The Application Program's Role

Using the display controller to create and manage GEWorlds

The application program calls display controller routines to create and manage Graphic Elements worlds. Remember that "application program," as we are using it here, refers to any software module which creates and manipulates GEWorlds.

The application creates a Graphic Elements world and installs it into a window by calling

```
GEWorldPtr NewGEWorld(GrafPtr worldPort, Rect *worldRect,
                      Fixed scale, CTabHandle worldColors);
```

Where:

- worldPort is a pointer to the (existing) color GrafPort into which the new Graphic Elements world is to be installed.
- worldRect is the bounding rectangle of the new GEWorld in this GrafPort, at a scale of one-to-one.
- scale is the ratio of the actual GEWorld size and position to the size and position of worldRect, expressed as a fixed-point number.
- worldColors is nil (if the system palette is to be used) or is the handle to a color table suitable for use with an 8-bit-deep GWorld.

The worldRect parameter specifies the size and position of a new GEWorld when the pixels of the GEWorld are in one-to-one corresponedence with those of its worldPort. The scale parameter determines the *actual* position and sizeof the GEWorld within worldPort.

In most cases, GEWorlds are created at full size, using the pre-defined scale value of scaleOneToOne.

But, for example, if a window is shrunk to half its original size, the application program might react by re-creating the GEWorld contained in that window at half its original scale. This will automatically maintain the original geometric relationship between the world and its window, and will automatically scale all actions within the world to its new size.

NewGEWorld returns nil if it cannot obtain the memory space it needs for its components.

All of the Graphic Elements functions your application program will call to deal with this world and its elements require a GEWorldPtr as one of their parameters. If there is only one GEWorld in the window, as is usually the case, this pointer can conveniently be stored into the window's wRefCon field:

```
SetWRefCon(myWindow, (long) myGEWorld);
```

It can then be passed to Graphic Elements functions as

```
(GEWorldPtr) GetWRefCon(myWindow)
```

A newly created Graphic Elements world is inactive, and must be activated to be displayed on the screen. GEWorlds are activated and deactivated by calling:

```
void ActivateWorld(GEWorldPtr world, Boolean turnItOn);
```

Where:

- world is a GEWorld obtained by calling NewGEWorld(),
- turnItOn is true to activate world or false to deactivate it.

The application program should normally deactivate a GEWorld only when that world's window is hidden.

While a Graphic Elements world exists and is active, it requires attention from the application program at three well-defined times. First, the application must allow the world to update itself frequently, normally once every time through its main event loop. Second, the world must be redrawn whenever the application receives an update event for the window containing it. Finally, if the Graphic Elements in the world include any sensor-type elements (Graphic Elements which respond to the user's mouse clicks), the application must give them an opportunity to perform their actions whenever the user presses the mouse button in the window containing the world.

Once each time through its main event loop, the application program should call:

```
void DoWorldUpdate(GEWorldPtr world, Boolean invalidate);
```

Where:

- world is an active Graphic Elements world created by the application,
- invalidate is true if the entire world is to be redrawn or false if only changes since the last call to DoWorldUpdate() are to be drawn.

When the application calls DoWorldUpdate() periodically from its main event loop, invalidate should be false. When the application receives an update event for the window containing the Graphic Elements world, it should call DoWorldUpdate() with invalidate true so that the entire world will be redrawn.

NOTE!! When calling DoWorldUpdate() in response to an update event, call it after calling EndUpdate() for the window containing the Graphic Elements world. For example:

```
void DoUpdate(WindowPtr window)
{
      if (window == myWindow)
      {
            SetPort( (GrafPtr) window );
            BeginUpdate(window);
            //Other drawing code here…
            EndUpdate(window);
            DoWorldUpdate((GEWorldPtr)
                        GetWRefCon(myWindow), true);
      }
}
```

If a Graphic Elements world contains any sensors, they should be given an opportunity to act whenever 1) the application receives a mouseDown event for the window containing that world, and 2) the window part returned by FindWindow() equals inContent. The application handles this by calling:

```
Boolean MouseDownInSensor(GEWorldPtr world, Point gMousePt);
```

Where:

- world is a Graphic Elements world belonging to the window for which the mouseDown event was received,
- gMousePt is the point where the mouse button was pressed, in global coordinates as retrieved from the event record.

If MouseDownInSensor() returns true, one of the interactive Graphic Elements in world has completely handled the mouseDown event.

When the application program has completely finished using a GEWorld, the memory it occupies can be freed by calling

```
void DisposeGEWorld(GEWorldPtr world);
```

Where:

- world is the Graphic Elements world to be disposed of.

NOTE!! DisposeGEWorld() disposes of all Graphic Elements in a GEWorld. However, it does not automatically free any memory that the application may have allocated and assigned to the Graphic Element fields drawData or changeData. Individual Graphic Elements can automate the freeing of this

memory — see "Custom Cleanup Procedures" in the section "Graphic Element Dynamics".

GEWorld Standard Time

All time values in the Graphic Elements system are specified in milliseconds. In general, the display controller handles all scheduling and timing for a GEWorld automatically. However, the application can explicitly control the flow of time in a GEWorld. The display controller provides the routines used for this purpose.

Time in a GEWorld starts and stops automatically when the world is activated and deactivated. However, the application program can explicitly "freeze" and "unfreeze" all action within a Graphic Elements world by calling:

```
void StopGETimer(GEWorldPtr world);
void StartGETimer(GEWorldPtr world);
```

Where:

— world is the GEWorld for which time is to start or stop.

The application can find out "what time it is" in a GEWorld — how many milliseconds have elapsed since the world was created — by calling:

```
unsigned long CurrentGETime(GEWorldPtr world);
```

Where:

— world is the GEWorld in question.

The rate at which time passes in a Graphic Elements world is variable. This rate is represented by an unsigned 4-byte fixed-point number, with the implied hexadecimal point after the first two bytes. Thus a value of 1 is represented as 0x00010000, which is predefined in the Graphic Elements system as geTimerStdRate. When this rate is used, 1 millisecond passes in the Graphic Elements world for each millisecond that passes in the real world. A rate of 0x00020000 would cause time to pass twice as fast in the GEWorld as in the real world; a rate of 0x00008000, half as fast. The application program can get the current rate of a Graphic Elements world by calling:

```
unsigned long GetGETimerRate(GEWorldPtr world);
```

Where:

— world is the GEWorld in question.

The application can set a new time rate for a Graphic Elements world by calling:

```
void SetGETimerRate(GEWorldPtr world, unsigned long newRate);
```

Where:

— world is the GEWorld in question,

— newRate is a fixed-point number representing the new rate, as defined above.

Other display controller routines for manipulating GEWorlds

The display controller provides other functions that may be called by the application program under specific circumstances.

When it is created, a new Graphic Elements world has no specified maximum projection rate — new frames are drawn and copied to the screen every time one or more elements move or change. Under some circumstances, the application can control the amount of processor time used to maintain a Graphic Elements world by specifying a minimum projection interval.

The maximum projection rate (minimum projection interval) attainable under a given set of conditions depends on several factors. The speed of the processor and graphics hardware are of primary importance. The number of objects in a given world, the sizes of those objects, and the rates at which they must be redrawn because of movement, frame changes, etc. are also extremely important. Finally, other processing performed by the application on each iteration of its main event loop can consume amounts of time which may be difficult to quantity. For example, the amount of time consumed by a call to WaitNextEvent() can vary greatly, depending on what other application programs are running when it is made. Thus the minimum projection interval in a Graphic Elements world is only a minimum — there is no way for the world to “guarantee” that it will run at a certain speed.

The application program can retrieve the current minimum projection interval of any world by calling

```
short GetProjectionRate(GEWorldPtr world);
```

Where:

— world is the Graphic Elements world in question.

The application program can set a new minimum projection interval by calling:

```
void SetProjectionRate(GEWorldPtr world, short newMSPerFrame);
```

Where:

— world is the Graphic Elements world in question,

— newMSPerFrame is the new minimum projection interval in milliseconds.

Occasionally, an application program may need to move an entire Graphic Elements world in relation to its window. The program accomplishes this by calling

```
void MoveGEWorld(GEWorldPtr world, short dh, short dv);
```

Or

```
void MoveGEWorldTo(GEWorldPtr world, short h, short v);
```

Where:

— world is the Graphic Elements world being moved,

— for MoveGEWorld, dh and dv are the horizontal and vertical distances to move the GEWorld, or

— for MoveGEWorldTo, h and v are the new horizontal and vertical locations of the upper left-hand corner of the GEWorld, in window coordinates.

In initializing a GEWorld, it is often necessary to position newly-created elements in relation to elements which already exist. For example, the application program might use the topleft corner of an existing element, plus or minus some horizontal and vertical offset, as the starting position for a new Graphic Element.

However, the actual horizontal and vertical offset desired depend on the scale of the world. For example, if the current world is half its “nominal” size, the offsets actually used should be halved, also. The application program can modify values to reflect the current scale of the world by calling

```
short ScaleToWorld(GEWorldPtr world, short value);
```

Where:

— world is the Graphic Elements world in which value is to be used,

— value is the quantity to be scaled.

Multiple GEWorlds

Graphic Elements provides system-level support for handling multiple GEWorlds in the same application. These worlds may overlap and may be displayed in the same window or in different windows. Multiple-world support is designed to be substantially invisible to the application program. For each of the worlds in a given system of GEWorlds, the application calls

```
Boolean AddToWorldList(GEWorldPtr world);
```

Where:

— **world** is the GEWorld to be treated as part of the multiple-world system.

This function returns **false** if it is unable to add **world** to the world list. In practice, this should never happen.

The application program treats a world which has been added to the world list exactly the same as one which has not. However, the application programmer should be aware of two special considerations that apply to GEWorlds which have been added to the world list:

— A call to **UpdateGEWorld()** for a world on the world list actually updates all worlds on the list.

— For efficiency, all worlds on the list share a single timer, and time passes at the same rate in all of them. Thus a call to **SetGETimerRate()** for any world on the list will affect the "speed" of all of them, while **StopGETime()** and **StartGETime()** freeze and unfreeze time in all worlds on the list.

The application program may explicitly remove a world from the world list by calling

```
void DeleteFromWorldList(GEWorldPtr world);
```

Where:

— **world** is the GEWorld to be deleted from the list.

Normally, however, the world is automatically deleted from the list when it is disposed.

Compatibility with other graphics

For reasons of its own, the application program may wish to draw some other type of graphic, for example a QuickTime movie or a TextEdit record, over the top of a GEWorld. In these cases, it is essential to keep the Graphic Elements system from drawing into the affected portion of the world. The application assumes control of such areas by calling

```
void CoverRect(GEWorldPtr world, Rect *coveredRect);
void CoverRegion(GEWorldPtr world, RgnHandle coveredRgn);
```

Where:

- **world** is the GEWorld into which the application will draw;
- **coveredRect/coveredRgn** is a rectangle or arbitrarily-shaped region for which the application program is assuming graphic responsibility;

When it has finished drawing within the boundaries of a GEWorld and wants to return an area to the Graphic Elements system for management, the application program calls

```
void UncoverRect(GEWorldPtr world, Rect *rectToUncover);
void UncoverRegion(GEWorldPtr world, RgnHandle rgnToUncover);
```

Where:

- **world** is the GEWorld into which the application has drawn;
- **rectToUncover/rgnToUncover** is the rectangle or region that the application program wants Graphic Elements to resume drawing.

Alternatively, the application may return all “covered” areas in a GEWorld to the Graphic Elements system by calling

```
void UncoverWorld(GEWorldPtr world);
```

Where:

— **world** is the GEWorld into which the application has drawn.

Other possibilities

It is usually best to keep code which actually manipulates individual Graphic Elements separate from the main line of the application program (or other enclosing aoftware entity), for reasons of portability and maintainability. But the Graphic Elements system itself does not enforce any such limitations. The application is free, at any time, to explicitly do anything to any element that the element might do to itself. The complete list of possible actions depends on the type of Graphic Element in question — see the section “Standard Graphic Element Types” for examples.