

Any frame of an animation can be loaded from disk and assigned to a variable as a bitmap object. If no optional frame number is specified then the first frame will be loaded.

load "*animation name*" **bitmap** *my_bitmap* { **frame** *frame_number* }

A bitmap can be saved as a new frame on the end of an animation file using the following instruction. An optional mode and background bitmap can be specified.

save "*animation name*" **bitmap** *my_bitmap* { **mode** *mode* }{ **background** *my_background* }

There are four possible modes of animation. These are **SAVE_SIMPLE**, **SAVE_DELTA**, **SAVE_OVERLAY** or **SAVE_SPRITE**. If the mode is overlay or sprite then the background bitmap must be specified.

For more details see Animations.

Example 1 - Converting an animation into bitmaps

The following example assumes that each frame of the AVI uses 24 bits of color. The frames are first

analyzed to build a color histogram with a bias towards white, then the optimum 256 color palette can be calculated. As each 256 color bitmap is captured from the 24 bit graphics window, its colors are remapped to the new palette.

```
for n = 0 to frame_count
    load "animation.avi" bitmap bmp24 frame n        // load a 24 bit frame
    display bitmap bmp24                             // display the frame
    analyze bias 10 color 255,255,255                // calculate color histogram
    optimize 256 foreground 64                       // get best 256 colors
    capture bitmap bmp8 mode PAL                     // convert to 256 colors
    save "bitmap", n, ".gif" bitmap bmp8             // save as a GIF
```

Example 2 - Converting bitmaps into an animation

Firstly every 5th bitmap is analyzed and the optimum 256 color palette is found for the entire animation. After the first frame has been saved with the new palette, then every other frame can be saved onto the end. Use a 24 bit graphics window.

```
for n = 0 to frame_count step 5                      // not every frame is analyzed
    load "bitmap",n,".bmp" bitmap bmp24              // load a bitmap
    display bitmap bmp24                             // display the bitmap
    analyze bias 10 color 255,255,255                // build color histogram

optimize 256 foreground 64                           // optimum palette for animation

for n = 0 to frame_count
    load "bitmap",n,".bmp" bitmap bmp24              // load a bitmap
    display bitmap bmp24                             // display the bitmap
    capture bitmap bmp8 mode PAL                     // convert to 256 colors
    save "animation.flc" bitmap bmp8                 // save to an FLC file
```

Graphics window

Information about the graphics window and video display mode are available from the following constants.

GRAPHICS_WINDOW - Windows handle for the graphics window (HWND)
GRAPHICS_WINDOW_XMAX - width of the graphics window (pixels)
GRAPHICS_ WINDOW_YMAX - height of the graphics window (pixels)
GRAPHICS_WINDOW_BITS - graphics window color resolution (8, 16 or 24 bits of color)
GRAPHICS_ DEVICE - Device context of the graphics window (HDC)
GRAPHICS_ DEVICE_XMAX - width of the video display (pixels)
GRAPHICS_ DEVICE_YMAX - height of the video display (pixels)
GRAPHICS_DEVICE_BITS - video mode color resolution (bits of color)

If the presentation is being played inside the environment then the following constant will return a handle to the graphics parent window. If the presentation is being played from the command line then it will return a handle to the application window.

APPLICATION_WINDOW - Windows handle for the application window (HWND)

The graphics window can be cleared. If no other color is specified then the window will be cleared to black.

clear window { **color** *red*, *green*, *blue* }

The visible state of the graphics window can be changed using the following instruction. Two possible modes are **SW_HIDE** and **SW_SHOW**.

show window mode *mode*

The image in the graphics window can be printed out to the Windows default printer.

print window

Update modes

When a bitmap is displayed, its pixels are first copied to the memory buffer of the graphics window. Then the contents of this buffer will be copied to the video card using the Windows display drivers. The bitmap will not actually appear on the video screen until it is copied to the video card.

You can control the way that areas of the buffer are copied to the video card by using the following instruction.

update mode *mode*

If the update mode is set to **OFF** then nothing will be copied to the video card. If the mode is set to **REFRESH**, then the area will always be copied to the video card. The default mode is refresh.

If the mode is **INVALIDATE** then the area will be marked as invalid but nothing will be copied to the video card. As more areas become invalid they will be combined into one total area. This total area can then be copied to the video card using the following instruction.

update window

An area of the graphics window can be updated by the following instruction. If the update mode is OFF then this instruction will be ignored. If the update mode is REFRESH then the area will be immediately copied to the video display. If the mode is INVALIDATE then the area will be marked as invalid.

update area *xpos,ypos,xlen,ylen*

Mouse cursor

The mouse cursor can be disabled. If the mode value is OFF then the mouse cursor will disappear from the display. If the mode value is ON then the mouse cursor will be shown.

mouse mode *mode*

The mouse cursor for the graphics window can be changed. The specified cursor will become the new project cursor until it is changed again. Some of the possible cursors are "Arrow", "Busy", "Circle", "Error" and "Finger".

mouse cursor "*cursor*"

The mouse cursor can be sent to any position in the graphics window.

mouse position 0,0

Defining the limits

The following function can be used in any numerical expression. It takes two parameters, an X and a Y value. If the given coordinate lies inside the graphics window then the function will return 1, otherwise it will return 0.

in_window *x*, *y*

The following instructions can be used to keep the values of the two variables within the limits of the graphics window. It assumes them to contain X and Y values. The first instruction will stop these values from going past the edges. The second instruction will wrap the variables so that if they move off one edge then they will come back at the opposite edge.

bound *x*, *y* **window**
wrap *x*, *y* **window**

Sprites

A sprite is an object which has a bitmap and a position. If a sprite is displayed at a position in the graphics window, and if it is later displayed again at another position, the old image will be smoothly replaced by the new image. Any number of sprites can be animated in the graphics window at the same time. The following instruction will create a new sprite and assign it to a variable.

sprite variable **= new sprite**

Before a sprite can be displayed, it must be given a bitmap and a position. It can also be given an optional depth value. If two or more sprites are displayed over the same area of the window then they will be appear in order of depth. The default depth value is zero. The greatest depth value will give the topmost image.

sprite *sprite variable* **bitmap** *array* **at** 0,0 (**depth** 0)

A transparency color can be specified for the bitmap. The transparency only needs to be specified once, usually after the bitmap has been loaded from disk.

bitmap *my_bitmap* **transparency** *red*, *green*, *blue*

If the bitmap is 256 colors then when it is displayed it will be displayed directly without any palette remapping. So the bitmap must have already been remapped to the current system palette. A bitmap only needs to be remapped once using the following instruction.

remap bitmap *my_bitmap*

Once all the sprites have been declared and assigned bitmaps and positions they can be displayed using the following instruction. This instruction will display every sprite in the system at the same time.

update sprites

The positions and bitmaps of the sprites can be changed before updating the sprites a second time. The positions and bitmaps can be changed again and again and sprites can be updated each time. The sprites should be updates about 12 times a second to achieve smooth animation.

Sprite properties

Sprites have properties which make them easier to manage. A sprite can be given a name just like any other object, **$***sprite***.name** = "*name*". The position of a sprite can be found using *sprite***.xpos** and *sprite***.ypos**. The width and height of a sprite can be found using *sprite***.xlen** and *sprite***.ylen**. The depth of a sprite is *sprite***.depth**.

Collision loops

A collision loop can be used to get the handle of every sprite that is currently in collision with a given sprite. All following statements with a greater level of indentation will be included in the loop. If there are no collisions then the lines in the loop will not be executed.

result variable **= sprite** *sprite variable* **collision** *mode* **loop** *count variable*
    …

The given sprite will be tested for collisions with other sprites. The first sprite found to be in collision will be assigned to the result variable and the count variable will be assigned with the value zero. Then all following statements with greater indentation will be executed. The result variable will then be assigned with next sprite in collision and the count variable will be incremented. The loop will continue until every

sprite in collision has been counted.

There are two modes of collision. SPRITE_EXTERNAL will test for any intersection of the two sprites. SPRITE_INTERNAL will test for one sprite fully engulfing the other.

Example

The following examples are extracts from the example game included with Formula Graphics.

```
update_sprite_time = TIMER
...
cannon_shot_sprite = new sprite
...
while ...
    ...
    if TIMER > update_sprite_time + 10     // update sprites 10 times per second
        update_sprite_time = TIMER
        ...
        sprite my_sprite bitmap my_bitmap at xpos-xmid, ypos-ymid depth ypos
        ...
        other_sprite = sprite cannon_shot_sprite collision SPRITE_EXTERNAL loop m
            if $other_sprite.name == "colored ball"
                pos = colored_ball_sprites find @other_sprite
                colored_ball_scripts remove pos
                colored_ball_sprites remove pos
                cannon_shot_script call explode
        ...
        update sprites
```

Splines

A spline can be used to find a curve which passes through a set of points. Any number of points can be given. Each point will have an X and a Y value. The X values must be a list of floating point values which start at a lower limit and increase to an upper limit. The Y values can be any floating point values.

A new spline object can be created using one of the following instructions. The first instruction takes a list of two or more points. The second instruction takes two floating point arrays and the number of elements in the arrays that are to be used.

spline handle **= new spline** *x1*, *y1*; *x2*, *y2*; **…**

spline handle **= new spline** *array1*, *array2* **size** *number of elements*

Once the spline object has been created, the expression *spline variable* **(** *x* **)** can then be used in any floating point expression to return a Y value for any given X value. The X value must be between the lower and upper limits. The Y value will lie on a smooth curved line which approximately passes through all the points in the list.

Example

The following example will initialize a spline using only three points. The points on the curve will then be displayed in the result window.

```
curve = new spline 0,0; 50,0; 100,100
for x = 0 to 100 step 10
    message "X value = ", x, ", Y value = ", curve(x)
```

The results will be:

```
X value = 0, Y value = 0
X value = 10, Y value = -7.085714
X value = 20, Y value = -10.97143
X value = 30, Y value = -11.31429
X value = 40, Y value = -7.771428
X value = 50, Y value = 0
X value = 60, Y value = 12.8
X value = 70, Y value = 30.4
X value = 80, Y value = 51.6
X value = 90, Y value = 75.2
X value = 100, Y value = 100
```

Polar coordinates

2D conversion

A 2D coordinate system normally uses X and Y values to give the position along the two axes. Another way to describe the position is to use a polar coordinate. A polar coordinate has an angle around the centre and a distance value.

The following instruction can be used to convert from normal coordinates to polar coordinates and back again. The angle will be taken clockwise around the centre with the coordinate (1, 0) having zero angle.

angle, *radius* **= polar** *x*, *y*
x, *y* **= angle** *angle* **radius** *radius*

3D conversion

A 3D coordinate system normally uses X, Y and Z values to give the position along the three axes. Using polar coordinates requires two angle values and a radius. The first angle will be the angle around the Y axis, and the second angle will be the angle between the horizontal plane and the Y axis. The coordinate (1, 0, 0) will have both angles equal to zero.

angle, *angle*, *radius* **= polar** *x*, *y*, *z*
x, *y*, *z* **= sphere** *angle*, *angle* **radius** *radius*

Sounds

If the **SOUND_DEVICE** constant is true, a sound device is present in the system, if it is false then there is no sound device. If the **MIDI_DEVICE** constant is true, a midi device is present in the system, if it is false then there is no midi device.

A sound object can be loaded from disk and assigned to a variable.

load "*filename*" **sound** *sound variable*

A sound object can be saved to disk.

save "*filename*" **sound** *sound variable*

Sounds have a number of properties. *my_sound***.samples** will return the number of samples, *my_sound***.channels** can be used to get the number of channels, *my_sound***.sample_rate** will return the number of samples per second and *my_sound***.sample_bits** return the bit size of each sample.

A common sound would have one or two channels, would play at 11025 or 22050 samples per second and have 8 or 16 bits per sample.

A sound object can be played by the system sound device. If no sound device exists, the instruction will be ignored. If any other sound is currently playing, the old sound will stop and the new sound will play.

play sound *sound variable*

If a sound is currently playing, the **SOUND_SAMPLE** variable can be used to find the current sound sample number. If no sound is playing, or the sound is finished then the value of this variable will be ERROR.

The following command will halt the execution of the program until the current sound has finished playing.

wait sound

The current sound can be stopped from playing.

stop sound

Files and directories

When a file name is given in an instruction it can be given as any string expression. If no path is specified then the file will be assumed to be in the project directory. Paths should be given in the form "*drive*:\\ *directory**filename*" where a backslash is written as '\\' instead of '\'.

A file can be copied from a source to a destination. If the file was compressed using the Microsoft "compress" utility then it can be expanded as it is copied.

copy "*filename*" **to** "*filename*"
expand "*filename*" **to** "*filename*"

The following instructions can be used to delete a file and rename a file.

delete "*filename*"
rename "*filename*" to "*filename*"

The following functions can be used in any numerical expression.

file_exist "*filename*" (true if the file exists)
file_length "*filename*" (the length of the file in bytes)
file_date "*filename*" (the time of creation in seconds since Jan 1970)

The following functions can be used in any string expression.

get_path "*filename*" (returns only the file path)
get_ext "*filename*" (returns only the file extension)
remove_path "*filename*" (returns the filename without a path)
remove_ext "*filename*" (returns a filename without an extension)

Directories

The following instruction can be used to create a new directory. The directory name must contain a drive letter. If the path has a number of subdirectories then these will also be created.

new directory "*directory*"

You can see if the instruction was successful by using the **dir_exist** "*directory*" function in any numerical expression to check the existence of the directory.

You can get the Windows directory or the Windows system directory using one of the following instructions.

$*directory* **= windows directory**
$*directory* **= system directory**

The current project directory is the directory that is searched for all files associated with elements, or any file which is referred to without specifying a path. This should be the directory that has the project file. You can get the current project directory or you can set it to a new location using one of the following instructions.

$*directory* **= project directory**
set project directory "*directory*"

You can also get the directory of the Formula Graphics executable file.

$*directory* **= formula directory**

The current working directory is the directory currently opened for use by the operating system. Formula Graphics does not use the current working directory but it may be used by other applications.

$*directory* **= current directory**
set current directory "*directory*"

The following instruction will copy the entire contents of one directory into another.

copy directory "*directory*" to "*directory*"

You can return the amount of disk space available on any drive by using the following function in any numerical expression.

disk_space "*directory*"

A directory loop can be used to get the name of every file matching a specification in a directory. The specification can have a path and a wildcard. All following statements with a greater level of indentation will be included in the loop. If there are no files found then the lines in the loop will not be executed.

$*filename* **= directory** "*file specification*" **loop** *count variable*
    …

The name of the first matching file will be assigned to the filename variable and the count variable will be assigned with zero. Then all following statements with greater indentation will be executed. The filename variable will then be assigned with next matching file name and the count variable will be incremented. The loop will continue until every matching file name has been counted.

Binary Files

The contents of any file can be loaded from disk and stored in a one dimensional byte array. The array will be allocated with enough space for the entire file. The contents of the file will be loaded into the array and the array will be assigned to the given variable.

load "*file name*" **byte** *array variable*

The entire contents of a byte array can be saved to a file. The file name can be given as any string expression.

save "*file name*" **byte** *array variable*

File and directory dialogs

A load file dialog box can be opened using the following instruction. The dialog box will have the given title. The chosen file name must already exist. Once a file name has been chosen it will be assigned to the given string variable. If the cancel button is pressed then the default filename will be assigned.

$*filename* **= dialog** "*dialog title*" **load** "*default file*"

Any number of files can be selected using the following instruction. A dialog box will open and a number of files can be selected by holding down the shift or control keys. The selected files will be returned in an array of strings. If the cancel button is pressed, the array will be empty.

result array **= dialog** "*dialog title*" **load multiple**

A save file dialog box can be opened using the following instruction. If the selected filename already exists, the user will be prompted to overwrite. Once a file name has been chosen it will be assigned to the given string variable. If the cancel button is pressed then the default filename will be assigned.

$*filename* **= dialog** "*dialog title*" **save** "*default file*"

A directory can be chosen using the following instruction. A directory selection dialog box will open with the given title.

$*filename* **= dialog** "*dialog title*" **directory** "*default directory*"

Operating system

The **VERSION** string constant can be used to determine the features available in the current player. This string will be of the form: **version 5.5 release 96.8.1 (32 bit)**.

The **OPERATING_SYSTEM** string constant can be used in any string expression to get the name of the current operating system. The return string will be **Windows 3.1**, **Windows 95** or **Windows NT**.

Any other application can be executed by the following command. The command line must contain the executable file name. If this file is not in the Windows search path then the full path must be included.

execute "*command line*"

The following instruction will execute any other application. Under Windows 3.1, if the application is already running then it will be reactivated. If the application cannot be found then a dialog box will prompt the user for the location of the file.

execute dialog "*command line*"

The application associated with a given document can be executed using the following instruction. The instruction will execute the application, then the application will be load and display the given document.

execute document "*document filename*"

As an example

execute document "myletter.doc"

would find that **.**doc files are associated with Word for Windows. Word for Windows would then execute and display the document.

Any Windows help file can be opened and displayed by the Windows help system. Each of the following instructions takes the name of the help file. The first instruction displays the table of contents. The second instruction opens a search window. The third instruction displays the topic referred to be the given topic number defined in the [MAP] section of the HPJ file.

help "*help file*"
help "*help file*" **search**
help "*help file*" **link** *topic number*

The **TIMER** value can be used in any numerical expression to return the number of hundredths of a second that have elapsed since the program began executing. This value can be used to create a delay or to synchronize events.

A delay can be specified in hundredths of a second. The following statement will stop the program for the given period of time. If the appropriate option is set then the delay can be stopped by pressing the space bar or Enter key.

delay *hundredths*

The following instruction will restart Windows. If any changes are made to the Windows configuration then Windows must be restarted before the changes can take effect.

restart system

Initialization files

Windows applications use initialization files in the Windows directory to store options and other important variables. Windows itself stores its configuration in the **win.ini** and **system.ini** files. You can use the following instructions to read or to change the contents of an initialization file.

set profile string "*filename*"**;** "*section*"**;** "*item*" **=** "*string*"

If the given filename does not exist, it will be created. If the section does not exist, it will be created. The name of the section is case independent; the string can be any combination of uppercase and lowercase letters.

$*string* **= profile string** "*filename*"**;** "*section*"**;** "*item*"

If the given filename does not contain a full path, the file will be expected to be in the Windows directory.

Dialog boxes

An input dialog box can be opened with a single line edit control. The dialog box will have the given title. The given string will be displayed in the edit control. If the OK button is pressed then the contents of the edit control will be assigned to the specified result variable.

$*result* **= dialog** "*title*" **string** "*string*"

The following instruction can be used to open a dialog box with the given title and message. The box can contain Yes, No, OK or Cancel buttons. The Yes or OK buttons will assign a value of 1 to the given result variable. Otherwise the variable will be assigned 0 or -1.

dialog "*title*" **ok** "*message*"
result **= dialog** "*title*" **yesno** "*message*"
result **= dialog** "*title*" **yesnocancel** "*message*"
result **= dialog** "*title*" **okcancel** "*message*"

Media control interface (MCI)

Microsoft Windows MCI (Multimedia Control Interface) can be used to play video and sound files as well as control CD's and VCR's. The following instruction can be used to execute an MCI command. A variable name can be specified to accept a return string.

{**$***result* **=** } **MCI** "*string*"

The following example uses the MCI to play an AVI animation with sound.

```
MCI "open example.avi alias example"
MCI "window example state show"
MCI "play example wait"
MCI "close example "
```

Reference

All commands accept the optional flags **wait** and **notify**. All commands, except **open** and **close**, also accept the optional flag **test**.

Commands

Open and close an MCI device: **open, close**
Obtain information about a device: **capability, info, status**
Configure a device or set operating parameters: **configure, set, setaudio, setvideo**
Control playback: **pause, play, resume, stop**
Control media position: **cue, seek, step**
Set signals: **signal**
Realize a palette: **realize**
Specify or obtain information about a window or display rectangle: **put, where, window**
Repaint the current frame into a display context: **update**

Selected command strings

open *devicename* **alias** *alias* **parent** *hwnd* **style** *stylevalue*
cue *devicename* **to** *position*
play *devicename* **from** *position* **to** *position* **fullscreen window**
pause *devicename*
stop *devicename*
close *devicename*

For more details on MCI see Microsoft multimedia documentation.

Dynamic Data Exchange (DDE)

Windows applications can communicate with one another using the DDE protocol. Formula Graphics can act as a DDE client by exchanging string data with a DDE server. Contact must be established using the appropriate server and topic names.

dde "*server*"**;** "*topic*" **execute** "*command*"
dde "*server*"**;** "*topic*"**;** "*item*" **string** "*data*"
$*result* **= dde** "*server*"**;** "*topic*"

By sending DDE commands to the program manager you can install a new icon. The following instruction will send a command to the program manager.

program manager "*command string*"

The following is an example of how to install an icon. For more information on DDE and program manager see the Microsoft Windows SDK documentation.

```
program manager "[CreateGroup(Example)]"          // create program group and icon
program manager "[ShowGroup(1)]"
program manager "[AddItem(",$destination,"\\example.exe,Example,",$iconfile,")]"
```

Dynamic link libraries (DLLs)

Dynamic link libraries (DLL's) can be used to extend the feature set of an application. Formula Graphics can load a DLL and execute a function within that DLL. The following instruction will load a DLL and assign a handle to the given variable. It must be a 16 bit DLL for the 16 bit version of Formula or a 32 bit DLL for the 32 bit version.

handle **= new dll** "*library* "

The following instruction will execute the function with the given name. If the function name cannot be found in the DLL then there will be an error. The function takes no parameters, but an optional integer value may be returned. The return integer will use 16 bits in the 16 bit version and 32 bits in the 32 bit version.

{ *result* **=** } **dll** *handle* **call** "*function name*"

The following instruction can be used to pass the graphics window to a DLL function. Two parameters will be passed, a far pointer to the device independent bitmap (DIB) of the graphics window, and the graphics window handle (HWND). The DLL must use the Pascal calling convention for 16 bits or the stdcall convention for 32 bits. An optional integer value may be returned.

{ *result* = } **dll** *handle* **call** "*function name*" **window**

The following instruction can be used to pass any number of parameters to a DLL function. The DLL must use the Pascal calling convention for 16 bits or the stdcall convention for 32 bits. An optional integer value may be returned.

{ *result* = } **dll** *handle* **call** "*function name*"**:** *parameter list*

Each parameter in the *parameter list* must have a type identifier followed by a number or object. Parameters must be separated by commas or semicolons. The parameters will appear as: *type value***;** *type value***; …**

TYPE_SHORT indicates that the value is a 16 bit integer and **TYPE_LONG** indicates a 32 bit integer. These types should be used to pass numbers to the function. Please note that standard C integers are short integers for 16 bit programs and long integers for 32 bits.

TYPE_PTR indicates a 32 bit pointer. This type should be used to pass a pointer to the data contained in an object such as an array or string.

Example 1

The Windows API consists of an wide range of system functions. These functions are stored in DLL's in the Windows system directory. The 16 bit API uses Pascal calling convention and the 32 bit API uses stdcall. This means that many of these functions can be called using Formula Graphics DLL instructions.

This example calls the "SetWindowText" function in the 16 bit "user.exe" DLL to change the title of the application window.

```
handle = new dll "user.exe"
hwnd = APPLICATION_WINDOW
dll handle call "SetWindowText": TYPE_SHORT hwnd; TYPE_PTR "New window title"
```

Example 2

This example uses drawing functions in the 16 bit "gdi.exe" DLL to draw a circle in the graphics window.

```
handle = new dll "gdi.exe"
hdc = GRAPHICS_DEVICE
brush = dll handle call "CreateSolidBrush": TYPE_LONG 16777215
oldbrush = dll handle call "SelectObject": TYPE_SHORT hdc, TYPE_SHORT brush
dll handle call "Ellipse": TYPE_SHORT hdc,\
    TYPE_SHORT 10, TYPE_SHORT 10, TYPE_SHORT 310, TYPE_SHORT 190
dll handle call "SelectObject": TYPE_SHORT hdc, TYPE_SHORT oldbrush
dll handle call "DeleteObject": TYPE_SHORT brush
```

All of the functions used in these examples are also available in 32 bit DLLs with different names. For example, the 32 bit version of "user.exe" is "user32.dll". For more information on Windows API functions, see the Microsoft Windows SDK documentation.

Object database connectivity (ODBC)

The ODBC (open database connectivity) standard is supported by most databases. With the help of an ODBC driver you can access the data stored in any type of database file. Formula Graphics provides a simple interface to databases using SQL (structured query language).

The following instructions can be used to open a database and assign it to a variable. The first instruction will open a dialog and you can choose a database from those installed on your system. The second instruction takes a specific database name. This will be the name assigned by the ODBC installation or administration software.

odbc handle **= new database dialog**
odbc handle **= new database** "*database*"

This next instruction can be used to send a query to the database. It can also be used to retrieve a table of data. The query can be any SQL statement supported by your ODBC driver.

{ *array variable* **=** } **database** *odbc handle* **query** "*query*" { **fieldwidth** *width* }

A variable name must be given if the instruction is to return a table of data. A three dimensional byte array will be allocated and assigned to this variable. The array will be of the form [row][column][field]. The data will be retrieved in text format and stored in this array.

The size of the first dimension will be equal to the number of rows returned. The size of the second dimension will be equal to the number of columns returned. You can get the number of rows as *array variable*(0) and you can get the number of columns as *array variable*(1).

You can choose the size of the third dimension using the field width specifier. The default size will be 32 bytes. The size of this dimension should be enough to store the longest string of data to be returned from the database, otherwise the data will be truncated.

Using ODBC

In order to use ODBC it must first be installed on your system. Microsoft provides a runtime version of ODBC which can be distributed under their license agreement. New operating systems come with built in ODBC support.

Before you can use a database file it must first be installed in your system. You can install a database using an ODBC administration utility. The Microsoft administration utility is called "odbcadm.exe". When you install a database file you must specify a database name, this is the name that you use in Formula Graphics to refer the database file.

Formula Graphics can be used to install the ODBC system and set up a database file. The following instruction will silently install an ODBC driver. Formula will use the Microsoft ODBC installer called "odbcinst.dll". It looks for the file called "odbc.inf" in the project directory. This information file contains a list of the file names to be installed.

install database "*driver*"

For more information refer to your ODBC driver documentation or the Microsoft ODBC SDK documentation.

ODBC Example

```
my_database = new database "My database"                    // open the database
```

```
$my_query = "SELECT * FROM my_table"              // return entire table
my_result = database my_database query $my_query                    // send SQL request

message "The number of rows are ", my_result(0)              // check result
message "The number of columns are ", my_result(1)
message "The first column of the first row is ", $my_result[0][0]

free my_database                                          // close the database
```

SQL examples

There is a lot of information available about programming in SQL. Using one simple SQL statement almost any subset of information can be returned from a database. One of the simplest SQL statements can be used to return all the rows and columns from one table.

SELECT * FROM *table name*

If you only wanted to get particular columns of data from the table then you can specify those column names.

SELECT *column name 1*, *column name 2*, **…** FROM *table name*

You can also ask for a range of data within certain conditions

SELECT * FROM *table name* WHERE *condition*

The *condition* specifier can compare the contents of any column with a value or a string. Comparisons can be combined using AND and OR operators. As an example:

first_column = some_value OR second column = "some_string"

Mouse and keyboard

Windows uses messages to inform applications about mouse and keyboard events. Most applications have a loop which reads these messages and responds to the ones that are important. When an event takes place, a message is generated and placed in a message queue. The application can then get the message from the queue and process it. A message contains a type number and two other numbers with additional information (wparam and lparam).

After executing an instruction, Formula Graphics will check the keyboard to see if the Escape key has been pressed. It checks the keyboard by getting the next message from the message queue. You can find out what the last keyboard or mouse message was by using one of the following system variables.

LAST_MESSAGE (last Windows message that was processed)
LAST_WPARAM (last wparam value for that message)
LAST_LPARAM (last lparam value for that message)
LAST_KEYPRESS (last key pressed down - wparam of a WM_KEYDOWN message)
LAST_MOUSE_XPOS (last mouse X position - loword of lparam for any mouse message)
LAST_MOUSE_YPOS (last mouse Y position - hiword of lparam for any mouse message)

Checking the keyboard after each instruction can be time consuming. It can also stop you from getting the next message yourself using a "peek" instruction. If you need all the speed you can get or if you use the "peek" instruction then you should disable keyboard checking with the following instruction. A mode value of zero will disable checking.

break mode *mode*

Remember that if you disable keyboard checking and your application is stuck in a loop then you will not be able to stop it and your system will lock up. The break mode will apply to all scripts, the default break mode is true (keyboard will be checked).

The following instruction can be used to get the next message from the Windows message queue. The message type will be assigned to the *message* variable. The wparam and the low word and high word of the lparam will also be assigned to variables.

peek message *message*, *wparam*, *loword*, *hiword*

A "peek" instruction does not wait for a message. If no message is available then the *message* variable will be assigned zero. In order for this instruction to work properly, keyboard checking must first be disabled with "break mode OFF".

If you wish to wait for the next message to be generated then you can use the "get" instruction.

get message *message*, *wparam*, *loword*, *hiword*

The following messages have been defined by Formula Graphics. Although there are many more Windows messages these are the only these messages that will be returned by the message instructions. For more information on messages see the Microsoft Windows SDK documentation.

WM_CHAR any key has been pressed
WM_KEYUP any key has been pushed down
WM_KEYDOWN any key has been released
WM_MOUSEMOVE the mouse has moved
WM_LBUTTONUP the left mouse button has been unclicked
WM_RBUTTONUP the right mouse button has been unclicked
WM_LBUTTONDOWN the left mouse button has been clicked
WM_RBUTTONDOWN the right mouse button has been clicked

WM_LBUTTONDBLCLK the left mouse button has been double clicked
WM_RBUTTONDBLCLK the right mouse button has been double clicked

The keyboard

If the message is WM_CHAR, WM_KEYUP or WM_KEYDOWN then the wparam will tell which key was pressed. Alphanumeric keys will return character values. The following are a few of the extended keyboard keys that have been defined. For a full list select Language from the Window menu.

VK_SPACE
VK_RETURN
VK_ESCAPE
VK_CONTROL
VK_LEFT
VK_UP
VK_RIGHT
VK_DOWN
VK_PAUSE

The following example will respond to certain key presses.

```
peekmessage m, w, l, h
    if m == WM_CHAR
        if   w == 'A'
             message "the letter 'A'   was pressed"
        if   w == VK_RETURN
             message "the 'Enter' key was pressed"
```

The mouse

If the message is WM_MOUSEMOVE, LBUTTONUP, LBUTTONDOWN or any other mouse message then the position of the mouse cursor can be found in the *loword* and *hiword* variables. The following example illustrates how to detect messages. For more information on Windows messages, see the Microsoft Windows 3.1 SDK documentation.

```
peekmessage m, w, x, y
    if m == WM_LBUTTONDOWN
        message "the left mouse button was pressed"
    if m == WM_MOUSEMOVE
        message "the mouse has moved to ", x, y
```

Printing

The entire contents of the graphics window can be printed out using the following instruction. The first instruction will use the default printer, the second instruction will open a printer selection dialog box.

print window
print window dialog

A hypertext document with full word processor formatting can be printed out using the following instruction.

print hypertext "*element*" **page** *page*

Printing forms

Formula Graphics can be used to print forms using any page layout. The following instructions will open a new document for printing. The first instruction will assigned the given variable with a handle to the Windows default printer. The second instruction will open a dialog and allow the user to select a printer.

printer variable **= new printer**
printer variable **= new printer dialog**

If the Cancel button is pressed and no printer is selected then any following instructions that use the handle will be ignored.

To design a form, you must first divide the page into rectangular areas. Inside each rectangle will be some text or graphics. If the rectangle contains text then only one font can be used for that entire rectangle. The following instructions are used to set the rectangle, set the font, and then fill the rectangle with text.

printer *printer variable* **area** *left*, *top*, *width*, *height*
printer *printer variable* **font** "*face name*" **size** *height* {, *weight* } {, *italic* }
printer *printer variable* **print** "*some text*" { **mode** *mode* }

All dimensions including the font height must be given in millimeters. The optional font weight value can be either **PRINT_NORMAL** or **PRINT_BOLD**. The optional italic value can be either TRUE or FALSE. The optional text printing mode can be **PRINT_CENTRE** or **PRINT_RIGHT** to centre or right align the text.

A black frame can be drawn around a rectangle. A bitmap can be printed inside a rectangle. A line can be drawn between any two points on the page.

printer *printer variable* **frame**
printer *printer variable* **bitmap** *bitmap handle*
printer *printer variable* **line** <x>,<y> to <x>,<y>

To get the width and height of the default printer paper in millimeters use the **PRINTER_XRES** and **PRINTER_YRES** system variables.

After each page of a document has been designed it can be sent to the printer using the following instruction. The page will be sent to the printer and a new page can then be designed, but the document won't be printed out until the printer handle is freed.

printer *printer variable* **page**
free *printer variable*

Example

The following example will print out a form with a title and sub title at the top, a body of text in the middle, a bitmap below it and a line across the bottom.

```
p = new printer                         // open the printer

printer p area 40, 20, 140, 10                  // print a title
printer p font "Arial" size 10, PRINT_BOLD
printer p print "TITLE" mode PRINT_CENTRE

printer p area 30, 40, 160, 7           // print a subtitle
printer p font "Arial" size 7
printer p print "Sub Title" mode PRINT_CENTRE

printer p area 20, 60, 180, 120
printer p frame
printer p area 21, 61, 178, 118
printer p font "Arial" size 3.5
printer p print $my_document                    // print some text

printer p area 60, 200, 100, 80
printer p bitmap my_bitmap              // print a bitmap

printer p page                          // finish the page
free p                                  // print the document
```

Fonts

The following instruction can be used to add another font to Windows list of available fonts. The instruction takes a font filename. The font file can be distributed along with your project and should be included separately in the project directory. The file may be a **.**FON font resource file, a **.**FNT raw bitmap font file, a **.**TTF raw TrueType file, or a **.**FOT TrueType resource file.

install font "*font filename*"

After each execution of this instruction, the font will only be available until the next time Windows is closed down.

Internet

The **SOCKETS** value can be used in any expression, this value will be true (will be non zero) if a sockets DLL is available. A sockets DLL is required before there can be any communication with the internet. All new operating systems have sockets.

The **server_exist** "server name" function can be used in any expression. The function will return true (a non zero value) if the given web server can be contacted. This instruction should be used test for a connection to the internet.

Email

Messages can be emailed across the internet using the following instruction. Messages will be sent using the SMPT protocol. The instruction must specify the email address of the sender, the email address of the receiver and the message to be sent.

email from "*sender*" **to** "*receiver*" **message** "*message*"

The given message should contain the following header information. This header information should then be followed by the text of the message. Each line in the header must use a separate line. The header and the body of the message should be separated by a blank line.

To: the receiver's real name
From: the sender's real name <and return email address>
Subject: the topic of the conversation
Date: today's date

Example

The following example sends an email message to an address. The header information must be included in the message. The backslashes '\' are a feature of the language which can be used to spread one instruction across multiple lines.

```
$my_message = "To: Harrow Media", CRLF,\
    "From: Harrow Software<formula@magna.com.au>", CRLF,\
    "Subject: Good tidings", CRLF,\
    "Date: ", DATE, CRLF,\
    "Good tidings to you my friends"

email from "formula@magna.com.au" to "harrow@magna.com.au" message $my_message
```

World Wide Web

Any instruction in the Formula Graphics language which takes a filename as a parameter will also take a world wide web path name. If a file name begins with "http://" then it will be expected to be found on the world wide web. For example the expression **file_exist** "*URL*" will return true if the given URL exists.

The following instruction can be used to download a file from a web server to a local drive. The file path name must begin with "http://". The downloaded file will not be cached. Remember to use double backslashes in the local filename.

download "*internet filename*" **to** "*local filename*"

The following instruction can be used to request a URL from a web server. If the request is for a file, then that file will be downloaded and stored in a byte array. The byte array will then be assigned to the given variable.

server_response **= get url** "*URL*"

The URL may also specify a CGI script on a UNIX server or an IDC file on a Windows NT server. The following example requests an IDC file. The NT server will run the IDC file, which sends a database request to an SQL server. The server will return a table of text data, with tabs between each column and a new line for each row. The response is then converted from a delimited text file into a three dimensional text array.

```
response = get url "http://www.database_server/request.idc#parameters"
array = parse database response delimiter "\t"; CRLF
```

The following instruction can be used to post information to a URL on a web server. The server's response string will be assigned to the given variable.

server_response **= post url** "*URL*" message &message

FTP

Some of the instructions in the Formula Graphics language which take filenames as parameters will also take FTP path names. If a file name begins with "ftp://" then it will be expected to be found on an ftp server.

The first step in the process of transferring a file using FTP is to log into the server with a username and password. The default username and password used by Formula Graphics is "anonymous" and "guest". The following instruction can be used to change the username and password used by Formula Graphics for all future FTP connections.

ftp "*username*"**;** "*password*"

The following instruction can be used to upload a file from a local drive to an FTP server. The destination filename must begin with "ftp://".

upload "*local filename*" **to** "*internet filename*"

The following instruction can be used to download a file from an ftp server to a local drive. The source filename must begin with "ftp://". The downloaded file will not be cached.

download "*internet filename*" **to** "*local filename*"

The following are examples of sending and receiving information to and from an FTP server. The first example downloads a binary or text file and stores it in a byte array. The second example stores the contents of a byte array to a file on an FTP server. The third example appends a string to the end of a file on an FTP server.

```
load "ftp://ftp.server.com/path/filename" byte my_byte_array

save "ftp://ftp.server.com/path/filename" byte my_byte_array

save "ftp://ftp.server.com/path/filename" string my_byte_array
```

3D Graphics system

Coordinates

Formula Graphics supports both left and right handed coordinate systems, left handed is the default. In a left handed coordinate system, the X axis runs from negative on the left to positive on the right, the Y axis runs from negative below to positive above and the Z axis runs from negative behind to positive in front. In a right handed system the Z axis runs from negative in front to positive behind.

Any coordinate in the system can be specified in terms of its position along the X, Y and Z axes respectively. The absolute size of an object is not important in Formula Graphics, a 10 unit object 10 units away will look the same as a 100 unit object 100 units away. Although It is recommended that the size of any object be greater than one.

All angle values must be specified in radians. For convenience, the **deg** function can be used in any expression to convert degrees to radians. In a left handed coordinate system, if you were at the negative end of an axis, facing towards the positive end, then a positive angle would rotate anticlockwise around the axis. A right handed system would rotate clockwise.

The pixel position of a visible coordinate in 3D space can be found using the following instruction

pixel *x variable*, *y variable* **= coordinate** *x*, *y*, *z*

The position list

Instructions such as **shift** and **rotate** are called position statements. When a position statement is executed, its details are placed on a position list. By executing a series of these instructions, the position list can be used to describe any position and direction in space.

As more position statements are executed, the position list will grow. The list will continue to grow even if a procedure is called, or a procedure in another script is called. But any shifts or rotates added to the list during a procedure will be taken off the list when the procedure returns. Similarly anything added to the list during a loop will be removed at the end of each iteration.

The shift instruction will shift the current position the given number of units in the direction of the specified axis. The axis specifier must be **x**, **y** or **z**. The position will be shifted relative to the current position.

shift *axis*, *length*

The rotate instruction will rotate the current position the given number of radians around the specified axis. The axis specifier must be **x**, **y** or **z**. The position will be rotated relative to the current position.

rotate *axis*, *length*

The position instruction will set the current position to the given coordinate. An optional mode value can be given. The default mode value of **POS_ABSOLUTE** will set the position with respect to coordinate 0,0,0. The **POS_RELATIVE** will set the position with respect to the current position.

position *x*, *y*, *z* { mode *mode* }

The direction instruction will set the current direction to point towards a given coordinate. An optional mode value can be given. The default mode value of **POS_ABSOLUTE** will set the direction to point to a coordinate given with respect to coordinate 0,0,0. The **POS_RELATIVE** will set the direction to point to a coordinate given with respect to the current position. The new direction will established by adding two rotate statements to the position list. A Y rotation and an X rotation.

direction *x*, *y*, *z* { mode *mode* }

The current position of the position list can be found using the following instruction. The X, Y and Z components of the position will be assigned to the given variables.

x variable, *y variable*, *z variable* **= get position**

The current direction of the position list can be found using the following instruction. The X, Y and Z components of the normalized direction vector will be assigned to the given variables.

x variable, *y variable*, *z variable* **= get direction**

The camera

The position and direction of the camera can be set to the current position and direction using the following instruction.

set camera

In the following example the position is set to 100, 300, -500 and the direction is rotated to the right 60 degrees and down 30 degrees. The camera is then fixed to this position and direction using the **set camera** statement.

```
set_camera_position:
    shift x, 100
    shift y, 300
    shift z, -500
    rotate x, deg 30
    rotate y, deg -60
    set camera
```

The order in which position statements are executed is important. The last position statement executed is actually the first to be applied to the camera. If the rotates were above the shifts then the position 100, 300, -500 would itself be rotated around the coordinate 0,0,0.

The projection of the camera can be set. The three values refer to the width and height of the view plane, and distance between the camera lens and the view plane. If the distance value is positive then the universe will be left handed, if it is negative then the universe will be right handed. The default values are 0.4, 0.4, 1.0.

camera projection *width*, *height*, *distance*

Materials

Every surface is made of some material. The color of the surface will be given to it by the material. A new material can be created using the following instruction. The red, green and blue components of the material's color must be specified. These values each have a range of 0 to 255.

material handle **= new material color** *red*, *green*, *blue*

Materials have a number of other properties. These properties can be used in any numerical expression or they can be assigned new values at any time. The color of the material is stored in stored in the red, green and blue properties with values between 0 and 255.

material handle.**red**
material handle.**green**

material handle.**blue**

Diffusion refers to the normal illumination of the surface by a light source. The default value is 1. When the diffusion value is 1 and when the light intensity is 1, the surface will be the specified color.

material handle.**diffusion**

Specularity refers to the reflection of light rays from the surface. A zero value will give a matte surface. The default value of 1 will give a glossy surface.

material handle.**specularity**

Shininess refers to the area over which reflection can be seen. A high value gives a pinpoint reflection. The typical range is 1 to 100 and the default value is 10.

material handle.**shininess**

Translucency refers to the ability to see through the surface. The default value of 0 will be totally opaque. A value of one will be transparent like glass.

material handle.**translucency**

Curved surfaces can be approximated using polygon meshes. In the photorealistic rendering mode, curved surfaces are interpolated so that the faces of the polygons appear to be one smooth surface. The smoothness value refers to the angle between polygons. If the angle between polygons is less than the given value then the surface will appear smooth. The default value of 1.5 radians is slightly less than 90 degrees so that the faces of a cube will not be affected.

material handle.**smoothness**

Polygons

The most convenient way to construct a 3D model is to use polygons. A polygon can have three or more vertices where each vertex is a coordinate in 3D space. The following instruction can be used to create a new polygon. The polygon will be constructed using the given material and then assigned to the specified polygon handle.

polygon handle = **new polygon** *x*,*y*,*z*; *x*,*y*,*z*; **... material** *material handle*

The number of vertices in a polygon can be found using the *polygon*.**vertices** property. You can get or set the X, Y and Z values of each vertex using the following instructions where *axis* is 0, 1 or 2 for X, Y or Z.

variable = *polygon* **[***vertex***][***axis***]**
polygon **[***vertex***][***axis***]** = *value*
polygon **[***vertex***][**0**]** = *x*, *y*, *z*

Rendering

The following instruction can be used to render a scene. If an optional object array or object list is specified then only polygons that appear on that list will be rendered, otherwise all polygons will be rendered, including polygons created by other scripts.

render { **mesh** *list* } **mode** *mode*

There are three modes available. The **RENDER_WIRE** mode will draw the polygons as wire frames. The **RENDER_QUICK** mode will draw the polygons with shading and texture maps. But texture maps will not

be shaded and hidden surfaces will not always appear to be correct. The **RENDER_PHOTO** mode can be used to draw photorealistic images. This mode has shading, texture mapping, translucency and polygon smoothing.

Example

```
call set_camera_position
call make_polygon
render mode RENDER_WIRE
return

set_camera_position:
    shift z,-50                    // move camera back to a good viewing distance
    set camera

make_polygon:
    my_material = new material color 200,200,200
    my_material.specularity = 0
    my_polygon = new polygon -10,-10,0; 10,-10,0; 10,10,0; -10,10,0 material my_material
```

Polygon meshes

A polygon mesh is a surface made up of connected polygons. A polygon mesh can be generated using mesh loop. Two loop variables can be given, each with starting and finishing values. As the two variables progress from start to finish they will generate the vertices of the polygon mesh.

Mesh loops use the variables 'x', 'y' and 'z'. After each iteration of the loop, the coordinate specified by these variables will become another vertex in the mesh. 'x', 'y' or 'z' may be used as the loop variables, or they may be assigned values which are dependent on some relationship with the loop variables.

The following instruction can be used to generate a mesh. The loop will contain all following statements with greater indentation. A new object list will be created and each new polygon will be placed on the list, then the list will be assigned to the given mesh variable.

mesh *list variable*: *u_var*, *v_var* = *u1*, *v1* **to** *u2*, *v2* (**div** *du*, *dv*) **material** *material handle*

A **div** (division) value can be given for each variable to indicate the number of steps between the starting and finishing values. For example if the starting value was zero, the finishing value was ten and the div value was five, then the values of the loop variable would be 0, 2, 4, 6, 8, 10. The default value of one means that only the starting and finishing values will be used for vertices.

Examples

As a simple example consider the generation of a rectangular plane in the X and Y axes with a width and height of 20 units. The division values of 2,2 will divide the mesh in half in both directions, so there will only be four polygons in the mesh.

```
call set_camera_position
call make_plane
render mode RENDER_WIRE
return

set_camera_position:
    shift z,-50                    // move camera back to a good viewing distance
    set camera
```

```
make_plane:
    my_material = new material color 200,200,200
    mesh my_plane: x, y = -10, -10 to 10, 10 div 2,2 material my_material
```

As a complex example consider the generation of a sphere with a radius of 100 units. P is used for the horizontal angle around the sphere. Q is used for the vertical angle down the sphere. The x, y and z variables have a relationship with p and q.

```
call set_camera_position
call make_sphere
render mode RENDER_WIRE
return

set_camera_position:
    shift z,-500
    set camera

make_sphere:
    m = new material color 200,0,200
    m.shininess = 4
    mesh my_sphere: p,q = -PI,-PI/2 to PI,PI/2 div 20,10 material m
        r = 100 * cos q
        y = 100 * sin q
        x = r * sin p
        z = r * cos p
```

Machined surfaces

A **machine** instruction can be included in a mesh loop. This instruction can be used to test the values of the loop variables. If the numerical expression given by this instruction is zero, then a section will be cut out of the mesh. The quality of the cut will depend upon the division values and the **sub** (subdivision) values of the loop.

mesh *mesh variable*: **...** (**div** *du*, *dv*) (**sub** *su*, *sv*) **...**

machine *condition*

Example

One of the many ways to generate a disk using a mesh loop would be to generate a square plane and then cut a circle out of the plane.

```
machine_disk:
    my_material = new material color 200,200,200
    mesh my_plane: x, y = -5, -5 to 5, 5 div 8,8 sub 8,8 material my_material
        machine x*x + y*y > 25
```

Solid blocks

Three loop variables can be used in a mesh loop to generate a volume of coordinates. Only the visible polygons on the outside of the volume will be added in the mesh.

mesh *mesh variable*: *x*,*y*,*z* **=** *x1*,*y1*,*z1* **to** *x2*,*y2*,*z2* (**div** *dx*,*dy*,*dz*)(**sub** *sx*,*sy*,*sz*) **material** *material handle*

In the following example, three variables are used to produce a 20 unit cube.

```
make_cube:
    my_material = new material color 200,200,200
    mesh my_cube: x,y,z = -10,-10,-10 to 10,10,10 material my_material
```

A **machine** instruction can also be included in a triple mesh loop. If the value given to this instruction is zero then a piece will be carved out of the block. The quality of the cut will depend upon the division and subdivision values of the loop.

Example

The following example uses a triple mesh loop to create a solid block. A sphere is then machined from the block.

```
make_sphere:
    my_material = new material color 200,200,200
    mesh my_sphere: x,y,z = -5,-5,-5 to 5,5,5 div 8,8,8 sub 8,8,8 material my_material
        machine x*x + y*y + z*z < 25
```

Lights

Ambient light is like background light which comes from no particular direction. The following instruction can be used to change the amount of ambient light in a scene. The red, green and blue values should be between 0 and 255. The default ambient light is 90, 90, 90.

ambient light *red*, *green*, *blue*

A new light source can be created at the current position using the following instruction. A handle to the light will be assigned to the given variable. An optional color can be given. The red, green and blue values are usually between 0 and 255, but higher values can be used to create brighter lights. The default color is 255, 255, 255.

light handle **= new light** { **color** *red*, *green*, *blue* }{ **radius** *radius* }{ **angle** *angle* }

If an optional radius value is given then the intensity of the light will be inversely proportional to the distance from the light source. At the given radius, the intensity of light will be equal to the programmed values. If an optional angle is specified then the light will be a spotlight pointing in the current direction. The angle value will control the width of the beam.

The following properties can be used in any numerical expression, or they can be assigned with new values.

light handle.**red**
light handle.**green**
light handle.**blue**
light handle.**radius**
light handle.**angle**

Example

As an example consider the 3D graph of a trigonometric function. The camera is above left of the graph looking down and a light is located directly above the graph.

```
call set_camera_position
call set_lights
call make_graph
```

```
render mode RENDER_QUICK
return

set_camera_position:
    position -15,15,-20
    direction 0,0,0
    set camera

set_lights:
    position 0,100,0
    my_light = new light

make_graph:
    my_material = new material color 200,0,200
    my_material.shininess = 4
    mesh my_graph: x,z = -10,-10 to 10,10 div 20,20 material my_material
        y = 3*cos(x/20*PI) * 3*cos(z/20*PI)
```

Atmosphere

Atmospheric attenuation can be used to make objects appear to fade into the distance. Fading will only effect objects further than the first specified radius value from the camera. Objects further than the second specified radius value will be completely faded to the given color. The red, green and blue must be between 0 and 255.

atmosphere radius *radius*, *radius* **color** *red*, *green*, *blue*

Texture maps

A texture map is a bitmap which is attached to the surface of a polygon. The colors of the bitmap will override the color of the polygon material. A texture can be applied to a polygon using the following instruction. If the bitmap has a transparency color then the polygon will be transparent in those areas.

texture *polygon handle* **bitmap** *bitmap handle*

The texture map will be attached so that the first vertex of the polygon will be mapped to the top left corner of the bitmap. The second vertex will be mapped to the top right and the third vertex will be mapped to the bottom right. The placing of the texture map on the polygon can be changed by assigning new positions to the vertices. The third and fourth components of a vertex are the u and v pixel positions of the texture map.

polygon **[***vertex***][**3**]** = *u*, *v*

A texture map can be applied to the whole surface of an object. The given object list can have polygons, polygon meshes or any other construction of polygons and lists. The following instructions will stretch the texture map to cover the full size of the object. The optional tile values will copy the bitmap the specified number of times in the x and y directions of the texture.

texture *object list* **bitmap** *bitmap handle* **planar** *axis* { **tile** *x*, *y* }
texture *object list* **bitmap** *bitmap handle* **cylinder** *axis* { **tile** *x*, *y* }
texture *object list* **bitmap** *bitmap handle* **sphere** *axis* { **tile** *x*, *y* }

Using planar mapping, the texture will be mapped to a plane passing through the object. The plane will be perpendicular to the specified axis. Using cylindrical mapping, the x direction of the texture will be wrapped around the object as if it were a cylinder, and the y direction of the texture will mapped to the specified axis of the object. Using spherical mapping, the y direction of the texture will also be wrapped in

the direction of the specified axis of the object.

Example

```
apply_texture:
    load "image.gif" bitmap my_bitmap
    texture my_sphere bitmap my_bitmap sphere y
```

Object position

Position statements can be used set the position and direction of a polygon. When a polygon is created, each vertex will be shifted and rotated by the contents of the position list before it is stored.

If an object is made up of several parts, then each part can be defined with respect to its own origin and the position list can be used to shift and rotate that part into the desired position. The following example shows how a propeller can be created by rotating each propeller blade by a multiple of 90 degrees.

```
make_propeller:
    propeller = new object list
    for n = 0 to 270 step 90
        rotate z, deg n
        blade = new polygon 0,0,0; 0,10,0; 2,8,1 material my_material
        propeller push @blade
```

Object hierarchy

If a 3D object has more than one part, then each part should be stored on an object list of its own. These part lists should then be placed on a parent list so that only one variable is needed to handle the complete object. If an object part can itself be broken down into smaller parts then an object part hierarchy can be defined by having smaller lists on bigger lists on even bigger lists.

Because objects can have names by assigning a string to the name property of the object, each part in the object part hierarchy can be given a name. There is an instruction available for finding any object of a given name anywhere in a list of objects. Using this instruction, any part on an object part hierarchy can be quickly found.

Example

```
$object.name = "name"
object = list variable find "name"
```

Moving Objects

The position list can also be used to change the position of an existing object. The following instruction will move the object to the position given by the position list. The object is given as an object list containing polygons and lights.

position *object list* (**mode** *mode*)

If the specified mode is POS_ABSOLUTE then the new position of the object will always be taken with respect to its original position. If the specified mode is POS_RELATIVE then the new position of the object will be taken relative to its last position.

Loading an object

A 3D model can be loaded from a 3DS binary file using the following instruction. A new object list will be

created and the components of the object will be placed on the list. The given material will be used for the object if no other materials are specified in the file. The object hierarchy will be preserved by using a separate list for each component, the lists will be stored in the same hierarchy and each list will have the same name as its component.

load "*file name*" **mesh** *list variable* **material** *material*

General Examples

```
////////////////////////////////////////////
// Formula Graphics Logo (3 balls animation)
//

delete "3dlogo.wdo"

call set_camera_position
call set_lights
call make_balls

for n = 0 to 11
        clear window
        rotate y, n / 6 * PI
        position mesh balls
        render mode RENDER_PHOTO
        capture bitmap b mode RGB16
        save "3dlogo.wdo" bitmap b mode SAVE_DELTA
return

set_camera_position:
        shift z,-800
        set camera

set_lights:
        position 1000,1000,0
        my_light = new light

make_balls:
        balls = new object list
        position -100,130,20
        m1 = new material color 250,0,0
        s1 = call make_sphere: @m1
        balls push @s1
        position 100,50,-50
        m2 = new material color 0,250,0
        s2 = call make_sphere: @m2
        balls push @s2
        position 0,-140,0
        m3 = new material color 0,0,250
        s3 = call make_sphere: @m3
        balls push @s3

make_sphere: m
        m.shininess = 4
        mesh my_sphere: p,q = -PI,-PI/2 to PI,PI/2 div 20,10 material m
                r = 100 * cos q
                y = 100 * sin q
```

```
            x = r * sin p
            z = r * cos p
return @my_sphere
```

Examples

Getting, storing and retrieving data from an edit box

As an example, consider a screen with an edit box. While the screen is displayed, the user may enter some data into the edit box. When the screen is finished we want to get the data and save it to disk. The next time the screen is displayed, we want to fill the edit box with the data that the user entered the last time.

There are many ways we can do this. In this example we will use a sophisticated method which does the job simply and effectively. The real beauty of this method is that it provides a general solution for an entire project.

Imagine we have a project with one screen. On this screen we have an edit box called "example_data". The name of the edit box is important in this example because we will also use this name as a field name for our data. The only requirements for the name are that it is relevant and that it is a single word (an underscore is used here to make it a single word).

Imagine we have a script called "example.sxt". This script has been assigned to be the main project script in the project options. The instructions in this script will be used to carry out the getting and setting of the data from the edit box and the loading and saving of the data from disk.

The first thing we want to do after the project begins is to load any data that we may have saved to disk the last time. So the first element on the screen will be an action element called "initialize data". In the "carry out these commands" box of the action element we can type

```
call load_data
```

This will call the following procedure in the script.

```
load_data:
    $my_data_file = "example.dat"
    if file_exist $data_file_name
        load $data_file_name structure my_data
    else
        my_data = new structure
        $my_data.example_data = ""
```

If the file "example.dat" exists then we can load it as a data structure. The structure will contain all of the data fields used by our project. If the file does not exist then we must create a new structure ourselves and set the data fields with default values. In this case our only data field is a string property called "example_data". Note that this is the same name as our edit box.

After we have loaded our data structure and displayed the edit box, we will need to call a procedure to set the contents of the edit box. We can add another action element to the screen which has the command

```
call set_data: "example_data"
```

The set_data procedure will look like:

```
set_data: elname
    editbox $elname = $my_data.{$elname}
```

This procedure will be passed a string which is both the name of the edit box and the name of the string property in the data structure. The reason we are passing the element name in rather than hard coding it is so that we can call this procedure for any edit box.

The bracket around the property name indicates that it is a soft property, where any string expression can

be used to specify the property we want to get. In this case the property will be $my_data.example_data.

When the screen is finished, just before the edit box is undisplayed, we want to get the contents of the edit box and store it in the data structure. Our edit box has an option in which we can specify a procedure name. We can set this procedure to be called when the edit box is undisplayed.

The procedure is passed two parameters, the name of the element, and the contents of the edit box. The following procedure will store the contents of the edit box into a string property with the same name as the edit box in the data structure.

```
get_data: elname, edtstr
    $my_data.{$elname} = $edtstr
    call save_data

save_data:
    save $my_data_file object my_data
```

After any information is added to the data structure, it will be saved to disk. This procedure could easily be elaborated so that it only saved the data if something changed, or only saved once at the end of the screen, instead of once for each element.