Graphic Element Dynamics

This section will discuss attributes of Graphic Elements which are common to all types of elements. The next section, "Standard Graphic Element Types," will discuss the specific types of Graphic Elements which are provided with the Graphic Elements system.

Regardless of its role in the application program, every graphic element has four required components:

— **Its ID:** Each Graphic Element is assigned a unique four-character identifier. This ID is specified by the application program when it creates the element, and it used for all subsequent access to and management of that element. If element is a pointer to an individual Graphic Element, element–>objectID is the element's ID.

— **Its location:** Each Graphic Element occupies a rectangular area in the GEWorld of which it is a member. If element is a pointer to a Graphic Element, element->animationRect is this location. In contexts where an element's location is considered as a point, the top-left corner of this rectangle is used.

— **Its plane:** Each Graphic Element exists in a certain "plane" above its background. This plane is represented by an integer between 0 and 32767. Elements with higher plane numbers will be drawn after elements with lower plane numbers, and will therefore obscure them partially or totally. If element is a pointer to a Graphic Element, element->drawPlane is the element's plane.

— **Its rendering procedure:** Each Graphic Element has a pointer to a procedure which is responsible for drawing all or part of that element, as requested by the display controller, into an environment provided by the display controller. If element is a pointer to a Graphic Element, element->renderIt is a pointer to the element's rendering procedure. The Graphic Elements system provides rendering procedures for all its standard element types. Rendering procedures will be discussed in depth in the section "Customizing Graphic Elements."

Creation and Disposal

Each type of Graphic Element has its own creation routine, which allocates

memory for the element and links it into the GEWorld's list by calling the low-level Display Controller routine **NewGrafElement()**, then initializes the element's fields as needed for the specific element type.

An individual Graphic Element of any type (along with any elements subordinate to it) can be disposed by calling

```
void DisposeGrafElement(GEWorldPtr world, OSType objectID);
```

Where:

- **world** is the GEWorld which contains the element being disposed;
- **objectID** is the four-character "name" of the element.

This frees all memory occupied by the Graphic Element and its various parts. **DisposeGrafElement()** recursively disposes all subordinate elements, i.e., the list of Graphic Elements beginning in this element's **slaveGrafEl** field and continuing through the **slaveGrafEl** fields of its subordinate elements. It also calls the custom cleanup procedure, if any, of each element disposed.

Adding Action

For any Graphic Element, regardless of its type, the display controller provides routines for controlling visibility, for movement, and for changing planes. With one exception, these routines can be called from anywhere: the application program, the autochange or collision procedures of the element itself, or the autochange or collision procedures of another Graphic Element. The single exception concerns visibility: once an element has been made invisible, by its own action or that of another Graphic Element, it cannot make itself visible again. The display controller "ignores" invisible elements, and does not call any of their procedures.

The following routine makes a Graphic Element visible or invisible:

```
void ShowElement(GEWorldPtr world, OSType elementID,
                                        Boolean showIt);
```

Where:

- world is the GEWorld containing the element,
- elementID is the unique four-character ID assigned to the element,
- showIt is true to make the element visible, false to make it invisible.

The following function returns true if a Graphic Element is presently visible, false if it is invisible or does not exist:

```
Boolean ElementVisible(GEWorldPtr world, OSType elementID);
```

Where:

— **world** is the GEWorld containing the element,

— **elementID** is the four-character ID assigned to the element.

The following routines move a Graphic Element, either to a new absolute position within its GEWorld or relative to its current position:

```
void MoveElement(GEWorldPtr world, OSType elementID,
                                        short h, short v);
void PtrMoveElement(GEWorldPtr world, GrafElPtr element,
                                        short h, short v);

void MoveElementTo(GEWorldPtr world, OSType elementID,
                                        short h, short v);
void PtrMoveElementTo(GEWorldPtr world, GrafElPtr element,
                                short h, short v, Boolean doScale);
```

Where:

— world is the GEWorld containing the element,

— elementID is the unique four-character ID assigned to the element, or (in the Ptr… versions) element is a pointer to the element.

— for MoveElement() and PtrMoveElement(), h and v are the horizontal and vertical distances to move the element; for MoveElementTo() and PtrMoveElementTo(), they are the new horizontal and vertical position of the element (i.e., the top-left corner of the element's animationRect) in the coordinate system of the element's GEWorld.

— in PtrMoveElementTo(), doScale is true if h and v are relative to current scale of world and false if they represent absolute numbers of pixels from its topleft corner.

The Ptr… versions of these move routines are provided for use in situations such as an element's autochange procedure, where a pointer to the element is already available.

PtrMoveElementTo() is the only movement routine that is able to disregard a world's current scale and to position elements at absolute locations within it. The application program can use this capability, for example, to position one Graphic Element in a fixed relation to the animationRect of another element.

The following routine changes the plane of a Graphic Element. It can be used, for example, to make one element "go behind" another one:

```
void SetElementPlane(GEWorldPtr world,
                                OSType elementID, short newPlane);
```

Where:

- world is the GEWorld containing the element,
- elementID is the unique four-character ID assigned to the element,
- newPlane is the new plane for the Graphic Element.

Optional Components

For some Graphic Elements — for example, a bush in the foreground of a game, or a meter readout which is run directly by the application program — the four required components are sufficient. But most "interesting" Graphic elements possess one or more optional components, which permit them to react to the passage of time, to contact with another Graphic element, and to the actions of the application's user:

- **The autochange procedure:** Each Graphic Element may have an autochange procedure, in conjunction with an autochange interval. This procedure is called automatically by the display controller as it creates updated frames for the screen display. Each element also has space for a pointer to additional data which may be used as desired by its autochange procedure. If element is a pointer to a Graphic Element, its autochange procedure is element->changeIt, its autochange interval is element->changeIntrvl, and its autochange data pointer is element->changeData.

- **The collision procedure:** Each Graphic Element may have a collision procedure and a collision plane. An element's collision procedure is called whenever it moves into contact with another Graphic Element, and the other element's plane is equal to the "colliding" element's collision plane. If element is a pointer to a Graphic Element, its collision procedure is element->doCollision, and its collision plane is element->collisionPlane.

- **The interaction procedures:** Any Graphic Element may interact with the user. This interaction has two parts: a tracking phase, which normally lasts as long as the user holds the mouse button down, and an action which is performed either repeatedly during the tracking phase (for example, while the user manipulates a slider control) or once at the end of it (for example, when the user "presses a button").

If element is a pointer to a Graphic Element, its tracking procedure is element->trackingProc and element —>actionProc is its action procedure.

— **The custom cleanup procedure:** Many Graphic Elements need to allocate memory for their own usage. They can return this memory to the system when they are freed by providing a custom cleanup procedure. This procedure will automatically be called when an element (or the world containing it) is disposed.

Automating Graphic Elements

The use of autochange, collision, and interaction procedures allows complete separation of graphics programming from the main line of the application program's code. These procedures are called automatically by the display controller under the appropriate conditions. If a Graphic Element's changeIntrvl has expired, its autochange procedure is called when the application program calls DoWorldUpdate(). Its collision procedure is called whenever it comes into contact with another element on its collision plane because of a MoveElement() or MoveElementTo() call. The tracking procedure of a sensor-type element is called by the display controller whenever the user presses the mouse button while the cursor is within its animationRect. The use of these facilities provided by the Graphic Elements system allows the programmer to create complex graphic scenes which are portable, maintainable, and reusable.

Autochange Procedures

The prototype for an autochange procedure is

```
typedef pascal void (*AutoChangeProc) (GEWorldPtr world,
                                       GrafElPtr element);
```

Where:

— world is the GEWorld for which the autochange procedure is being called, and

— element is the Graphic Element for which it is being called.

Autochange procedures are executed in the context of the application program, and therefore have free access to global variables, other application functions, etc. However, it is best to limit their actions to those which directly affect the Graphic Elements for which they are called, since reliance upon such external facilities reduces their portability.

The display controller provides the following procedure which the application calls during the initialization of a Graphic Element to install its autochange capability:

```
void SetAutoChange(GEWorldPtr world, OSType elementID,
                        AutoChangeProc changeProc,
                        Ptr changeData, short changeIntrvl);
```

Where:

— world is the GEWorld containing the element,

— elementID is the unique four-character ID assigned to the element,

— changeProc is a pointer to the autochange procedure for the element,

— changeData is a pointer to any auxiliary data to be used by the autochange procedure, for example a pointer to a path record or a motion record for the element,

— changeIntrvl is the desired interval, in milliseconds, between calls to the autochange procedure for the element.

Collision Procedures

The Graphic Elements system includes powerful built-in facilities for detecting and reporting collisions between elements. At any time after it creates a new Graphic Element, the application program can specify that this element is to react to collisions with elements in any specified plane by calling:

```
void SetCollision(GEWorldPtr world, OSType elementID,
            CollisionProc collideProc, short collidePlane);
```

Where:

— world is the GEWorld containing the element,

— elementID is the unique four-character ID assigned to the element,

— collideProc is a pointer to the element's collision procedure, which can be nil,

— collidePlane is the number of the plane containing elements with which this element will collide.

When collision detection is active for a given Graphic Element, the display controller follows this sequence of steps for handling possible collisions each time it is moved:

- The display controller checks whether the rectangle representing the current location of the element intersects the rectangle of any other element on the element's collision plane.
- If an intersection is found, the display controller determines the “direction” of the collision by the manner in which the two rectangles overlap.
- If a collision is detected, the display controller determines the “phase” of the collision by comparing the element hit to the element with which this Graphic Element was previously in collision.
- If no collision is detected, the display controller determines whether this element was previously in collision with another.
- If a collision has occurred or this element was previously in collision, and the element has a collision procedure, that procedure is called.
- If the Graphic Element with which this element is (or was) colliding has a collision plane equal to this element's plane and has a collision procedure, that element's collision procedure is called.

The Graphic Elements system defines the following enumerated types to assist in dealing with collisions:

```
typedef enum {none, up, left, down, right,
            upLeft, upRight, downLeft, downRight} GEDirection;

typedef enum {collisionBegin, collisionContinue,
                        collisionEnd} CollisionPhase;
```

The collision procedure for an individual Graphic Element has the prototype:

```
typedef pascal void (*CollisionProc)(GEWorldPtr world,
                  GrafElPtr element, GEDirection dir,
                  CollisionPhase phase, GrafElPtr hitElement);
```

Where:

- world is the GEWorld for which the collision procedure is being called.
- element is the Graphic Element for which it is being called.
- dir is the direction of the collision, as calculated by the display controller. This direction specifies which part of this element has touched the “hit” element.
- phase is the current status of the collision. If this element has just entered into contact with hitElement, phase will be collisionBegin. For as long as this element remains in contact with hitElement, its collision procedure will be called repeatedly with phase equal to collisionContinue. Immediately after the two elements cease to touch one another, the collision procedure will be called with phase equal to collisionEnd.
- hitElement is a pointer to the element with which this element has collided.

Like Autochange procedures, collision procedures are executed in the context of the application. The choice of parameters passed to them represents a compromise between speed and flexibility. Collision detection and reporting is a complex subject. The amount of information needed to handle a collision varies widely, depending on the Graphic Element in question. For example, while the mere fact of collision might be enough information for a Graphic Element representing a bomb in an arcade game, another element representing a physical object in a simulation might need to calculate a new motion vector, taking into account its own motion and that of the object it has collided with. Some other Graphic Element might react only to the beginning and ending of the contact. The display controller's collision routines are designed to be fast and efficient, while still providing data which is useful as a starting point for more detailed calculations.

Interaction Procedures

In general, the tracking capabilities of interactive Graphic Elements are specific to a given type of sensor, and are set up during the creation of these elements. This is true for the standard sensor-type elements. The writing of tracking procedures will be discussed in the section “Customizing Graphic Elements.” A sensor's action procedure, on the other hand, is specific to an individual Graphic Element. This procedure has the prototype:

```
typedef pascal void (*SensorAction)(GEWorldPtr world,
                        GrafElPtr sensor, short sensorState);
```

Where:

- world is the GEWorld containing the sensor,
- **sensor** is the sensor which has been activated,
- sensorState is the state of the sensor (for example, the present setting of a slider-type sensor) when the action procedure is called.

The application program calls the following procedure during the initialization of a sensor-type Graphic Element to install its action procedure:

```
void SetSensorAction(GEWorldPtr world, OSType sensorID,
                                    SensorAction newAction);
```

Where:

- world is the GEWorld containing the sensor,
- sensorID is the unique four-character ID assigned to the sensor element,
- newAction is a pointer to the action procedure to be assigned to the sensor element.

Custom Cleanup Procedures

During initialization, a Graphic Element may need to allocate system memory to itself, for example to hold the data pointed to by
element—>changeData. Naturally, it should return this memory to the system when it is freed. It accomplishes this by providing a custom cleanup procedure. This procedure has the prototype:

```
typedef pascal void (*CleanupProc)(GEWorldPtr world,
```

```
                                                    GrafElPtr element);
```

Such a procedure should free any memory specific to an individual Graphic Element, for example that pointed to by the element's drawData or changeData fields.

The custom cleanup procedure for a Graphic Element can be set by calling

```
void SetCleanupProc(GEWorldPtr world, OSType elementID,
                                CleanupProc elemCleanup);
```

Where:

- world is the GEWorld containing the element,
- elementID is the unique four-character ID assigned to the element,
- elemCleanup is a pointer to the custom cleanup procedure.