Customizing Graphic Elements

The Graphic Elements system is so designed that anything that can be drawn on the screen can become a Graphic Element. The application program introduces a new kind of element into the system by providing a creation function and a rendering procedure for the element.

Graphic Elements also makes it possible for the application program to provide a custom load function for acquiring needed resources (e.g., graphic data from disk) for a given element or elements, and a custom bit-copy procedure for special offscreen-to-offscreen transfer requirements. Custom load functions and bit-copy procedures are mentioned here for completeness only, and will be discussed in this section only as they apply to the creation of custom Graphic Elements.

In order to create a new type of Graphic Element, the application program must provide two things. The first is a creation function which returns a GrafElPtr to an element of the new type. The second is a rendering procedure, which draws the element in the appropriate location on the display.

Creation Functions

The creation function for a custom Graphic Element has the general form

```
GrafElPtr CreateMyElement(GEWorldPtr world, OSType id,
                          short plane, <other parameters>);
```

Where

— **world** is the GEWorld for which the element is being created;

— **id** is the ID to be assigned to the object;

— **plane** is element's "height" above the background;

— **<other parameters>** are whatever is necessary for the specific type of Graphic Element being created.

At a minimum, the custom creation function must do the following three things:

1) Allocate memory for the element and link it into its GEWorld by calling the low-level Graphic Elements allocation function

```
GrafElPtr NewGrafElement(GEWorldPtr world, OSType id, short plane,
                  short elemSize, GraphicLoadFunc loadProc,
                  short  resStart, short nRsrcs);
```

Where:

— **world**, **id**, and **plane** have the meanings given above;

— **elemSize** is the number of bytes of memory to allocated for a Graphic Element of this type, as given by **sizeof(ThisElementType)**;

— **loadProc** is a pointer to a function which is responsible for acquiring any graphics data from mass storage and assembling them into an offscreen graphic image. For most Graphic Elements, this is either nil, in which case the default **GraphicLoadFunc** for the specified world is used, or **NoLoader**, a predefined “null loader function” for Graphic Elements which don't load any data from disk and have no “permanent” offscreen representation.

— **resStart** and **nRsrcs** are values passed to **loadProc**, nominally representing the number of the first resource and the number of successive resources (beginning with **resStart**) to be loaded.

2) Initialize the fields of the newly-created element as appropriate. In general, the custom creation function should set the following fields:

— **element->animationRect** is the total rectangle occupied by the Graphic Element, in the coordinates of its GEWorld.

— **element->flags** should be initialized to the value **geShown** if the element is initially visible, or 0 if it is initially invisible.

— **element->renderIt** should be set to point to the element's rendering procedure.

— **element->drawIt** should be set to point to the element's low-level bit-copy procedure, if any. For a custom Graphic Element, this is optional, since this procedure is only called by the rendering procedure.

— **element->drawData** should be set to nil.

— **element->changeIntrvl** should be set to 0.

— **element->lastChangeTime** should be set to **CurrentGETime(world)**.

Not all of these fields are applicable for all Graphic Elements. For example, if a given type of element has no low-level bit-copy procedure, **element->drawIt** and **element->drawData** are irrelevant to it. Similarly, if a particular sort of

Graphic Element will have no autochange procedure, the program need not initialize **element->lastChangeTime,** although **element->changeIntrvl** should still be initialized to 0 .

Naturally, all fields specific to the custom element in question should also be initialized at this point.

3) Call **ChangedRect(world, &newElement->animationRect)** so that the new element will be drawn on the next update cycle.

Rendering Procedures

A Graphic Elements rendering procedure has the prototype

```
pascal void (*RenderProc)(GrafElPtr element,
                                GWorldPtr destGWorld);
```

Where:

- **element** is a pointer to the Graphic Element to be drawn;
- **destGWorld** is a pointer to the GWorld into which it will be drawn. Note that this is a standard offscreen GWorld, **not** a GEWorld.

On entry, the graphics environment is already set up so that a rendering procedure can draw into it using any and all QuickDraw calls or by directly altering pixels in **destGWorld**.

- as mentioned above, **element->animationRect** is the total rectangle occupied by this Graphic Element. The rendering procedure should not draw outside this rectangle.
- **element->drawRect** is the portion of **element->animationRect** which needs to be drawn in the current cycle.

Note that useful effects can sometimes be obtained by using a rendering procedure which calls the standard rendering procedure for a given type of Graphic Element, then performs some postprocessing. As an oversimplified example, here is a rendering procedure which will superimpose the frame number over a Frame Sequence Graphic:

```
pascal void RenderFrameWithNum(GrafElPtr graphic, GWorldPtr destGWorld)
{
      SeqGraphicPtr anim = (SeqGraphicPtr) graphic;

      RenderFrameGraphic(graphic, destGWorld);
      MoveTo(graphic->animationRect.left + 2,
                        graphic->animationRect.top + 8);
```

```
	// Obviously, this only works for 9 frames or fewer
	DrawChar('0' + anim->currentFrame);
}
```

Example 1: The Simplest Graphic Element

The null Graphic Element is the simplest element possible. It doesn't actually do anything except take up space in its world. The null Graphic Element's creation function looks like this:

```
GrafElPtr NewNullElement(GEWorldPtr world, OSType id, short plane,
                                                      Rect *animRect)
{
	GrafElPtr   element;

	element = NewGrafElement(world, id, plane,
                    sizeof(GrafElement), NoLoader, 0, 0);
	if (element != nil) {
		element->animationRect = *animRect;
		element->flags = geShown;
		element->renderIt = RenderNullElement;
		element->drawIt = nil;
		element->drawData = nil;
		element->changeIntrvl = 0;
	}
	return element;
}
```

As this function shows, the animationRect is the *only* relevant part of a null Graphic Element. The null element's rendering procedure is even simpler:

```
pascal void RenderNullElement(GrafElPtr element, GWorldPtr destGWorld)
{
	// For debugging, could do
	// FrameRect(&element->animationRect);
}
```

These few lines of code suffice to define a complete Graphic Element. Although it is invisible and doesn't do anything, it is “there”.

Such an element has many uses. For example, when connected to a collision procedure, it can be used to detect and react to the entry of another Graphic Element into a given area of the screen. In conjunction with a tracking function which just “eats” the mouse-down, it can be used to “disable” another sensor element on a “lower” plane. As the subordinate of another Graphic element, it can be used to alter the area or plane of that element's collision sensitivity.

Example 2: A Filled Rectangle

A Graphic Element which consists only of a filled rectangle is almost as simple as the null element. However, it needs an additional data field, to hold the specification for its fill color. We will define a new type for it, with the fill-color field added at the end of the basic Graphic Element:

```
typedef struct {
      GrafElement       baseGraphic;
      RGBColor          rectColor;        //Fill color for rect
} FRectGraphic, *FRectGraphicPtr;
```

Now we can define a creation function for an **FRectGraphic** as follows:

```
GrafElPtr NewFRect(GEWorldPtr world, OSType id, short plane,
                                    Rect destRect, RGBColor fillColor)
{
      GrafElPtr   element;

      element = NewGrafElement(world, id, plane,
                        sizeof(FRectGraphic), NoLoader, 0, 0);
      if (element) {
            element->flags = geShown;
            element->renderIt = RenderFRect;
            element->drawIt = nil;
            element->animationRect = destRect;
            ((FRectGraphicPtr) element)->rectColor = fillColor;
            ChangedRect(world, &element->animationRect);
      }
      return element;
}
```

The rendering procedure for an FRectGraphic is equally simple:

```
pascal void RenderFRect(GrafElPtr graphic, GWorldPtr destGWorld)
{
      RGBColor    saveColor;

      GetForeColor(&saveColor);

      RGBForeColor(&((FRectGraphicPtr) graphic)->rectColor);
      FillRect(&graphic->drawRect, (ConstPatternParam) &qd.black);

      RGBForeColor(&saveColor);
}
```

Note that our rendering procedure acts only on **graphic->drawRect**, which is the portion of **graphic->animationRect** that needs to be drawn on a given cycle. Note also that it saves and restores the foreground color — a Graphic Element's rendering procedure should always restore the graphic environment to the state it was in upon entry.

Customized Sensors — Tracking Functions

The tracking function of a sensor-type Graphic Element is called when the user pushes the mouse button in the animation rectangle of the sensor element. In general, it: 1) tracks the position of the mouse until the button is released; 2) alters the appearance of the sensor as appropriate to reflect the current mouse position; and 3) calls the sensor's action procedure as needed.

A Graphic Elements tracking function has the prototype

```
pascal Boolean (*SensorTrack)(GEWorldPtr world, GrafElPtr sensor);
```

Where:

- **world** is the GEWorld containing the sensor which has been activated;
- **sensor** is the sensor which is being tracked.

A tracking function returns **true** to indicate that it has handled the mouseDown event.

A "generic" tracking procedure behaves as follows:

```
{
      initial state = sensor's state;
      previous state = initial state;

      while (mouse button is down) {
            get current mouse location;
            new state =  (convert mouse location to sensor state);
            if (new state != previous state) {
                  change graphic to match changed state;
                  if (this is a continuous sensor such as a slider)
                        call sensor's action procedure;
                  update world;
                  previous state = new state;
            }
      }

      if (this is a non-continuous sensor such as a button)
            if (previous state != initial state)
                  call sensor's action procedure;
      sensor's state = previous state;
}
```

Of course, real tracking functions can vary greatly in complexity. For example, for the use of a null Graphic Element mentioned above, as a "disabler" for another sensor-type element, the complete tracking function is:

```
pascal Boolean TrackDisabler(GEWorldPtr world, GrafElPtr disabler)
{
      return true;
```

```
}
```

A more realistic example is the following tracking function for a “pushbutton” sensor. In this example, the “button” is a Frame Sequence Element with two frames, respectively representing the “off” and “on” positions of the sensor. While the mouse button is down, the sensor is “on” if the cursor is within its **animationRect**, “off” otherwise. If the mouse button is released while the sensor is “on”, it calls its action procedure if it has one and resets the sensor to its “off” state.

```
pascal Boolean TrackButtonSensor(GEWorldPtr world, GrafElPtr sensor)
{
	Point localPt;
	Boolean buttonOn = false;

	while (StillDown()) {
		GetMouse(&localPt);
		if (PtInRect(localPt, &sensor->animationRect)) {
			//Button goes on if not already on
			if (!buttonOn) {
				SetFrame(world, sensor->objectID, 2);
				buttonOn = true;
			}
		}
		else {
			//Button goes off if not already off
			if (buttonOn)
				SetFrame(world, sensor->objectID, 1);
				buttonOn = false;
		}
		DoWorldUpdate(world, false);
	}
	if (buttonOn)
	{
		if (sensor->actionProc)
			(sensor->actionProc)(world, sensor, sensorOn);
		SetFrame(world, sensor->objectID, 1);
	}
	return true;
}
```