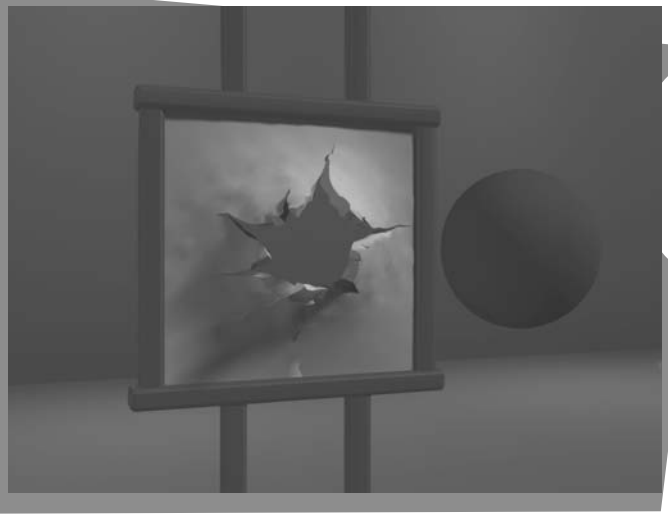
Interactive Input Methods and Graphical User Interfaces



Although we can construct programs and provide input data using the methods and program commands discussed in the previous chapters, it is often useful to be able to specify graphical input interactively. During the execution of a program, for example, we might want to change the position of the camera or the location of an object in a scene by pointing to a screen position, or we might want to change animation parameters using menu selections. In design applications, control-point coordinates for spline constructions are chosen interactively, and pictures are often constructed using interactive painting or drawing methods. There are several kinds of data that are used by a graphics program, and a variety of interactive input methods have been devised for processing these data values. In addition, interfaces for systems now involve extensive interactive graphics, including display windows, icons, menus, and a mouse or other cursor-control devices.

1 Graphical Input Data

Graphics programs use several kinds of input data, such as coordinate positions, attribute values, character-string specifications, geometric-transformation values, viewing conditions, and illumination parameters. Many graphics packages, including the International Standards Organization (ISO) and American National Standards Institute (ANSI) standards, provide an extensive set of input functions for processing such data. But input procedures require interaction with display-window managers and specific hardware devices. Therefore, some graphics systems, particularly those that provide mainly device-independent functions, often include relatively few interactive procedures for dealing with input data.

A standard organization for input procedures in a graphics package is to classify the functions according to the type of data that is to be processed by each function. This scheme allows any physical device, such as a keyboard or a mouse, to input any data class, although most input devices can handle some data types better than others.

2 Logical Classification of Input Devices

When input functions are classified according to data type, any device that is used to provide the specified data is referred to as a **logical input device** for that data type. The standard logical input-data classifications are

LOCATOR	A device for specifying one coordinate position.
STROKE	A device for specifying a set of coordinate positions.
STRING	A device for specifying text input.
VALUATOR	A device for specifying a scalar value.
CHOICE	A device for selecting a menu option.
PICK	A device for selecting a component of a picture.

Locator Devices

Interactive selection of a coordinate point is usually accomplished by positioning the screen cursor at some location in a displayed scene, although other methods, such as menu options, could be used in certain applications. We can use a mouse, touchpad, joystick, trackball, spaceball, thumbwheel, dial, hand cursor, or digitizer stylus for screen-cursor positioning. In addition, various buttons, keys, or switches can be used to indicate processing options for the selected location.

Keyboards are used for locator input in several ways. A general-purpose keyboard usually has four cursor-control keys that move the screen cursor up, down, left, and right. With an additional four keys, we can move the cursor diagonally as well. Rapid cursor movement is accomplished by holding down the selected cursor key. Sometimes a keyboard includes a touchpad, joystick, trackball, or other device for positioning the screen cursor. For some applications, it may also be convenient to use a keyboard to type in numerical values or other codes to indicate coordinate positions.

Other devices, such as a light pen, have also been used for interactive input of coordinate positions. But light pens record screen positions by detecting light from the screen phosphors, and this requires special implementation procedures.

Stroke Devices

This class of logical devices is used to input a sequence of coordinate positions, and the physical devices used for generating locator input are also used as stroke devices. Continuous movement of a mouse, trackball, joystick, or hand cursor is translated into a series of input coordinate values. The graphics tablet is one of the more common stroke devices. Button activation can be used to place the tablet into "continuous" mode. As the cursor is moved across the tablet surface, a stream of coordinate values is generated. This procedure is used in paintbrush systems to generate drawings using various brush strokes. Engineering systems also use this process to trace and digitize layouts.

String Devices

The primary physical device used for string input is the keyboard. Character strings in computer-graphics applications are typically used for picture or graph labeling.

Other physical devices can be used for generating character patterns for special applications. Individual characters can be sketched on the screen using a stroke or locator-type device. A pattern recognition program then interprets the characters using a stored dictionary of predefined patterns.

Valuator Devices

We can employ valuator input in a graphics program to set scalar values for geometric transformations, viewing parameters, and illumination parameters. In some applications, scalar input is also used for setting physical parameters such as temperature, voltage, or stress-strain factors.

A typical physical device used to provide valuator input is a panel of control dials. Dial settings are calibrated to produce numerical values within some predefined range. Rotary potentiometers convert dial rotation into a corresponding voltage, which is then translated into a number within a defined scalar range, such as -10.5 to 25.5. Instead of dials, slide potentiometers are sometimes used to convert linear movements into scalar values.

Any keyboard with a set of numeric keys can be used as a valuator device. Although dials and slide potentiometers are more efficient for fast input.

Joysticks, trackballs, tablets, and other interactive devices can be adapted for valuator input by interpreting pressure or movement of the device relative to a scalar range. For one direction of movement, say left to right, increasing scalar values can be input. Movement in the opposite direction decreases the scalar input value. Selected values are usually echoed on the screen for verification.

Another technique for providing valuator input is to display graphical representations of sliders, buttons, rotating scales, and menus on the video monitor. Cursor positioning, using a mouse, joystick, spaceball, or other device, can be used to select a value on one of these valuators. As a feedback mechanism for the user, selected values are usually displayed in text or color fields elsewhere within the graphical display belonging to the application.

Choice Devices

Menus are typically used in graphics programs to select processing options, parameter values, and object shapes that are to be used in constructing a picture. Commonly used choice devices for selecting a menu option are cursor-positioning devices such as a mouse, trackball, keyboard, touch panel, or button box.

Keyboard function keys or separate button boxes are often used to enter menu selections. Each button or function key is programmed to select a particular

operation or value, although preset buttons or keys are sometimes included on an input device.

For screen selection of listed menu options, we use a cursor-positioning device. When a screen-cursor position (x, y) is selected, it is compared to the coordinate extents of each listed menu item. A menu item with vertical and horizontal boundaries at the coordinate values x_{min}, x_{max}, y_{min}, and y_{max} is selected if the input coordinates satisfy the inequalities

$$x_{min} \leq x \leq x_{max}, \qquad y_{min} \leq y \leq y_{max} \tag{1}$$

For larger menus with relatively few options displayed, a touch panel is commonly used. A selected screen position is compared to the coordinate extents of the individual menu options to determine what process is to be performed.

Alternate methods for choice input include keyboard and voice entry. A standard keyboard can be used to type in commands or menu options. For this method of choice input, some abbreviated format is useful. Menu listings can be numbered or given short identifying names. A similar encoding scheme can be used with voice input systems. Voice input is particularly useful when the number of options is small (20 or fewer).

Pick Devices

We use a pick device to select a part of a scene that is to be transformed or edited in some way. Several different methods can be used to select a component of a displayed scene, and any input mechanism used for this purpose is classified as a pick device. Most often, pick operations are performed by positioning the screen cursor. Using a mouse, joystick, or keyboard, for example, we can perform picking by positioning the screen cursor and pressing a button or key to record the pixel coordinates. This screen position can then be used to select an entire object, a facet of a tessellated surface, a polygon edge, or a vertex. Other pick methods include highlighting schemes, selecting objects by name, or a combination of methods.

Using the cursor-positioning approach, a pick procedure could map a selected screen position to a world-coordinate location using the inverse viewing and geometric transformations that were specified for the scene. Then, the world-coordinate position can be compared to the coordinate extents of objects. If the pick position is within the coordinate extents of a single object, the pick object has been identified. The object name, coordinates, or other information about the object can then be used to apply the desired transformation or editing operations. But if the pick position is within the coordinate extents of two or more objects, further testing is necessary. Depending on the type of object to be selected and the complexity of a scene, several levels of search may be required to identify the pick object. For example, if we are attempting to pick a sphere whose coordinate extents overlap the coordinate extents of some other three-dimensional object, the pick position could be compared to the coordinate extents of the individual surface facets of the two objects. If this test fails, the coordinate extents of individual line segments can be tested.

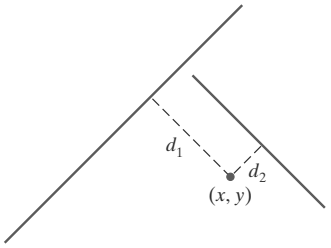


FIGURE 1
Distances to line segments from a pick position.

When coordinate-extent tests do not uniquely identify a pick object, the distances from the pick position to individual line segments could be computed. Figure 1 illustrates a pick position that is within the coordinate extents of two line segments. For a two-dimensional line segment with pixel endpoint coordinates (x_1, y_1) and (x_2, y_2), the perpendicular distance squared from a pick position (x, y) to the line is calculated as

$$d^2 = \frac{[\Delta x(y - y_1) - \Delta y(x - x_1)]^2}{\Delta x^2 + \Delta y^2} \tag{2}$$

where $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$. Other methods, such as comparing distances to endpoint positions, have been proposed to simplify the line-picking operations.

Pick procedures can be simplified if coordinate-extent testing is not carried out for the surface facets and line segments of an object. When the pick position is within the coordinate extents of two or more objects, the pick procedures can simply return a list of all candidate pick objects.

Another picking technique is to associate a **pick window** with a selected cursor position. The pick window is centered on the cursor position, as shown in Figure 2, and clipping procedures are used to determine which objects intersect the pick window. For line picking, we can set the pick-window dimensions w and h to very small values, so that only one line segment intersects the pick window. Some graphics packages implement three-dimensional picking by reconstructing a scene using the viewing and projection transformations with the pick window as the clipping window. Nothing is displayed from this reconstruction, but clipping procedures are applied to determine which objects are within the pick view volume. A list of information for each object in the pick view volume can then be returned for processing. This list can contain information such as object name and depth range, where the depth range could be used to select the nearest object in the pick view volume.

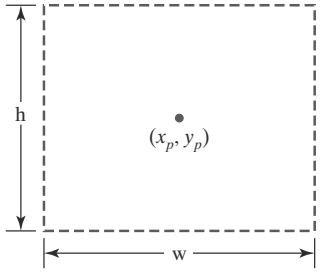


FIGURE 2
A pick window with center coordinates (x_p, y_p), width w, and height h.

Highlighting can also be used to facilitate picking. One way to do this is to successively highlight those objects whose coordinate extents overlap a pick position (or pick window). As each object is highlighted, a user could issue a "reject" or "accept" action using keyboard keys. The sequence stops when the user accepts a highlighted object as the pick object. Picking could also be accomplished simply by successively highlighting all objects in the scene without selecting a cursor position. The highlighting sequence can be initiated with a button or function key, and a second button can be used to stop the process when the desired object is highlighted. If very many objects are to be searched in this way, additional buttons can be used to speed up the highlighting process. One button initiates a rapid successive highlighting of structures. A second button is activated to stop the process, and a third button is used to back up slowly through the highlighting process. Finally, a stop button could be pressed to complete the pick procedure.

If picture components can be selected by name, keyboard input can be used to pick an object. This is a straightforward, but less interactive, pick-selection method. Some graphics packages allow picture components to be named at various levels down to the individual primitives. Descriptive names can be used to help a user in the pick process, but this approach has drawbacks. It is generally slower than interactive picking on the screen, and a user will probably need prompts to remember the various structure names.

3 Input Functions for Graphical Data

Graphics packages that use the logical classification for input devices provide several functions for selecting devices and data classes. These functions allow a user to specify the following options:

- The input interaction mode for the graphics program and the input devices. Either the program or the devices can initiate data entry, or both can operate simultaneously.
- Selection of a physical device that is to provide input within a particular logical classification (for example, a tablet used as a stroke device).
- Selection of the input time and device for a particular set of data values.

Input Modes

Some input functions in an interactive graphics system are used to specify how the program and input devices should interact. A program could request input at a particular time in the processing (request mode), or an input device could independently provide updated input (sample mode), or the device could independently store all collected data (event mode).

In **request mode,** the application program initiates data entry. When input values are requested, processing is suspended until the required values are received. This input mode corresponds to the typical input operation in a general programming language. The program and the input devices operate alternately. Devices are put into a wait state until an input request is made; then the program waits until the data are delivered.

In **sample mode,** the application program and input devices operate independently. Input devices may be operating at the same time that the program is processing other data. New values obtained from the input devices replace previously input data values. When the program requires new data, it samples the current values that have been stored from the device input.

In **event mode,** the input devices initiate data input to the application program. The program and the input devices again operate concurrently, but now the input devices deliver data to an input queue, also called an *event queue*. All input data is saved. When the program requires new data, it goes to the data queue.

Typically, any number of devices can be operating at the same time in sample and event modes. Some can be operating in sample mode, while others are operating in event mode. But only one device at a time can deliver input in request mode.

Other functions in the input library are used to specify physical devices for the logical data classes. The input procedures in an interactive package can involve complicated processing for some kinds of input. For instance, to obtain a world-coordinate position, the input procedures must process an input screen location back through the viewing and other transformations to the original world-coordinate description of a scene. This processing also involves information from the display-window routines.

Echo Feedback

Requests can usually be made in an interactive input program for an echo of input data and associated parameters. When an echo of the input data is requested, it is displayed within a specified screen area. Echo feedback can include, for example, the size of the pick window, the minimum pick distance, the type and size of a cursor, the type of highlighting to be employed during pick operations, the range (mininum and maximum) for valuator input, and the resolution (scale) for valuator input.

Callback Functions

For device-independent graphics packages, a limited set of input functions can be provided in an auxiliary library. Input procedures can then be handled as callback functions that interact with the system software. These functions specify what actions are to be taken by a program when an input event occurs. Typical input events are moving a mouse, pressing a mouse button, or pressing a key on the keyboard.

4 Interactive Picture-Construction Techniques

A variety of interactive methods are often incorporated into a graphics package as aids in the construction of pictures. Routines can be provided for positioning objects, applying constraints, adjusting the sizes of objects, and designing shapes and patterns.

Basic Positioning Methods

We can interactively choose a coordinate position with a pointing device that records a screen location. How the position is used depends on the selected processing option. The coordinate location could be an endpoint position for a new line segment, or it could be used to position some object—for instance, the selected screen location could reference a new position for the center of a sphere; or the location could be used to specify the position for a text string, which could begin at that location or it could be centered on that location. As an additional positioning aid, numeric values for selected positions can be echoed on the screen. With the echoed coordinate values as a guide, a user could make small interactive adjustments in the coordinate values using dials, arrow keys, or other devices.

Dragging

Another interactive positioning technique is to select an object and drag it to a new location. Using a mouse, for instance, we position the cursor at the object position, press a mouse button, move the cursor to a new position, and release the button. The object is then displayed at the new cursor location. Usually, the object is displayed at intermediate positions as the screen cursor moves.

Constraints

Any procedure for altering input coordinate values to obtain a particular orientation or alignment of an object is called a constraint. For example, an input line segment can be constrained to be horizontal or vertical, as illustrated in Figures 3 and 4. To implement this type of constraint, we compare the input coordinate values at the two endpoints. If the difference in the y values of the two endpoints is smaller than the difference in the x values, a horizontal line is displayed. Otherwise, a vertical line is drawn. The horizontal-vertical constraint is useful, for instance, in forming network layouts, and it eliminates the need for precise positioning of endpoint coordinates.

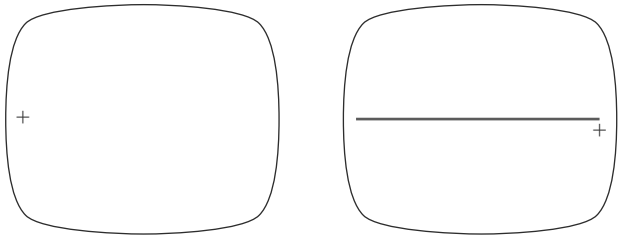


FIGURE 3
Horizontal line constraint.

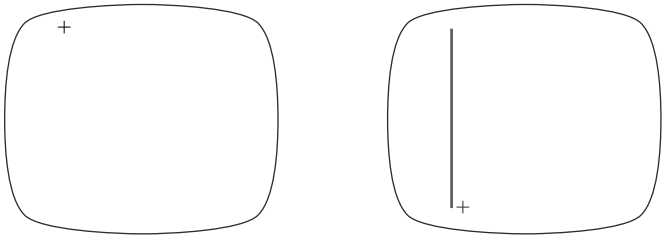


FIGURE 4
Vertical line constraint.

Other kinds of constraints can be applied to input coordinates to produce a variety of alignments. Lines could be constrained to have a particular slant, such as 45°, and input coordinates could be constrained to lie along predefined paths, such as circular arcs.

Grids

Another kind of constraint is a rectangular grid displayed in some part of the screen area. With an activated grid constraint, input coordinates are rounded to the nearest grid intersection. Figure 5 illustrates line drawing using a grid. Each of the cursor positions in this example is shifted to the nearest grid intersection point, and a line is drawn between these two grid positions. Grids facilitate object constructions, because a new line can be joined easily to a previously drawn line by selecting any position near the endpoint grid intersection of one end of the displayed line. Spacing between grid lines is often an option, and partial grids or grids with different spacing could be used in different screen areas.

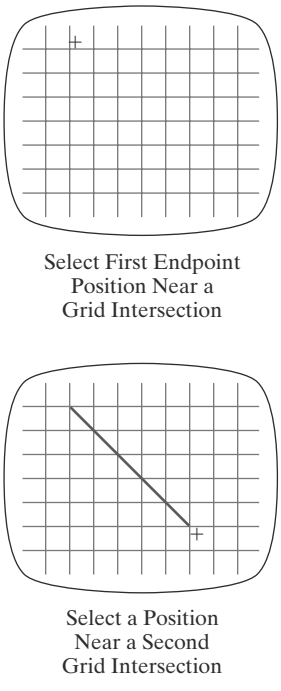


FIGURE 5
Construction of a line segment with endpoints constrained to grid intersection positions.

Rubber-Band Methods

Line segments and other basic shapes can be constructed and positioned using rubber-band methods that allow the sizes of objects to be interactively stretched or contracted. Figure 6 demonstrates a rubber-band method for interactively specifying a line segment. First, a fixed screen position is selected for one endpoint of the line. Then, as the cursor moves around, the line is displayed from the start position to the current position of the cursor. The second endpoint of the line is input when a button or key is pressed. Using a mouse, we construct a rubber-band line while pressing a mouse key. When the mouse key is released, the line display is completed.

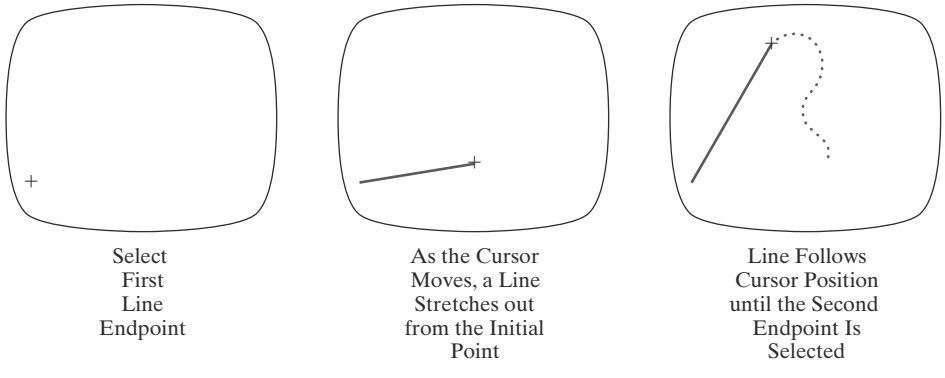


FIGURE 6
A rubber-band method for constructing and positioning a straight-line segment.

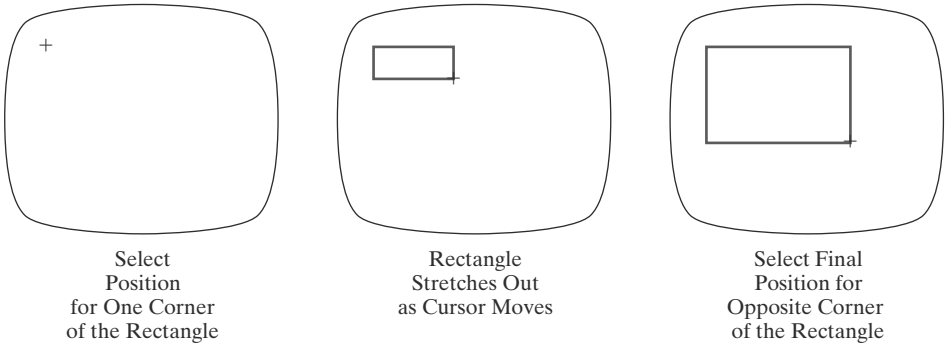


FIGURE 7
A rubber-band method for constructing a rectangle.

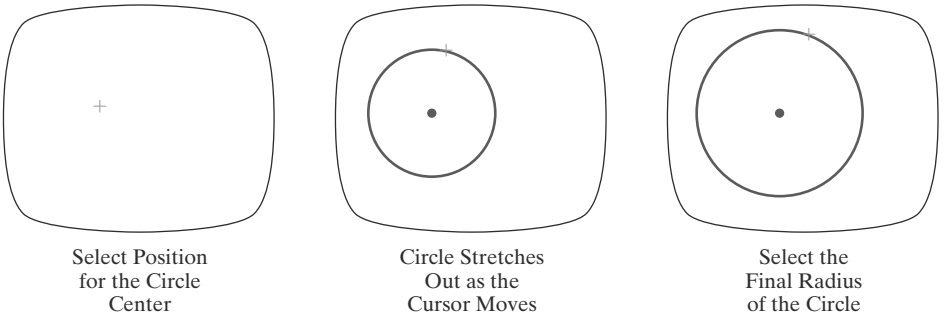


FIGURE 8
Constructing a circle using a rubber-band method.

We can use similar rubber-band methods to construct rectangles, circles, and other objects. Figure 7 demonstrates rubber-band construction of a rectangle, and Figure 8 shows a rubber-band circle construction. We can implement rubber-band constructions in various ways. For example, the shape and size of a rectangle can be adjusted by independently moving only the top edge of the rectangle, or the bottom edge, or one of the side edges.

Gravity Field

In the construction of figures, we sometimes need to connect lines at positions between endpoints that are not at grid intersections. Because exact positioning of the screen cursor at the connecting point can be difficult, a graphics package can include a procedure that converts any input position near a line segment into a position on the line using a *gravity field* area around the line. Any selected position within the gravity field of a line is moved ("gravitated") to the nearest position on the line. A gravity field area around a line is illustrated with the shaded region shown in Figure 9.

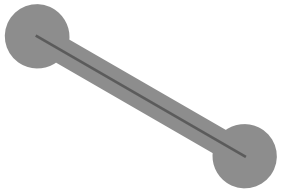


FIGURE 9
A gravity field around a line. Any selected point in the shaded area is shifted to a position on the line.

Gravity fields around the line endpoints are enlarged to make it easier for a designer to connect lines at their endpoints. Selected positions in one of the circular areas of the gravity field are attracted to the endpoint in that area. The size of gravity fields is chosen large enough to aid positioning, but small enough to reduce chances of overlap with other lines. If many lines are displayed, gravity areas can overlap, and it may be difficult to specify points correctly. Normally, the boundary for the gravity field is not displayed.

Interactive Painting and Drawing Methods

Options for sketching, drawing, and painting come in a variety of forms. Straight lines, polygons, and circles can be generated with methods discussed in the

previous sections. Curve-drawing options can be provided using standard curve shapes, such as circular arcs and splines, or with freehand sketching procedures. Splines are interactively constructed by specifying a set of control points or a freehand sketch that gives the general shape of the curve. Then the system fits the set of points with a polynomial curve. In freehand drawing, curves are generated by following the path of a stylus on a graphics tablet or the path of the screen cursor on a video monitor. Once a curve is displayed, the designer can alter the curve shape by adjusting the positions of selected points along the curve path.

Line widths, line styles, and other attribute options are also commonly found in painting and drawing packages. Various brush styles, brush patterns, color combinations, object shapes, and surface texture patterns are also available on many systems, particularly those designed as artists' workstations. Some paint systems vary the line width and brush strokes according to the pressure of the artist's hand on the stylus. Color Plate 23 shows a window and menu system used with a painting package that allows an artist to select variations of a specified object shape, different surface textures, and a variety of lighting conditions for a scene.

5 Virtual-Reality Environments

A typical virtual-reality environment is illustrated in Color Plate 24. Interactive input is accomplished in this environment with a data glove, which is capable of grasping and moving objects displayed in a virtual scene. The computer-generated scene is displayed through a head-mounted viewing system as a stereographic projection. Tracking devices compute the position and orientation of the headset and data glove relative to the object positions in the scene. With this system, a user can move through the scene and rearrange object positions with the data glove.

Another method for generating virtual scenes is to display stereographic projections on a raster monitor, with the two stereographic views displayed on alternate refresh cycles. The scene is then viewed through stereographic glasses. Interactive object manipulations can again be accomplished with a data glove and a tracking device to monitor the glove position and orientation relative to the position of objects in the scene.

6 OpenGL Interactive Input-Device Functions

Interactive device input in an OpenGL program is handled with routines in the OpenGL Utility Toolkit (GLUT), because these routines need to interface with a window system. In GLUT, we have functions to accept input from standard devices, such as a mouse or a keyboard, as well as from tablets, space balls, button boxes, and dials. For each device, we specify a procedure (the call back function) that is to be invoked when an input event from that device occurs. These GLUT commands are placed in the `main` procedure along with the other GLUT statements. In addition, a combination of functions from the basic library and the GLU library can be used with the GLUT mouse function for pick input.

GLUT Mouse Functions

We use the following function to specify ("register") a procedure that is to be called when the mouse pointer is in a display window and a mouse button is pressed or released:

```
glutMouseFunc (mouseFcn);
```

This mouse callback procedure, which we named `mouseFcn`, has four arguments:

```
void mouseFcn (GLint button, GLint action, GLint xMouse,
                                    GLint yMouse)
```

Parameter `button` is assigned a GLUT symbolic constant that denotes one of the three mouse buttons, and parameter `action` is assigned a symbolic constant that specifies which button action we want to use to trigger the mouse activation event. Allowable values for `button` are `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON`, and `GLUT_RIGHT_BUTTON`. (If we have only a two-button mouse, then we use just the left-button and right-button designations; with a one-button mouse, we can assign parameter `button` only the value `GLUT_LEFT_BUTTON`.) Parameter action can be assigned either `GLUT_DOWN` or `GLUT_UP`, depending on whether we want to initiate an action when we press a mouse button or when we release it. When procedure `mouseFcn` is invoked, the display-window location of the mouse cursor is returned as the coordinate position (`xMouse`, `yMouse`). This location is relative to the top-left corner of the display window, so that `xMouse` is the pixel distance from the left edge of the display window and `yMouse` is the pixel distance down from the top of the display window.

By activating a mouse button while the screen cursor is within the display window, we can select a position for displaying a primitive such as a single point, a line segment, or a fill area. We could also use the mouse as a pick device by comparing the returned screen position with the coordinate extents of displayed objects in a scene. However, OpenGL does provide routines for using the mouse as a pick device, and we discuss these routines in a later section.

As a simple example of the use of the `glutMouseFunc` routine, the following program plots a red point, with a point size equal to 3, at the position of the mouse cursor in the display window, each time that we press the left mouse button. Because the coordinate origin for the OpenGL primitive functions is the lower-left corner of the display window, we need to flip the returned `yMouse` value in the procedure `mousePtPlot`.

```
#include <GL/glut.h>

GLsizei winWidth = 400, winHeight = 300;   // Initial display-window size.

void init (void)
{
   glClearColor (0.0, 0.0, 1.0, 1.0)  // Set display-window color to blue.

   glMatrixMode (GL_PROJECTION);
   gluOrtho2D (0.0, 200.0, 0.0, 150.0);
}
```

```
void displayFcn (void)
{
   glClear (GL_COLOR_BUFFER_BIT);     //  Clear display window.

   glColor3f (1.0, 0.0, 0.0);         //  Set point color to red.
   glPointSize (3.0);                 //  Set point size to 3.0.
}

void winReshapeFcn (GLint newWidth, GLint newHeight)
{
   /*  Reset viewport and projection parameters  */
   glViewport (0, 0, newWidth, newHeight);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ( );
   gluOrtho2D (0.0, GLdouble (newWidth), 0.0, GLdouble (newHeight));

   /*  Reset display-window size parameters.  */
   winWidth = newWidth;
   winHeight = newHeight;
}

void plotPoint (GLint x, GLint y)
{
   glBegin (GL_POINTS);
      glVertex2i (x, y);
   glEnd ( );
}

void mousePtPlot (GLint button, GLint action, GLint xMouse, GLint yMouse)
{
   if (button == GLUT_LEFT_BUTTON && action == GLUT_DOWN)
      plotPoint (xMouse, winHeight - yMouse);

   glFlush ( );
 }

 void main (int argc, char** argv)
 {
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition (100, 100);
    glutInitWindowSize (winWidth, winHeight);
    glutCreateWindow ("Mouse Plot Points");

    init ( );
    glutDisplayFunc (displayFcn);
    glutReshapeFunc (winReshapeFcn);
    glutMouseFunc (mousePtPlot);

    glutMainLoop ( );
}
```

The next program example uses mouse input to select an endpoint position for a straight-line segment. Selected line segments are connected to demonstrate interactive construction of a polyline. Initially, two display-window locations must be selected with the left mouse button to generate the first line segment. Each subsequent position that we select adds another segment to the polyline.

```
#include <GL/glut.h>

GLsizei winWidth = 400, winHeight = 300;   // Initial display-window size.
GLint endPtCtr = 0;                    // Initialize line endpoint counter.

class scrPt {
public:
   GLint x, y;
};

void init (void)
{
   glClearColor (0.0, 0.0, 1.0, 1.0)   // Set display-window color to blue.

   glMatrixMode (GL_PROJECTION);
   gluOrtho2D (0.0, 200.0, 0.0, 150.0);
}

void displayFcn (void)
{
   glClear (GL_COLOR_BUFFER_BIT);
}

void winReshapeFcn (GLint newWidth, GLint newHeight)
{
   /*  Reset viewport and projection parameters  */
   glViewport (0, 0, newWidth, newHeight);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ( );
   gluOrtho2D (0.0, GLdouble (newWidth), 0.0, GLdouble (newHeight));

   /*  Reset display-window size parameters.  */
   winWidth  = newWidth;
   winHeight = newHeight;
}

void drawLineSegment (scrPt endPt1, scrPt endPt2)
{
   glBegin (GL_LINES);
      glVertex2i (endPt1.x, endPt1.y);
      glVertex2i (endPt2.x, endPt2.y);
   glEnd ( );
}
```

```
void polyline (GLint button, GLint action, GLint xMouse, GLint yMouse)
{
   static scrPt endPt1, endPt2;

   if (ptCtr == 0) {
      if (button == GLUT_LEFT_BUTTON && action == GLUT_DOWN) {
         endPt1.x = xMouse;
         endPt1.y = winHeight - yMouse;
         ptCtr = 1;
      }
      else
         if (button == GLUT_RIGHT_BUTTON)        // Quit the program.
            exit (0);
   }
   else
      if (button == GLUT_LEFT_BUTTON && action == GLUT_DOWN) {
         endPt2.x = xMouse;
         endPt2.y = winHeight - yMouse;
         drawLineSegment (endPt1, endPt2);

         endPt1 = endPt2;
      }
      else
         if (button == GLUT_RIGHT_BUTTON)        // Quit the program.
            exit (0);

   glFlush ( );
}

void main (int argc, char** argv)
{
   glutInit (&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
   glutInitWindowPosition (100, 100);
   glutInitWindowSize (winWidth, winHeight);
   glutCreateWindow ("Draw Interactive Polyline");

   init ( );
   glutDisplayFunc (displayFcn);
   glutReshapeFunc (winReshapeFcn);
   glutMouseFunc (polyline);

   glutMainLoop ( );
}
```

Another GLUT mouse routine that we can use is

```
glutMotionFunc (fcnDoSomething);
```

This routine invokes `fcnDoSomething` when the mouse is moved within the display window with one or more buttons activated. The function that is invoked in this case has two arguments:

```
void fcnDoSomething (GLint xMouse, GLint yMouse)
```

where (xMouse, yMouse) is the mouse location in the display window relative to the top-left corner, when the mouse is moved with a button pressed.

Similarly, we can perform some action when we move the mouse within the display window without pressing a button:

```
glutPassiveMotionFunc (fcnDoSomethingElse);
```

Again, the mouse location is returned to fcnDoSomethingElse as coordinate position (xMouse, yMouse), relative to the top-left corner of the display window.

GLUT Keyboard Functions

With keyboard input, we use the following function to specify a procedure that is to be invoked when a key is pressed:

```
glutKeyboardFunc (keyFcn);
```

The specified procedure has three arguments:

```
void keyFcn (GLubyte key, GLint xMouse, GLint yMouse)
```

Parameter key is assigned a character value or the corresponding ASCII code. The display-window mouse location is returned as position (xMouse, yMouse) relative to the top-left corner of the display window. When a designated key is pressed, we can use the mouse location to initiate some action, independently of whether any mouse buttons are pressed.

In the following code, we present a simple curve-drawing procedure using keyboard input. A freehand curve is generated by moving the mouse within the display window while holding down the "c" key. This displays a sequence of red dots at each recorded mouse position. By slowly moving the mouse, we can obtain a solid curved line. Mouse buttons have no effect in this example.

```
#include <GL/glut.h>

GLsizei winWidth = 400, winHeight = 300;   // Initial display-window size.

void init (void)
{
   glClearColor (0.0, 0.0, 1.0, 1.0);   // Set display-window color to blue.

   glMatrixMode (GL_PROJECTION);
   gluOrtho2D (0.0, 200.0, 0.0, 150.0);
}

void displayFcn (void)
{
   glClear (GL_COLOR_BUFFER_BIT);          //  Clear display window.

   glColor3f (1.0, 0.0, 0.0);              //  Set point color to red.
   glPointSize (3.0);                      //  Set point size to 3.0.
}
```

```
void winReshapeFcn (GLint newWidth, GLint newHeight)
{
   /*  Reset viewport and projection parameters  */
   glViewport (0, 0, newWidth, newHeight);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ( );
   gluOrtho2D (0.0, GLdouble (newWidth), 0.0, GLdouble (newHeight));

   /*  Reset display-window size parameters.  */
   winWidth  = newWidth;
   winHeight = newHeight;
}

void plotPoint (GLint x, GLint y)
{
   glBegin (GL_POINTS);
      glVertex2i (x, y);
   glEnd ( );
}

/*  Move cursor while pressing c key enables freehand curve drawing.  */
void curveDrawing (GLubyte curvePlotKey, GLint xMouse, GLint yMouse)
{
   GLint x = xMouse;
   GLint y = winHeight - yMouse;
   switch (curvePlotKey)
   {
      case 'c':
         plotPoint (x, y);
         break;
      default:
         break;
   }
   glFlush ( );
}

void main (int argc, char** argv)
{
   glutInit (&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
   glutInitWindowPosition (100, 100);
   glutInitWindowSize (winWidth, winHeight);
   glutCreateWindow ("Keyboard Curve-Drawing Example");

   init ( );
   glutDisplayFunc (displayFcn);
   glutReshapeFunc (winReshapeFcn);
   glutKeyboardFunc (curveDrawing);

   glutMainLoop ( );
}
```

For function keys, arrow keys, and other special-purpose keys, we can use the command

```
glutSpecialFunc (specialKeyFcn);
```

The specified procedure has the same three arguments:

```
void specialKeyFcn (GLint specialKey, GLint xMouse,
                                    GLint yMouse)
```

but now parameter `specialKey` is assigned an integer-valued GLUT symbolic constant. To select a function key, we use one of the constants `GLUT_KEY_F1` through `GLUT_KEY_F12`. For the arrow keys, we use constants such as `GLUT_KEY_UP` and `GLUT_KEY_RIGHT`. Other keys can be designated using `GLUT_KEY_PAGE_DOWN`, `GLUT_KEY_HOME`, and similar constants for the page up, end, and insert keys. The backspace, delete, and escape keys can be designated with the `glutKeyboardFunc` routine using their ASCII codes, which are 8, 127, and 27, respectively.

An interactive program using the mouse, keyboard, and function keys is demonstrated in the following code. Mouse input is used to select a location for the lower-left corner of a red square. Keyboard input is used to scale the size of the square, and a new square is obtained with each click of the left mouse button.

```
#include <GL/glut.h>
#inclue <stdlib.h>

GLsizei winWidth = 400, winHeight = 300;   // Initial display-window size.
GLint edgeLength = 10;                  // Initial edge length for square.

void init (void)
{
   glClearColor (0.0, 0.0, 1.0, 1.0)   // Set display-window color to blue.

   glMatrixMode (GL_PROJECTION);
   gluOrtho2D (0.0, 200.0, 0.0, 150.0);
}

void displayFcn (void)
{
   glClear (GL_COLOR_BUFFER_BIT);          //  Clear display window.

   glColor3f (1.0, 0.0, 0.0);              //  Set fill color to red.
}

void winReshapeFcn (GLint newWidth, GLint newHeight)
{
   /*  Reset viewport and projection parameters  */
   glViewport (0, 0, newWidth, newHeight);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ( );
   gluOrtho2D (0.0, GLdouble (newWidth), 0.0, GLdouble (newHeight));
```

```
   /*  Reset display-window size parameters.  */
   winWidth  = newWidth;
   winHeight = newHeight;
}

/*  Display a red square with a selected edge-length size.  */
void fillSquare (GLint button, GLint action, GLint xMouse, GLint yMouse)
{
   GLint x1, y1, x2, y2;

   /*  Use left mouse button to select a position for the
    *  lower-left corner of the square.
    */
   if (button == GLUT_LEFT_BUTTON && action == GLUT_DOWN)
   {
      x1 = xMouse;
      y1 = winHeight - yMouse;
      x2 = x1 + edgeLength;
      y2 = y1 + edgeLength;
      glRecti (x1, y1, x2, y2);
   }
   else
      if (button == GLUT_RIGHT_BUTTON)   //  Use right mouse button to quit.
         exit (0);

   glFlush ( );
}

/*  Use keys 2, 3, and 4 to enlarge the square.  */
void enlargeSquare (GLubyte sizeFactor, GLint xMouse, GLint yMouse)
{
   switch (sizeFactor)
   {
      case '2':
         edgeLength *= 2;
         break;
      case '3':
         edgeLength *= 3;
         break;
      case '4':
         edgeLength *= 4;
         break;
      default:
         break;
   }
}

/*  Use function keys F2 and F4 for reduction factors 1/2 and 1/4.  */
void reduceSquare (GLint reductionKey, GLint xMouse, GLint yMouse)
{
   switch (reductionKey)
   {
      case GLUT_KEY_F2:
         edgeLength /= 2;
         break;
```

```
      case GLUT_KEY_F3:
         edgeLength /= 4;
         break;
      default:
         break;
   }
}

void main (int argc, char** argv)
{
   glutInit (&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
   glutInitWindowPosition (100, 100);
   glutInitWindowSize (winWidth, winHeight);
   glutCreateWindow ("Display Squares of Various Sizes");

   init ( );
   glutDisplayFunc (displayFcn);
   glutReshapeFunc (winReshapeFcn);
   glutMouseFunc (fillSquare);
   glutKeyboardFunc (enlargeSquare);
   glutSpecialFunc (reduceSquare);

   glutMainLoop ( );
}
```

GLUT Tablet Functions

Usually, tablet activation occurs only when the mouse cursor is in the display window. A button event for tablet input is then recorded with

```
glutTabletButtonFunc (tabletFcn);
```

and the arguments for the invoked function are similar to those for a mouse:

```
void tabletFcn (GLint tabletButton, GLint action,
                     GLint xTablet, GLint yTablet)
```

We designate a tablet button with an integer identifier such as 1, 2, 3, and so on, and the button action is again specified with either `GLUT_UP` or `GLUT_DOWN`. The returned values `xTablet` and `yTablet` are the tablet coordinates. We can determine the number of available tablet buttons with the command

```
glutDeviceGet (GLUT_NUM_TABLET_BUTTONS);
```

Motion of a tablet stylus or cursor is processed with the following function:

```
glutTabletMotionFunc (tabletMotionFcn);
```

where the invoked function has the form

```
void tabletMotionFcn (GLint xTablet, GLint yTablet)
```

The returned values `xTablet` and `yTablet` give the coordinates on the tablet surface.

GLUT Spaceball Functions

We use the following function to specify an operation when a spaceball button is activated for a selected display window:

```
glutSpaceballButtonFunc (spaceballFcn);
```

The callback function has two parameters:

```
void spaceballFcn (GLint spaceballButton, GLint action)
```

Spaceball buttons are identified with the same integer values as a tablet, and parameter `action` is assigned either the value `GLUT_UP` or the value `GLUT_DOWN`. We can determine the number of available spaceball buttons with a call to `glutDeviceGet` using the argument `GLUT_NUM_SPACEBALL_BUTTONS`.

Translational motion of a spaceball, when the mouse is in the display window, is recorded with the function call

```
glutSpaceballMotionFunc (spaceballTranlFcn);
```

The three-dimensional translation distances are passed to the invoked function as, for example:

```
void spaceballTranslFcn (GLint tx, GLint ty, GLint tz)
```

These translation distances are normalized within the range from −1000 to 1000.

Similarly, a spaceball rotation is recorded with

```
glutSpaceballRotateFunc (spaceballRotFcn);
```

The three-dimensional rotation angles are then available to the callback function, as follows:

```
void spaceballRotFcn (GLint thetaX, GLint thetaY, GLint thetaZ)
```

GLUT Button-Box Function

Input from a button box is obtained with the following statement:

```
glutButtonBoxFunc (buttonBoxFcn);
```

Button activation is then passed to the invoked function:

```
void buttonBoxFcn (GLint button, GLint action);
```

The buttons are identified with integer values, and the button action is specified as `GLUT_UP` or `GLUT_DOWN`.

GLUT Dials Function

A dial rotation can be recorded with the following routine:

```
glutDialsFunc (dialsFcn);
```

In this case, we use the callback function to identify the dial and obtain the angular amount of rotation:

```
void dialsFcn (GLint dial, GLint degreeValue);
```

Dials are designated with integer values, and the dial rotation is returned as an integer degree value.

OpenGL Picking Operations

In an OpenGL program, we can interactively select objects by pointing to screen positions. However, the picking operations in OpenGL are not straightforward.

Basically, we perform picking using a designated pick window to form a revised view volume. We assign integer identifiers to objects in a scene, and the identifiers for those objects that intersect the revised view volume are stored in a pick-buffer array. Thus, to use the OpenGL pick features, we need to incorporate the following procedures into a program:

- Create and display a scene.
- Pick a screen position and, within the mouse callback function, do the following:
  - Set up a pick buffer.
  - Activate the picking operations (selection mode).
  - Initialize an ID name stack for object identifiers.
  - Save the current viewing and geometric-transformation matrix.
  - Specify a pick window for the mouse input.
  - Assign identifiers to objects and reprocess the scene using the revised view volume. (Pick information is then stored in the pick buffer.)
  - Restore the original viewing and geometric-transformation matrix.
  - Determine the number of objects that have been picked, and return to the normal rendering mode.
  - Process the pick information.

We can also use a modification of these procedures to select objects without interactive input from a mouse. This is accomplished by specifying the vertices for the revised view volume, instead of designating a pick window.

A pick-buffer array is set up with the command

```
glSelectBuffer (pickBuffSize, pickBuffer);
```

Parameter `pickBuffer` designates an integer array with `pickBuffSize` elements. The `glSelectBuffer` function must be invoked before the OpenGL picking operations (selection mode) are activated. An integer information record is stored in pick-buffer array for each object that is selected with a single pick input. Several records of information can be stored in the pick buffer, depending on the size and location of the pick window. Each record in the pick buffer contains the following information:

1. The stack position of the object, which is the number of identifiers in the name stack, up to and including the position of the picked object.
2. The minimum depth of the picked object.
3. The maximum depth of the picked object.
4. The list of the identifiers in the name stack from the first (bottom) identifier to the identifier for the picked object.

The integer depth values stored in the pick buffer are the original values in the range from 0 to 1.0, multiplied by $2^{32} - 1$.

The OpenGL picking operations are activated with

```
glRenderMode (GL_SELECT);
```

This places us in selection mode, which means that a scene is processed through the viewing pipeline but not stored in the frame buffer. A record of information for each object that would have been displayed in the normal rendering mode is placed in the pick buffer. In addition, this command returns the number of picked objects, which is equal to the number of information records in the pick buffer. To return to the normal rendering mode (the default), we invoke the `glRenderMode` routine using the argument `GL_RENDER`. A third option is the argument `GL_FEEDBACK`, which stores object coordinates and other information in a feedback buffer without displaying the objects. Feedback mode is used to obtain information about primitive types, attributes, and other parameters associated with the objects in a scene.

We use the following statement to activate the integer-ID name stack for the picking operations:

```
glInitNames ( );
```

The ID stack is initially empty, and this stack can be used only in selection mode. To place an unsigned integer value on the stack, we can invoke the following function:

```
glPushName (ID);
```

This places the value for parameter `ID` on the top of the stack and pushes the previous top name down to the next position in the stack. We can also simply replace the top of the stack using

```
glLoadName (ID);
```

but we cannot use this command to place a value on an empty stack. To eliminate the top of the ID stack, we issue the command

```
glPopName ( );
```

A pick window within a selected viewport is defined using the following GLU function:

```
gluPickMatrix (xPick, yPick, widthPick, heightPick, vpArray);
```

Parameters `xPick` and `yPick` give the double-precision, screen-coordinate location for the center of the pick window relative to the lower-left corner of the viewport. When these coordinates are given with mouse input, the mouse coordinates are relative to the upper-left corner, and thus we need to invert the input `yMouse` value. The double-precision values for the width and height of the pick window are specified with parameters `widthPick` and `heightPick`. Parameter `vpArray` designates an integer array containing the coordinate position and size parameters for the current viewport. We can obtain the viewport parameters using the `glGetIntegerv` function. This pick window is then used as a clipping window to construct a revised view volume for the viewing transformations. Information for objects that intersect this revised view volume is placed in the pick buffer.

We illustrate the OpenGL picking operations in the following program, which displays three color rectangles with the colors red, blue, and green. For this picking example, we use a 5×5 pick window, and the center of the pick window is given

with mouse input. Therefore, we need to invert the input `yMouse` value using the viewport height, which is the fourth element of the array `vpArray`. The red rectangle is assigned ID = 30, the blue rectangle is assigned ID = 10, and the green rectangle is assigned ID = 20. Depending on the input mouse position, we can pick no rectangles, one rectangle, two of the rectangles, or all three rectangles at one time. The rectangle identifiers are entered into the ID stack in the color order: red, blue, green. Therefore, when we process a picked rectangle, we could use either its identifier or its stack position number. For example, if the stack position number, which is the first item in the pick record, is 2, then we have picked the blue rectangle and there are two rectangle identifiers listed at the end of the record. Alternatively, we could use the last entry in the record, which is the identifier for the picked object. In this example program, we simply list the contents of the pick buffer. The rectangles are defined in the xy plane, so all depth values are 0. A sample output is given in Example 1 for a mouse input position that is near the boundary between the red and green rectangles. No mechanism is provided for terminating the program, so any number of mouse inputs can be processed.

```
#include <GL/glut.h>
#include <stdio.h>

const GLint pickBuffSize = 32;

/*  Set initial display-window size.  */
GLsizei winWidth = 400, winHeight = 400;


void init (void)
{
   /*  Set display-window color to white.  */
   glClearColor (1.0, 1.0, 1.0, 1.0);
}

/*  Define 3 rectangles and associated IDs.  */
void rects (GLenum mode)
{
   if (mode == GL_SELECT)
      glPushName (30);          //  Red rectangle.
   glColor3f (1.0, 0.0, 0.0);
   glRecti (40, 130, 150, 260);

   if (mode == GL_SELECT)
      glPushName (10);          //  Blue rectangle.
   glColor3f (0.0, 0.0, 1.0);
   glRecti (150, 130, 260, 260);

   if (mode == GL_SELECT)
      glPushName (20);          //  Green rectangle.
   glColor3f (0.0, 1.0, 0.0);
   glRecti (40, 40, 260, 130);
}

/*  Print the contents of the pick buffer for each mouse selection.  */
void processPicks (GLint nPicks, GLuint pickBuffer [ ])
```

```
{
   GLint j, k;
   GLuint objID, *ptr;

   printf (" Number of objects picked = %d\n", nPicks);
   printf ("\n");
   ptr = pickBuffer;

   /*  Output all items in each pick record.  */
   for (j = 0; j < nPicks; j++) {
      objID = *ptr;

      printf ("   Stack position = %d\n", objID);
      ptr++;

      printf ("   Min depth = %g,", float (*ptr/0x7fffffff));
      ptr++;

      printf ("   Max depth = %g\n", float (*ptr/0x7fffffff));
      ptr++;

      printf ("   Stack IDs are: \n");
      for (k = 0; k < objID; k++) {
         printf ("   %d ",*ptr);
         ptr++;
      }
      printf ("\n\n");
   }
}

void pickRects (GLint button, GLint action, GLint xMouse, GLint yMouse)
{
   GLuint pickBuffer [pickBuffSize];
   GLint nPicks, vpArray [4];

   if (button != GLUT_LEFT_BUTTON || action != GLUT_DOWN)
      return;

   glSelectBuffer (pickBuffSize, pickBuffer);  //  Designate pick buffer.
   glRenderMode (GL_SELECT);             //  Activate picking operations.
   glInitNames ( );                   //  Initialize the object-ID stack.

   /*  Save current viewing matrix.  */
   glMatrixMode (GL_PROJECTION);
   glPushMatrix ( );
   glLoadIdentity ( );

   /*  Obtain the parameters for the current viewport.  Set up
    *  a 5 x 5 pick window, and invert the input yMouse value
    *  using the height of the viewport, which is the fourth
    *  element of vpArray.
    */
   glGetIntegerv (GL_VIEWPORT, vpArray);
   gluPickMatrix (GLdouble (xMouse), GLdouble (vpArray [3] - yMouse),
                     5.0, 5.0, vpArray);
```

```
      gluOrtho2D (0.0, 300.0, 0.0, 300.0);
      rects (GL_SELECT);       // Process the rectangles in selection mode.

      /*  Restore original viewing matrix.  */
      glMatrixMode (GL_PROJECTION);
      glPopMatrix ( );

      glFlush ( );

      /*  Determine the number of picked objects and return to the
       *  normal rendering mode.
       */
      nPicks = glRenderMode (GL_RENDER);

      processPicks (nPicks, pickBuffer);  // Process picked objects.

      glutPostRedisplay ( );
   }

   void displayFcn (void)
   {
      glClear (GL_COLOR_BUFFER_BIT);
      rects (GL_RENDER);               // Display the rectangles.
      glFlush ( );
   }

   void winReshapeFcn (GLint newWidth, GLint newHeight)
   {
      /*  Reset viewport and projection parameters.  */
      glViewport (0, 0, newWidth, newHeight);
      glMatrixMode (GL_PROJECTION);
      glLoadIdentity ( );

      gluOrtho2D (0.0, 300.0, 0.0, 300.0);
      glMatrixMode (GL_MODELVIEW);

      /* Reset display-window size parameters.  */
      winWidth  = newWidth;
      winHeight = newHeight;
   }

   void main (int argc, char** argv)
   {
      glutInit (&argc, argv);
      glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
      glutInitWindowPosition (100, 100);
      glutInitWindowSize (winWidth, winHeight);
      glutCreateWindow ("Example Pick Program");

      init ( );
      glutDisplayFunc (displayFcn);
      glutReshapeFunc (winReshapeFcn);
      glutMouseFunc (pickRects);

      glutMainLoop ( );
}
```

EXAMPLE 1 **Sample Output from Procedure** `pickrects`.

```
Number of objects picked = 2

  Stack position = 1
  Min depth = 0,   Max depth = 0
  Stack IDs are:
  30

  Stack position = 3
  Min depth = 0,   Max depth = 0
  Stack IDs are:
  30    10    20
```

7 OpenGL Menu Functions

In addition to the input-device routines, GLUT contains various functions for adding simple pop-up menus to programs. With these functions, we can set up and access a variety of menus and associated submenus. The GLUT menu commands are placed in procedure `main` along with the other GLUT functions.

Creating a GLUT Menu

A pop-up menu is created with the statement

```
glutCreateMenu (menuFcn);
```

where parameter `menuFcn` is the name of a procedure that is to be invoked when a menu entry is selected. This procedure has one argument, which is the integer value corresponding to the position of a selected option.

```
void menuFcn (GLint menuItemNumber)
```

The integer value passed to parameter `menuItemNumber` is then used by `menuFcn` to perform an operation. When a menu is created, it is associated with the current display window.

Once we have designated the menu function that is to be invoked when a menu item is selected, we must specify the options that are to be listed in the menu. We do this with a series of statements that list the name and position for each option. These statements have the general form

```
glutAddMenuEntry (charString, menuItemNumber);
```

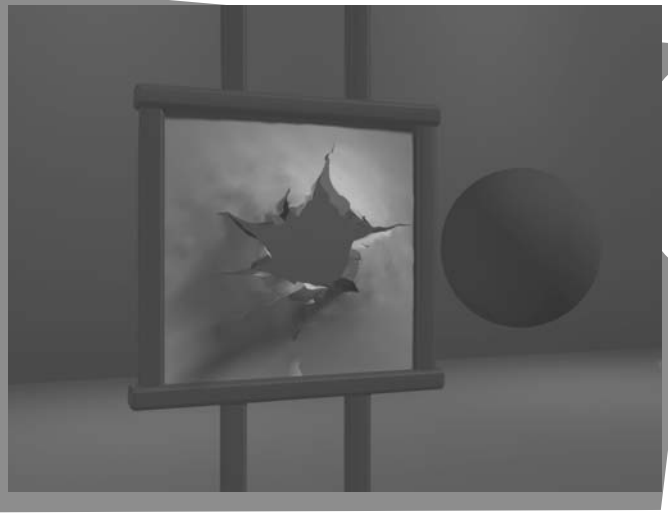
Parameter `charString` specifies text that is to be displayed in the menu, and parameter `menuItemNumber` gives the location for that entry in the menu. For example, the following statements create a menu with two options:

```
glutCreateMenu (menuFcn);
   glutAddMenuEntry ("First Menu Item", 1);
   glutAddMenuEntry ("Second Menu Item", 2);
```

Next, we must specify a mouse button that is to be used to select a menu option. This is accomplished with

```
glutAttachMenu (button);
```

Computer Animation



Computer-graphics methods are now commonly used to produce animations for a variety of applications, including entertainment (motion pictures and cartoons), advertising, scientific and engineering studies, and training and education. Although we tend to think of animation as implying object motion, the term **computer animation** generally refers to any time sequence of visual changes in a picture. In addition to changing object positions using translations or rotations, a computer-generated animation could display time variations in object size, color, transparency, or surface texture. Advertising animations often transition one object shape into another: for example, transforming a can of motor oil into an automobile engine. We can also generate computer animations by varying camera parameters, such as position, orientation, or focal length, and variations in lighting effects or other parameters and procedures associated with illumination and rendering can be used to produce computer animations.

Another consideration in computer-generated animation is realism. Many applications require realistic displays. An accurate

representation of the shape of a thunderstorm or other natural phenomena described with a numerical model is important for evaluating the reliability of the model. Similarly, simulators for training aircraft pilots and heavy-equipment operators must produce reasonably accurate representations of the environment. Entertainment and advertising applications, on the other hand, are sometimes more interested in visual effects. Thus, scenes may be displayed with exaggerated shapes and unrealistic motions and transformations. However, there are many entertainment and advertising applications that do require accurate representations for computer-generated scenes. Also, in some scientific and engineering studies, realism is not a goal. For example, physical quantities are often displayed with pseudo-colors or abstract shapes that change over time to help the researcher understand the nature of the physical process.

Two basic methods for constructing a motion sequence are **real-time animation** and **frame-by-frame animation.** In a real-time computer-animation, each stage of the sequence is viewed as it is created. Thus the animation must be generated at a rate that is compatible with the constraints of the refresh rate. For a frame-by-frame animation, each frame of the motion is separately generated and stored. Later, the frames can be recorded on film, or they can be displayed consecutively on a video monitor in "real-time playback" mode. Simple animation displays are generally produced in real time, while more complex animations are constructed more slowly, frame by frame. However, some applications require real-time animation, regardless of the complexity of the animation. A flight-simulator animation, for example, is produced in real time because the video displays must be generated in immediate response to changes in the control settings. In such cases, special hardware and software systems are often developed to allow the complex display sequences to be developed quickly.

1 Raster Methods for Computer Animation

Most of the time, we can create simple animation sequences in our programs using real-time methods. In general, though, we can produce an animation sequence on a raster-scan system one frame at a time, so that each completed frame could be saved in a file for later viewing. The animation can then be viewed by cycling through the completed frame sequence, or the frames could be transferred to film. If we want to generate an animation in real time, however, we need to produce the motion frames quickly enough so that a continuous motion sequence is displayed. For a complex scene, one frame of the animation could take most of the refresh cycle time to construct. In that case, objects generated first would be displayed for most of the frame refresh time, but objects generated toward the end of the refresh cycle would disappear almost as soon as they were displayed. For very complex animations, the frame construction time could be greater than the time to refresh the screen, which can lead to erratic motion and fractured frame displays. Because the screen display is generated from successively modified pixel values in the refresh buffer, we can take advantage of some of the characteristics of the raster screen-refresh process to produce motion sequences quickly.

Double Buffering

One method for producing a real-time animation with a raster system is to employ two refresh buffers. Initially, we create a frame for the animation in one

of the buffers. Then, while the screen is being refreshed from that buffer, we construct the next frame in the other buffer. When that frame is complete, we switch the roles of the two buffers so that the refresh routines use the second buffer during the process of creating the next frame in the first buffer. This alternating buffer process continues throughout the animation. Graphics libraries that permit such operations typically have one function for activating the double-buffering routines and another function for interchanging the roles of the two buffers.

When a call is made to switch two refresh buffers, the interchange could be performed at various times. The most straightforward implementation is to switch the two buffers at the end of the current refresh cycle, during the vertical retrace of the electron beam. If a program can complete the construction of a frame within the time of a refresh cycle, say $\frac{1}{60}$ of a second, each motion sequence is displayed in synchronization with the screen refresh rate. However, if the time to construct a frame is longer than the refresh time, the current frame is displayed for two or more refresh cycles while the next animation frame is being generated. For example, if the screen refresh rate is 60 frames per second and it takes $\frac{1}{50}$ of a second to construct an animation frame, each frame is displayed on the screen twice and the animation rate is only 30 frames each second. Similarly, if the frame construction time is $\frac{1}{25}$ of a second, the animation frame rate is reduced to 20 frames per second because each frame is displayed three times.

Irregular animation frame rates can occur with double buffering when the frame construction time is very nearly equal to an integer multiple of the screen refresh time. As an example of this, if the screen refresh rate is 60 frames per second, then an erratic animation frame rate is possible when the frame construction time is very close to $\frac{1}{60}$ of a second, or $\frac{2}{60}$ of a second, or $\frac{3}{60}$ of a second, and so forth. Because of slight variations in the implementation time for the routines that generate the primitives and their attributes, some frames could take a little more time to construct and some a little less time. Thus, the animation frame rate can change abruptly and erratically. One way to compensate for this effect is to add a small time delay to the program. Another possibility is to alter the motion or scene description to shorten the frame construction time.

Generating Animations Using Raster Operations

We can also generate real-time raster animations for limited applications using block transfers of a rectangular array of pixel values. This animation technique is often used in game-playing programs. A simple method for translating an object from one location to another in the xy plane is to transfer the group of pixel values that define the shape of the object to the new location. Two-dimensional rotations in multiples of 90^o are also simple to perform, although we can rotate rectangular blocks of pixels through other angles using antialiasing procedures. For a rotation that is not a multiple of 90^o, we need to estimate the percentage of area coverage for those pixels that overlap the rotated block. Sequences of raster operations can be executed to produce realtime animation for either two-dimensional or three-dimensional objects, so long as we restrict the animation to motions in the projection plane. Then no viewing or visible-surface algorithms need be invoked.

We can also animate objects along two-dimensional motion paths using **color-table transformations.** Here we predefine the object at successive positions along the motion path and set the successive blocks of pixel values to color-table entries. The pixels at the first position of the object are set to a foreground color, and the pixels at the other object positions are set to the background color. The animation

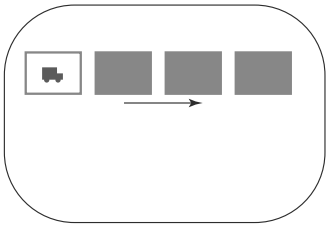


FIGURE 1
Real-time raster color-table animation.

is then accomplished by changing the color-table values so that the object color at successive positions along the animation path becomes the foreground color as the preceding position is set to the background color (Figure 1).

2 Design of Animation Sequences

Constructing an animation sequence can be a complicated task, particularly when it involves a story line and multiple objects, each of which can move in a different way. A basic approach is to design such animation sequences using the following development stages:

- Storyboard layout
- Object definitions
- Key-frame specifications
- Generation of in-between frames

The **storyboard** is an outline of the action. It defines the motion sequence as a set of basic events that are to take place. Depending on the type of animation to be produced, the storyboard could consist of a set of rough sketches, along with a brief description of the movements, or it could just be a list of the basic ideas for the action. Originally, the set of motion sketches was attached to a large board that was used to present an overall view of the animation project. Hence, the name "storyboard."

An **object definition** is given for each participant in the action. Objects can be defined in terms of basic shapes, such as polygons or spline surfaces. In addition, a description is often given of the movements that are to be performed by each character or object in the story.

A **key frame** is a detailed drawing of the scene at a certain time in the animation sequence. Within each key frame, each object (or character) is positioned according to the time for that frame. Some key frames are chosen at extreme positions in the action; others are spaced so that the time interval between key frames is not too great. More key frames are specified for intricate motions than for simple, slowly varying motions. Development of the key frames is generally the responsibility of the senior animators, and often a separate animator is assigned to each character in the animation.

In-betweens are the intermediate frames between the key frames. The total number of frames, and hence the total number of in-betweens, needed for an animation is determined by the display media that is to be used. Film requires 24 frames per second, and graphics terminals are refreshed at the rate of 60 or more frames per second. Typically, time intervals for the motion are set up so that there are from three to five in-betweens for each pair of key frames. Depending on the speed specified for the motion, some key frames could be duplicated. As an example, a 1-minute film sequence with no duplication requires a total of 1,440 frames. If five in-betweens are required for each pair of key frames, then 288 key frames would need to be developed.

There are several other tasks that may be required, depending on the application. These additional tasks include motion verification, editing, and the production and synchronization of a soundtrack. Many of the functions needed to produce general animations are now computer-generated. Figures 2 and 3 show examples of computer-generated frames for animation sequences.

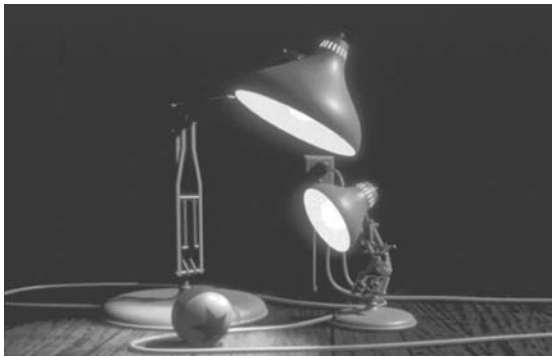


FIGURE 2
One frame from the award-winning computer-animated short film *Luxo Jr.* The film was designed using a key-frame animation system and cartoon animation techniques to provide lifelike actions of the lamps. Final images were rendered with multiple light sources and procedural texturing techniques. (*Courtesy of Pixar.* © *1986 Pixar.*)

FIGURE 3
One frame from the short film *Tin Toy,* the first computer-animated film to win an Oscar. Designed using a key-frame animation system, the film also required extensive facial-expression modeling. Final images were rendered using procedural shading, self-shadowing techniques, motion blur, and texture mapping. (*Courtesy of Pixar.* © *1988 Pixar.*)

3 Traditional Animation Techniques

Film animators use a variety of methods for depicting and emphasizing motion sequences. These include object deformations, spacing between animation frames, motion anticipation and follow-through, and action focusing.

One of the most important techniques for simulating acceleration effects, particularly for nonrigid objects, is **squash and stretch.** Figure 4 shows how this technique is used to emphasize the acceleration and deceleration of a bouncing ball. As the ball accelerates, it begins to stretch. When the ball hits the floor and stops, it is first compressed (squashed) and then stretched again as it accelerates and bounces upwards.

Another technique used by film animators is **timing,** which refers to the spacing between motion frames. A slower moving object is represented with more closely spaced frames, and a faster moving object is displayed with fewer frames over the path of the motion. This effect is illustrated in Figure 5, where the position changes between frames increase as a bouncing ball moves faster.

Object movements can also be emphasized by creating preliminary actions that indicate an **anticipation** of a coming motion. For example, a cartoon character

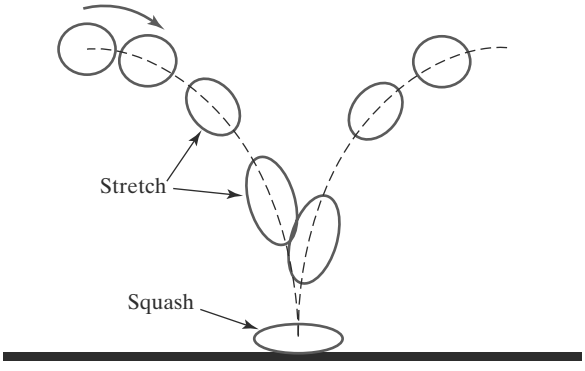


FIGURE 4
A bouncing-ball illustration of the "squash and stretch" technique for emphasizing object acceleration.

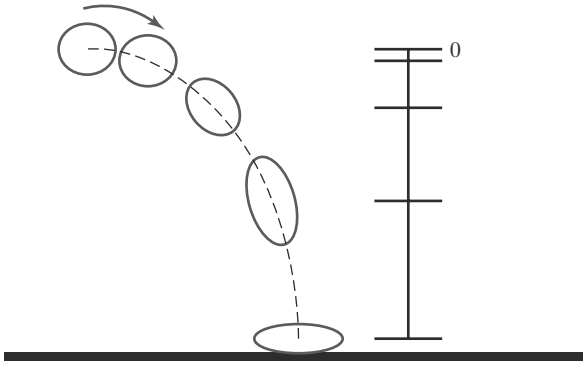


FIGURE 5
The position changes between motion frames for a bouncing ball increase as the speed of the ball increases.

might lean forward and rotate its body before starting to run; or a character might perform a "windup" before throwing a ball. Similarly, **follow-through actions** can be used to emphasize a previous motion. After throwing a ball, a character can continue the arm swing back to its body; or a hat can fly off a character that is stopped abruptly. An action also can be emphasized with **staging,** which refers to any method for focusing on an important part of a scene, such as a character hiding something.

4 General Computer-Animation Functions

Many software packages have been developed either for general animation design or for performing specialized animation tasks. Typical animation functions include managing object motions, generating views of objects, producing camera motions, and the generation of in-between frames. Some animation packages, such as Wavefront for example, provide special functions for both the overall animation design and the processing of individual objects. Others are special-purpose packages for particular features of an animation, such as a system for generating in-between frames or a system for figure animation.

A set of routines is often provided in a general animation package for storing and managing the object database. Object shapes and associated parameters are stored and updated in the database. Other object functions include those for generating the object motion and those for rendering the object surfaces. Movements can be generated according to specified constraints using two-dimensional or three-dimensional transformations. Standard functions can then be applied to identify visible surfaces and apply the rendering algorithms.

Another typical function set simulates camera movements. Standard camera motions are zooming, panning, and tilting. Finally, given the specification for the key frames, the in-betweens can be generated automatically.

5 Computer-Animation Languages

We can develop routines to design and control animation sequences within a general-purpose programming language, such as C, C++, Lisp, or Fortran, but several specialized animation languages have been developed. These languages typically include a graphics editor, a key-frame generator, an in-between generator, and standard graphics routines. The graphics editor allows an animator to design and modify object shapes, using spline surfaces, constructive solid-geometry methods, or other representation schemes.

An important task in an animation specification is *scene description*. This includes the positioning of objects and light sources, defining the photometric parameters (light-source intensities and surface illumination properties), and setting the camera parameters (position, orientation, and lens characteristics). Another standard function is *action specification,* which involves the layout of motion paths for the objects and camera. We need the usual graphics routines: viewing and perspective transformations, geometric transformations to generate object movements as a function of accelerations or kinematic path specifications, visible-surface identification, and the surface-rendering operations.

Key-frame systems were originally designed as a separate set of animation routines for generating the in-betweens from the user-specified key frames. Now, these routines are often a component in a more general animation package. In the simplest case, each object in a scene is defined as a set of rigid bodies connected at the joints and with a limited number of degrees of freedom. As an example, the

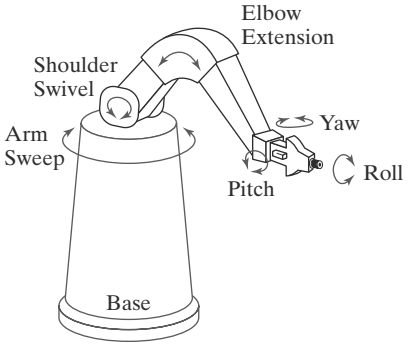


FIGURE 6
Degrees of freedom for a stationary, single-armed robot.

single-armed robot in Figure 6 has 6 degrees of freedom, which are referred to as arm sweep, shoulder swivel, elbow extension, pitch, yaw, and roll. We can extend the number of degrees of freedom for this robot arm to 9 by allowing three-dimensional translations for the base (Figure 7). If we also allow base rotations, the robot arm can have a total of 12 degrees of freedom. The human body, in comparison, has more than 200 degrees of freedom.

Parameterized systems allow object motion characteristics to be specified as part of the object definitions. The adjustable parameters control such object characteristics as degrees of freedom, motion limitations, and allowable shape changes.

Scripting systems allow object specifications and animation sequences to be defined with a user-input *script*. From the script, a library of various objects and motions can be constructed.

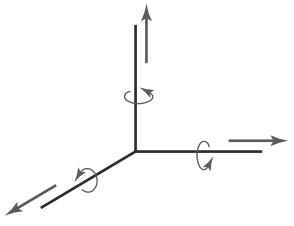


FIGURE 7
Translational and rotational degrees of freedom for the base of the robot arm.

6 Key-Frame Systems

A set of in-betweens can be generated from the specification of two (or more) key frames using a key-frame system. Motion paths can be given with a *kinematic description* as a set of spline curves, or the motions can be *physically based* by specifying the forces acting on the objects to be animated.

For complex scenes, we can separate the frames into individual components or objects called **cels** (celluloid transparencies). This term developed from cartoon-animation techniques where the background and each character in a scene were placed on a separate transparency. Then, with the transparencies stacked in the order from background to foreground, they were photographed to obtain the completed frame. The specified animation paths are then used to obtain the next cel for each character, where the positions are interpolated from the key-frame times.

With complex object transformations, the shapes of objects may change over time. Examples are clothes, facial features, magnified detail, evolving shapes, and exploding or disintegrating objects. For surfaces described with polygon meshes, these changes can result in significant changes in polygon shape such that the number of edges in a polygon could be different from one frame to the next. These changes are incorporated into the development of the in-between frames by adding or subtracting polygon edges according to the requirements of the defining key frames.

Morphing

Transformation of object shapes from one form to another is termed **morphing,** which is a shortened form of "metamorphosing." An animator can model morphing by transitioning polygon shapes through the in-betweens from one key frame to the next.

Given two key frames, each with a different number of line segments specifying an object transformation, we can first adjust the object specification in one of the frames so that the number of polygon edges (or the number of polygon vertices) is the same for the two frames. This preprocessing step is illustrated in Figure 8. A straight-line segment in key frame k is transformed into two line segments in key frame $k + 1$. Because key frame $k + 1$ has an extra vertex, we add a vertex between vertices 1 and 2 in key frame k to balance the number of vertices (and edges) in the two key frames. Using linear interpolation to generate the in-betweens, we transition the added vertex in key frame k into vertex 3′ along the straight-line path shown in Figure 9. An example of a triangle linearly expanding into a quadrilateral is given in Figure 10.

We can state general preprocessing rules for equalizing key frames in terms of either the number of edges or the number of vertices to be added to a key frame. We first consider equalizing the edge count, where parameters L_k and L_{k+1} denote the number of line segments in two consecutive frames. The maximum and minimum number of lines to be equalized can be determined as

$$L_{\max} = \max(L_k, L_{k+1}), \qquad L_{\min} = \min(L_k, L_{k+1}) \tag{1}$$

Next we compute the following two quantities:

$$N_e = L_{\max} \bmod L_{\min}$$
$$N_s = int\left(\frac{L_{\max}}{L_{\min}}\right) \tag{2}$$

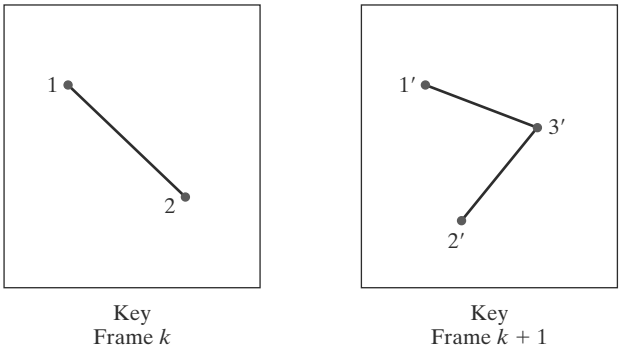


FIGURE 8
An edge with vertex positions 1 and 2 in key frame k evolves into two connected edges in key frame $k + 1$.

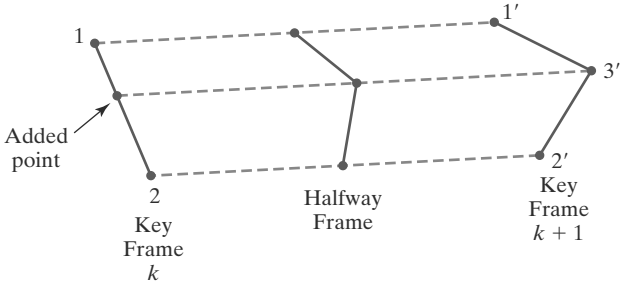


FIGURE 9
Linear interpolation for transforming a line segment in key frame k into two connected line segments in key frame $k + 1$.

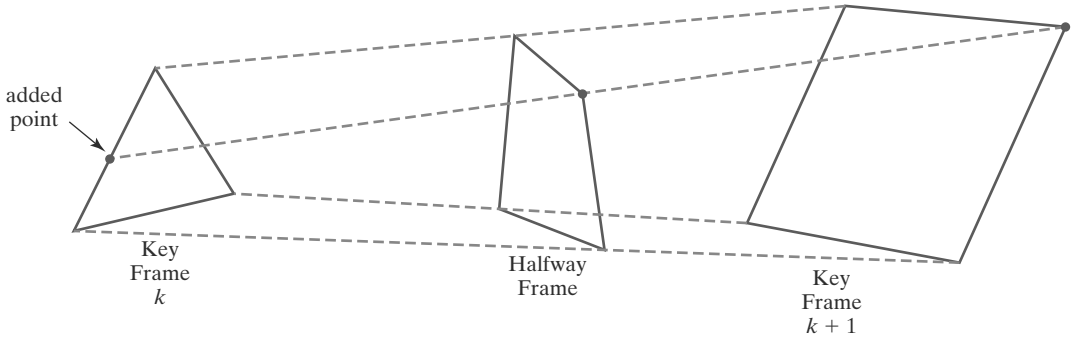


FIGURE 10
Linear interpolation for transforming a triangle into a quadrilateral.

The preprocessing steps for edge equalization are then accomplished with the following two procedures:

1. Divide N_e edges of $keyframe_{\min}$ into $N_s + 1$ sections.
2. Divide the remaining lines of $keyframe_{\min}$ into N_s sections.

As an example, if $L_k = 15$ and $L_{k+1} = 11$, we would divide four lines of $keyframe_{k+1}$ into two sections each. The remaining lines of $keyframe_{k+1}$ are left intact.

If we equalize the vertex count, we can use parameters V_k and V_{k+1} to denote the number of vertices in the two consecutive key frames. In this case, we determine the maximum and minimum number of vertices as

$$V_{\max} = \max(V_k, V_{k+1}), \qquad V_{\min} = \min(V_k, V_{k+1}) \tag{3}$$

Then we compute the following two values:

$$\begin{aligned} N_{ls} &= (V_{\max} - 1) \bmod (V_{\min} - 1) \\ N_p &= int\left(\frac{V_{\max} - 1}{V_{\min} - 1}\right) \end{aligned} \tag{4}$$

These two quantities are then used to perform vertex equalization with the following procedures:

1. Add N_p points to N_{ls} line sections of $keyframe_{\min}$.
2. Add $N_p - 1$ points to the remaining edges of $keyframe_{\min}$.

For the triangle-to-quadrilateral example, $V_k = 3$ and $V_{k+1} = 4$. Both N_{ls} and N_p are 1, so we would add one point to one edge of $keyframe_k$. No points would be added to the remaining lines of $keyframe_k$.

Simulating Accelerations

Curve-fitting techniques are often used to specify the animation paths between key frames. Given the vertex positions at the key frames, we can fit the positions with linear or nonlinear paths. Figure 11 illustrates a nonlinear fit of key-frame positions. To simulate accelerations, we can adjust the time spacing for the in-betweens.

If the motion is to occur at constant speed (zero acceleration), we use equal-interval time spacing for the in-betweens. For instance, with n in-betweens and

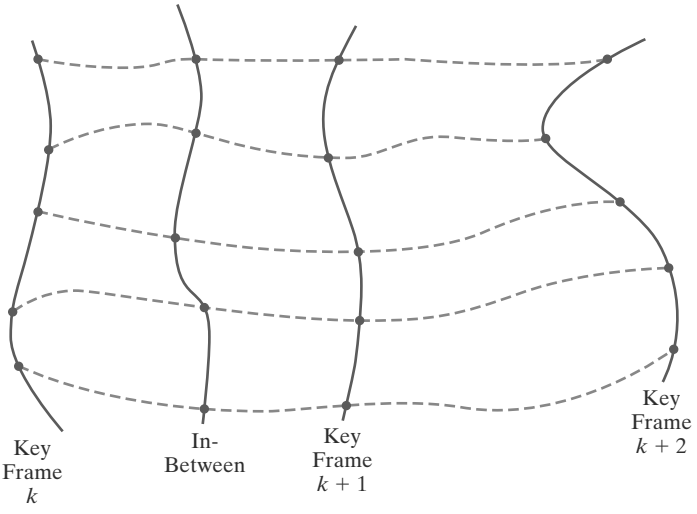


FIGURE 11
Fitting key-frame vertex positions with nonlinear splines.

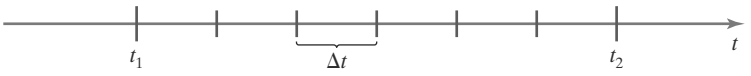


FIGURE 12
In-between positions for motion at constant speed.

key-frame times of t_1 and t_2 (Figure 12), the time interval between the key frames is divided into $n+1$ equal subintervals, yielding an in-between spacing of

$$\Delta t = \frac{t_2 - t_1}{n+1} \tag{5}$$

The time for the jth in-between is

$$tB_j = t_1 + j\Delta t, \qquad j = 1, 2, \ldots, n \tag{6}$$

and this time value is used to calculate coordinate positions, color, and other physical parameters for that frame of the motion.

Speed changes (nonzero accelerations) are usually necessary at some point in an animation film or cartoon, particularly at the beginning and end of a motion sequence. The startup and slowdown portions of an animation path are often modeled with spline or trigonometric functions, but parabolic and cubic time functions have been applied to acceleration modeling. Animation packages commonly furnish trigonometric functions for simulating accelerations.

To model increasing speed (positive acceleration), we want the time spacing between frames to increase so that greater changes in position occur as the object moves faster. We can obtain an increasing size for the time interval with the function

$$1 - \cos\theta, \qquad 0 < \theta < \pi/2$$

For n in-betweens, the time for the jth in-between would then be calculated as

$$tB_j = t_1 + \Delta t \left[1 - \cos\frac{j\pi}{2(n+1)}\right], \qquad j = 1, 2, \ldots, n \tag{7}$$

where Δt is the time difference between the two key frames. Figure 13 gives a plot of the trigonometric acceleration function and the in-between spacing for $n = 5$.

We can model decreasing speed (deceleration) using the function $\sin\theta$, with $0 < \theta < \pi/2$. The time position of an in-between is then determined as

$$tB_j = t_1 + \Delta t \sin\frac{j\pi}{2(n+1)}, \qquad j = 1, 2, \ldots, n \tag{8}$$

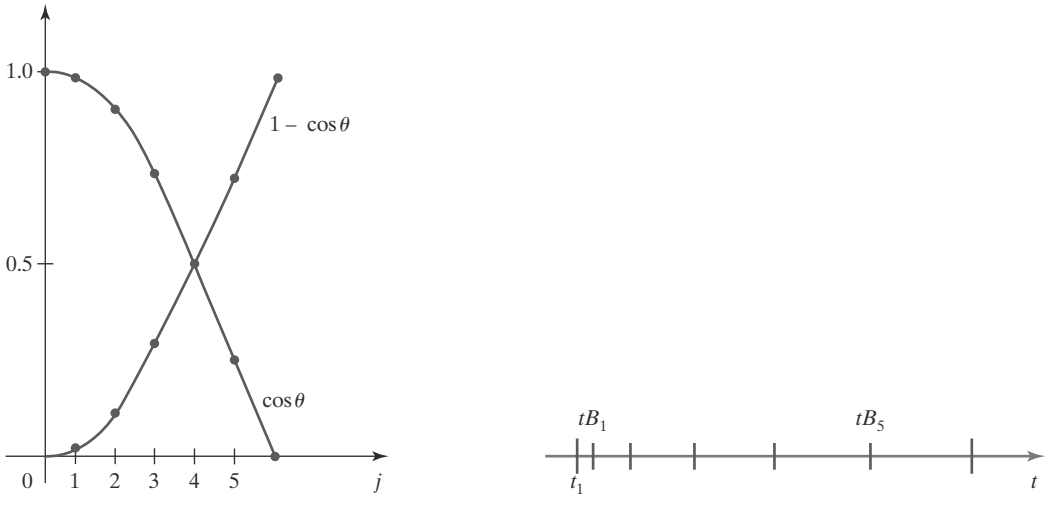


FIGURE 13
A trigonometric acceleration function and the corresponding in-between spacing for $n = 5$ and $\theta = j\pi/12$ in Equation 7, producing increased coordinate changes as the object moves through each time interval.

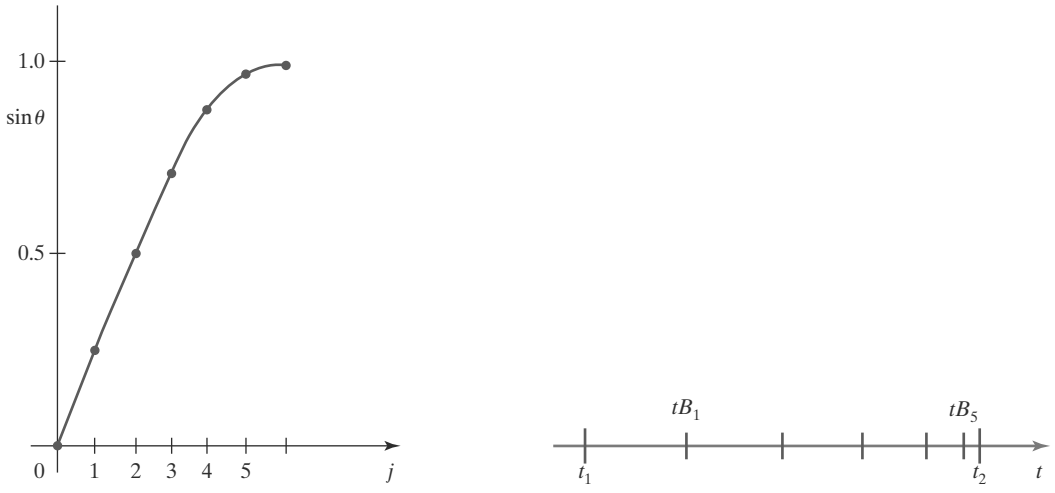


FIGURE 14
A trigonometric deceleration function and the corresponding in-between spacing for $n = 5$ and $\theta = j\pi/12$ in Equation 8, producing decreased coordinate changes as the object moves through each time interval.

A plot of this function and the decreasing size of the time intervals is shown in Figure 14 for five in-betweens.

Often, motions contain both speedups and slowdowns. We can model a combination of increasing–decreasing speed by first increasing the in-between time spacing and then decreasing this spacing. A function to accomplish these time changes is

$$\frac{1}{2}(1 - \cos\theta), \qquad 0 < \theta < \pi/2$$

The time for the jth in-between is now calculated as

$$tB_j = t_1 + \Delta t \left\{ \frac{1 - \cos[j\pi/(n+1)]}{2} \right\}, \qquad j = 1, 2, \ldots, n \tag{9}$$

with Δt denoting the time difference between the two key frames. Time intervals for a moving object first increase and then decrease, as shown in Figure 15.

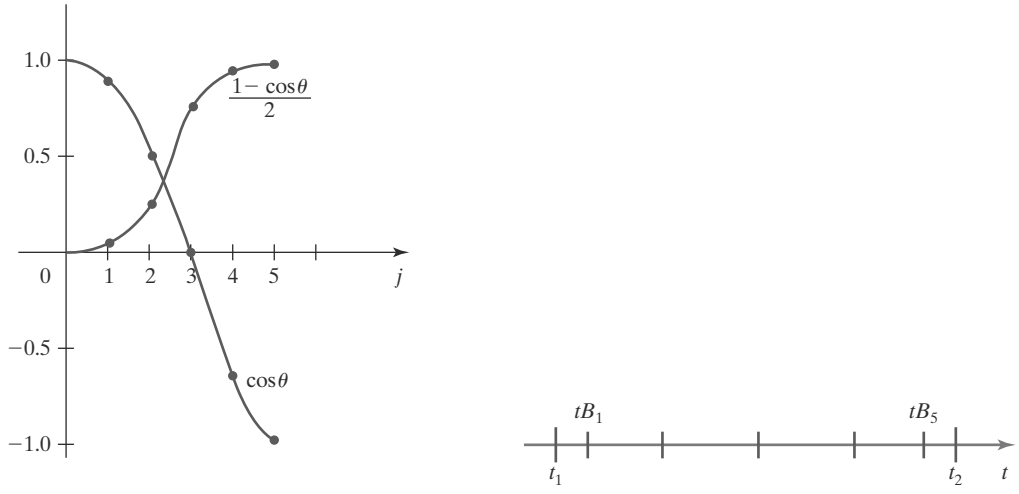


FIGURE 15
The trigonometric accelerate–decelerate function $(1 - \cos\theta)/2$ and the corresponding in-between spacing for $n = 5$ in Equation 9.

Processing the in-betweens is simplified by initially modeling "skeleton" (wire-frame) objects so that motion sequences can be interactively adjusted. After the animation sequence is completely defined, objects can be fully rendered.

7 Motion Specifications

General methods for describing an animation sequence range from an explicit specification of the motion paths to a description of the interactions that produce the motions. Thus, we could define how an animation is to take place by giving the transformation parameters, the motion path parameters, the forces that are to act on objects, or the details of how objects interact to produce motion.

Direct Motion Specification

The most straightforward method for defining an animation is *direct motion specification* of the geometric-transformation parameters. Here, we explicitly set the values for the rotation angles and translation vectors. Then the geometric transformation matrices are applied to transform coordinate positions. Alternatively, we could use an approximating equation involving these parameters to specify certain kinds of motions. We can approximate the path of a bouncing ball, for instance, with a damped, rectified, sine curve (Figure 16):

$$y(x) = A|\sin(\omega x + \theta_0)|e^{-kx} \tag{10}$$

where A is the initial amplitude (height of the ball above the ground), ω is the angular frequency, θ_0 is the phase angle, and k is the damping constant. This method for motion specification is particularly useful for simple user-programmed animation sequences.

Goal-Directed Systems

At the opposite extreme, we can specify the motions that are to take place in general terms that abstractly describe the actions in terms of the final results. In other words, an animation is specified in terms of the final state of the movements. These systems are referred to as *goal-directed*, since values for the motion parameters are determined from the goals of the animation. For example, we could specify that

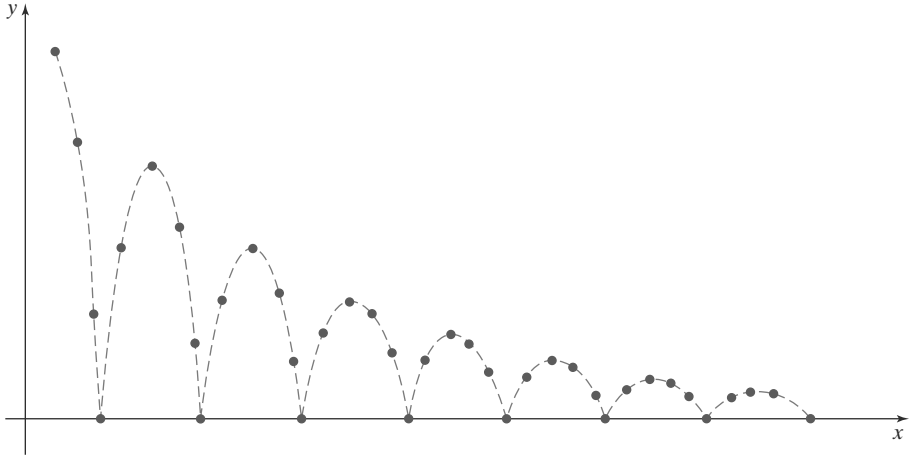


FIGURE 16
Approximating the motion of a bouncing ball with a damped sine function (Eq. 10).

we want an object to "walk" or to "run" to a particular destination; or we could state that we want an object to "pick up" some other specified object. The input directives are then interpreted in terms of component motions that will accomplish the described task. Human motions, for instance, can be defined as a hierarchical structure of submotions for the torso, limbs, and so forth. Thus, when a goal, such as "walk to the door" is given, the movements required of the torso and limbs to accomplish this action are calculated.

Kinematics and Dynamics

We can also construct animation sequences using *kinematic* or *dynamic* descriptions. With a kinematic description, we specify the animation by giving motion parameters (position, velocity, and acceleration) without reference to causes or goals of the motion. For constant velocity (zero acceleration), we designate the motions of rigid bodies in a scene by giving an initial position and velocity vector for each object. For example, if a velocity is specified as (3, 0, −4) km per sec, then this vector gives the direction for the straight-line motion path and the speed (magnitude of velocity) is calculated as 5 km per sec. If we also specify accelerations (rate of change of velocity), we can generate speedups, slowdowns, and curved motion paths. Kinematic specification of a motion can also be given by simply describing the motion path. This is often accomplished using spline curves.

An alternate approach is to use *inverse kinematics*. Here, we specify the initial and final positions of objects at specified times and the motion parameters are computed by the system. For example, assuming zero acceleration, we can determine the constant velocity that will accomplish the movement of an object from the initial position to the final position. This method is often used with complex objects by giving the positions and orientations of an end node of an object, such as a hand or a foot. The system then determines the motion parameters of other nodes to accomplish the desired motion.

Dynamic descriptions, on the other hand, require the specification of the forces that produce the velocities and accelerations. The description of object behavior in terms of the influence of forces is generally referred to as *physically based modeling*. Examples of forces affecting object motion include electromagnetic, gravitational, frictional, and other mechanical forces.

Object motions are obtained from the force equations describing physical laws, such as Newton's laws of motion for gravitational and frictional processes, Euler or Navier-Stokes equations describing fluid flow, and Maxwell's equations for electromagnetic forces. For example, the general form of Newton's second law for a particle of mass m is

$$\mathbf{F} = \frac{d}{dt}(m\mathbf{v}) \tag{11}$$

where $\mathbf{F}$ is the force vector and $\mathbf{v}$ is the velocity vector. If mass is constant, we solve the equation $\mathbf{F} = m\mathbf{a}$, with $\mathbf{a}$ representing the acceleration vector. Otherwise, mass is a function of time, as in relativistic motions or the motions of space vehicles that consume measurable amounts of fuel per unit time. We can also use *inverse dynamics* to obtain the forces, given the initial and final positions of objects and the type of motion required.

Applications of physically based modeling include complex rigid-body systems and such nonrigid systems as cloth and plastic materials. Typically, numerical methods are used to obtain the motion parameters incrementally from the dynamical equations using initial conditions or boundary values.

8 Character Animation

Animation of simple objects is relatively straightforward. When we consider the animation of more complex figures such as humans or animals, however, it becomes much more difficult to create realistic animation. Consider the animation of walking or running human (or humanoid) characters. Based upon observations in their own lives of walking or running people, viewers will expect to see animated characters move in particular ways. If an animated character's movement doesn't match this expectation, the believability of the character may suffer. Thus, much of the work involved in character animation is focused on creating believable movements.

Articulated Figure Animation

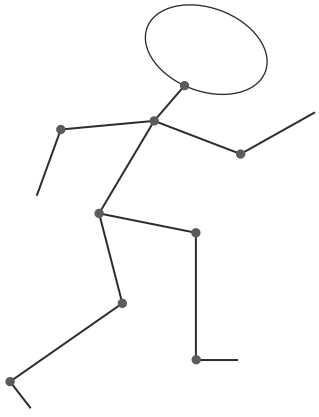


FIGURE 17
A simple articulated figure with nine joints and twelve connecting links, not counting the oval head.

A basic technique for animating people, animals, insects, and other critters is to model them as **articulated figures,** which are hierarchical structures composed of a set of rigid links that are connected at rotary joints (Figure 17). In less formal terms, this just means that we model animate objects as moving stick figures, or simplified skeletons, that can later be wrapped with surfaces representing skin, hair, fur, feathers, clothes, or other outer coverings.

The connecting points, or hinges, for an articulated figure are placed at the shoulders, hips, knees, and other skeletal joints, which travel along specified motion paths as the body moves. For example, when a motion is specified for an object, the shoulder automatically moves in a certain way and, as the shoulder moves, the arms move. Different types of movement, such as walking, running, or jumping, are defined and associated with particular motions for the joints and connecting links.

A series of walking leg motions, for instance, might be defined as in Figure 18. The hip joint is translated forward along a horizontal line, while the connecting links perform a series of movements about the hip, knee, and angle joints. Starting with a straight leg [Figure 18(a)], the first motion is a knee bend as the hip moves forward [Figure 18(b)]. Then the leg swings forward, returns to the vertical position, and swings back, as shown in Figures 18(c), (d), and (e). The final motions are a wide swing back and a return to the straight

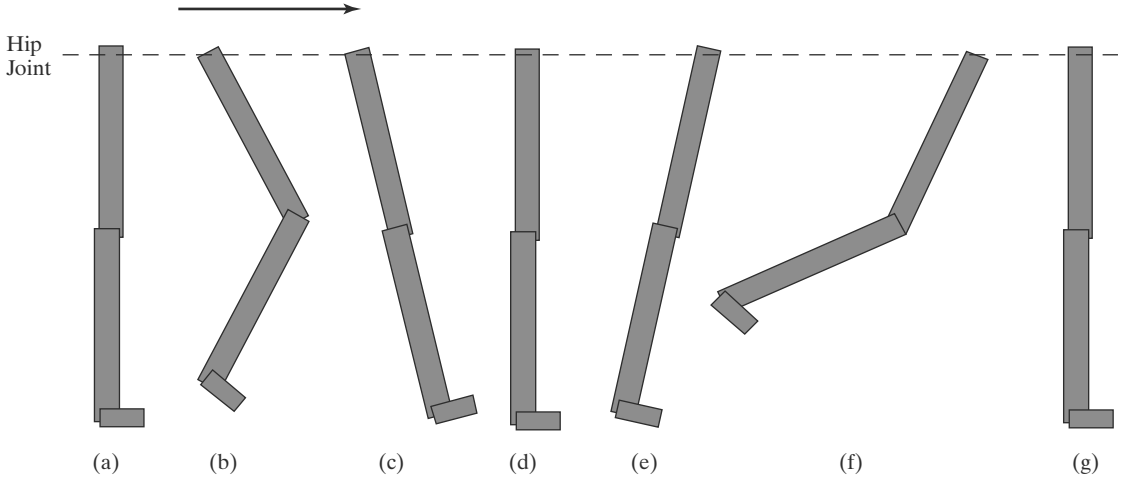


FIGURE 18
Possible motions for a set of connected links representing a walking leg.

vertical position, as in Figures 18(f) and (g). This motion cycle is repeated for the duration of the animation as the figure moves over a specified distance or time interval.

As a figure moves, other movements are incorporated into the various joints. A sinusoidal motion, often with varying amplitude, can be applied to the hips so that they move about on the torso. Similarly, a rolling or rocking motion can be imparted to the shoulders, and the head can bob up and down.

Both kinematic-motion descriptions and inverse kinematics are used in figure animations. Specifying the joint motions is generally an easier task, but inverse kinematics can be useful for producing simple motion over arbitrary terrain. For a complicated figure, inverse kinematics may not produce a unique animation sequence: Many different rotational motions may be possible for a given set of initial and final conditions. In such cases, a unique solution may be possible by adding more constraints, such as conservation of momentum, to the system.

Motion Capture

An alternative to determining the motion of a character computationally is to digitally record the movement of a live actor and to base the movement of an animated character on that information. This technique, known as *motion capture* or *mo-cap*, can be used when the movement of the character is predetermined (as in a scripted scene). The animated character will perform the same series of movements as the live actor.

The classic motion capture technique involves placing a set of markers at strategic positions on the actor's body, such as the arms, legs, hands, feet, and joints. It is possible to place the markers directly on the actor, but more commonly they are affixed to a special skintight body suit worn by the actor. The actor is them filmed performing the scene. Image processing techniques are then used to identify the positions of the markers in each frame of the film, and their positions are translated to coordinates. These coordinates are used to determine the positioning of the body of the animated character. The movement of each marker from frame to frame in the film is tracked and used to control the corresponding movement of the animated character.

To accurately determine the positions of the markers, the scene must be filmed by multiple cameras placed at fixed positions. The digitized marker data from each recording can then be used to triangulate the position of each marker in three dimensions. Typical motion capture systems will use up to two dozen cameras, but systems with several hundred cameras exist.

Optical motion capture systems rely on the reflection of light from a marker into the camera. These can be relatively simple passive systems using photo-reflective markers that reflect illumination from special lights placed near the cameras, or more advanced active systems in which the markers are powered and emit light. Active systems can be constructed so that the markers illuminate in a pattern or sequence, which allows each marker to be uniquely identified in each frame of the recording, simplifying the tracking process.

Non-optical systems rely on the direct transmission of position information from the markers to a recording device. Some non-optical systems use inertial sensors that provide gyroscope-based position and orientation information. Others use magnetic sensors that measure changes in magnetic flux. A series of transmitters placed around the stage generate magnetic fields that induce current in the magnetic sensors; that information is then transmitted to receivers.

Some motion capture systems record more than just the gross movements of the parts of the actor's body. It is possible to record even the actor's

facial movements. Often called *performance capture* systems, these typically use a camera trained on the actor's face and small light-emitting diode (LED) lights that illuminate the face. Small photoreflective markers attached to the face reflect the light from the LEDs and allow the camera to capture the small movements of the muscles of the face, which can then be used to create realistic facial animation on a computer-generated character.

9 Periodic Motions

When we construct an animation with repeated motion patterns, such as a rotating object, we need to be sure that the motion is sampled frequently enough to represent the movements correctly. In other words, the motion must be synchronized with the frame-generation rate so that we display enough frames per cycle to show the true motion. Otherwise, the animation may be displayed incorrectly.

A typical example of an undersampled periodic-motion display is the wagon wheel in a Western movie that appears to be turning in the wrong direction. Figure 19 illustrates one complete cycle in the rotation of a wagon wheel with one red spoke that makes 18 clockwise revolutions per second. If this motion is recorded on film at the standard motion-picture projection rate of 24 frames per second, then the first five frames depicting this motion would be as shown in Figure 20. Because the wheel completes $\frac{3}{4}$ of a turn every $\frac{1}{24}$ of a second, only one animation frame is generated per cycle, and the wheel thus appears to be rotating in the opposite (counterclockwise) direction.

In a computer-generated animation, we can control the sampling rate in a periodic motion by adjusting the motion parameters. For example, we can set the

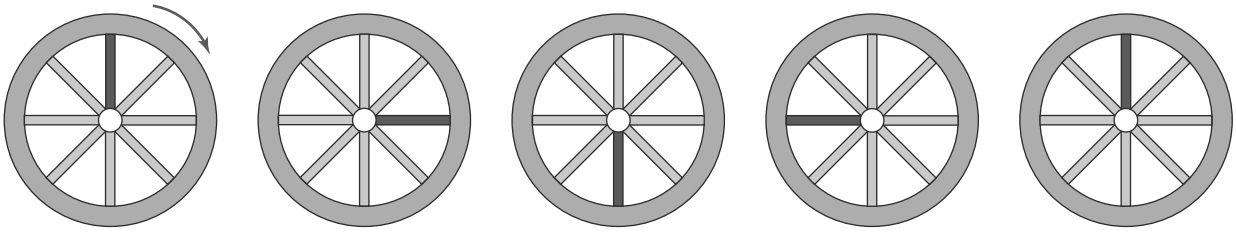


FIGURE 19
Five positions for a red spoke during one cycle of a wheel motion that is turning at the rate of 18 revolutions per second.

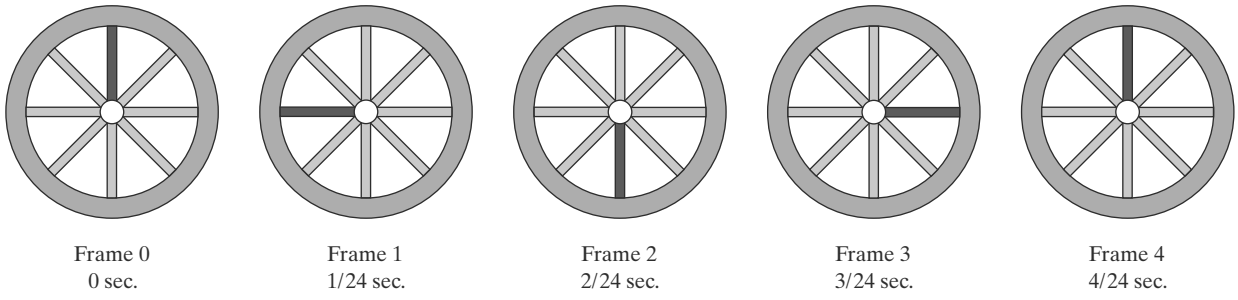


FIGURE 20
The first five film frames of the rotating wheel in Figure 19 produced at the rate of 24 frames per second.

angular increment for the motion of a rotating object so that multiple frames are generated in each revolution. Thus, a 3° increment for a rotation angle produces 120 motion steps during one revolution, and a 4° increment generates 90 steps. For faster motion, larger rotational steps could be used, so long as the number of samples per cycle is not too small and the motion is clearly displayed. When complex objects are to be animated, we also must take into account the effect that the frame construction time might have on the refresh rate, as discussed in Section 1. The motion of a complex object can be much slower than we want it to be if it takes too long to construct each frame of the animation.

Another factor that we need to consider in the display of a repeated motion is the effect of round-off in the calculations for the motion parameters. We can reset parameter values periodically to prevent the accumulated error from producing erratic motions. For a continuous rotation, we could reset parameter values once every cycle (360°).

10 OpenGL Animation Procedures

Raster operations and color-index assignment functions are available in the core library, and routines for changing color-table values are provided in GLUT. Other raster-animation operations are available only as GLUT routines because they depend on the window system in use. In addition, computer-animation features such as double buffering may not be included in some hardware systems.

Double-buffering operations, if available, are activated using the following GLUT command:

```
glutInitDisplayMode (GLUT_DOUBLE);
```

This provides two buffers, called the *front buffer* and the *back buffer,* that we can use alternately to refresh the screen display. While one buffer is acting as the refresh buffer for the current display window, the next frame of an animation can be constructed in the other buffer. We specify when the roles of the two buffers are to be interchanged using

```
glutSwapBuffers ( );
```

To determine whether double-buffer operations are available on a system, we can issue the following query:

```
glGetBooleanv (GL_DOUBLEBUFFER, status);
```

A value of `GL_TRUE` is returned to array parameter `status` if both front and back buffers are available on a system. Otherwise, the returned value is `GL_FALSE`.

For a continuous animation, we can also use

```
glutIdleFunc (animationFcn);
```

where parameter `animationFcn` can be assigned the name of a procedure that is to perform the operations for incrementing the animation parameters. This procedure is continuously executed whenever there are no display-window events that must be processed. To disable the `glutIdleFunc`, we set its argument to the value `NULL` or the value 0.

An example animation program is given in the following code, which continuously rotates a regular hexagon in the xy plane about the z axis. The origin of

three-dimensional screen coordinates is placed at the center of the display window so that the z axis passes through this center position. In procedure `init`, we use a display list to set up the description of the regular hexagon, whose center position is originally at the screen-coordinate position (150, 150) with a radius (distance from the polygon center to any vertex) of 100 pixels. In the display function, `displayHex`, we specify an initial 0° rotation about the z axis and invoke the `glutSwapBuffers` routine. To activate the rotation, we use procedure `mouseFcn`, which continually increments the rotation angle by 3° once we press the middle mouse button. The calculation of the incremented rotation angle is performed in procedure `rotateHex`, which is called by the `glutIdleFunc` routine in procedure `mouseFcn`. We stop the rotation by pressing the right mouse button, which causes the `glutIdleFunc` to be invoked with a `NULL` argument.

```
#include <GL/glut.h>
#include <math.h>
#include <stdlib.h>

const double TWO_PI = 6.2831853;

GLsizei winWidth = 500, winHeight = 500;    // Initial display window size.
GLuint regHex;                              //  Define name for display list.
static GLfloat rotTheta = 0.0;

class scrPt {
public:
    GLint x, y;
};

static void init (void)
{
   scrPt hexVertex;
   GLdouble hexTheta;
   GLint k;

   glClearColor (1.0, 1.0, 1.0, 0.0);

   /*  Set up a display list for a red regular hexagon.
    *  Vertices for the hexagon are six equally spaced
    *  points around the circumference of a circle.
    */
   regHex = glGenLists (1);
   glNewList (regHex, GL_COMPILE);
      glColor3f (1.0, 0.0, 0.0);
      glBegin (GL_POLYGON);
         for (k = 0; k < 6; k++) {
            hexTheta = TWO_PI * k / 6;
            hexVertex.x = 150 + 100 * cos (hexTheta);
            hexVertex.y = 150 + 100 * sin (hexTheta);
            glVertex2i (hexVertex.x, hexVertex.y);
         }
      glEnd ( );
   glEndList ( );
}
```

```
void displayHex (void)
{
   glClear (GL_COLOR_BUFFER_BIT);

   glPushMatrix ( );
   glRotatef (rotTheta, 0.0, 0.0, 1.0);
   glCallList (regHex);
   glPopMatrix ( );

   glutSwapBuffers ( );

   glFlush ( );
}

void rotateHex (void)
{
   rotTheta += 3.0;
   if (rotTheta > 360.0)
      rotTheta -= 360.0;

   glutPostRedisplay ( );
}

void winReshapeFcn (GLint newWidth, GLint newHeight)
{
   glViewport (0, 0, (GLsizei) newWidth, (GLsizei) newHeight);

   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ( );
   gluOrtho2D (-320.0, 320.0, -320.0, 320.0);

   glMatrixMode (GL_MODELVIEW);
   glLoadIdentity ( );

   glClear (GL_COLOR_BUFFER_BIT);
}

void mouseFcn (GLint button, GLint action, GLint x, GLint y)
{
   switch (button) {
      case GLUT_MIDDLE_BUTTON:           //  Start the rotation.
         if (action == GLUT_DOWN)
            glutIdleFunc (rotateHex);
         break;
      case GLUT_RIGHT_BUTTON:            //  Stop the rotation.
         if (action == GLUT_DOWN)
            glutIdleFunc (NULL);
         break;
      default:
         break;
   }
}
```

```
void main (int argc, char** argv)
{
   glutInit (&argc, argv);
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
   glutInitWindowPosition (150, 150);
   glutInitWindowSize (winWidth, winHeight);
   glutCreateWindow ("Animation Example");

   init ( );
   glutDisplayFunc (displayHex);
   glutReshapeFunc (winReshapeFcn);
   glutMouseFunc (mouseFcn);

   glutMainLoop ( );
}
```

11 Summary

An animation sequence can be constructed frame by frame, or it can be generated in real time. When separate frames of an animation are constructed and stored, the frames can later be transferred to film or displayed in rapid succession on a video monitor. Animations involving complex scenes and motions are commonly produced one frame at a time, while simpler motion sequences are displayed in real time.

On a raster system, double-buffering methods can be used to facilitate motion displays. One buffer is used to refresh the screen, while a second buffer is being loaded with the screen values for the next frame of the motion. Then the roles of the two buffers are interchanged, usually at the end of a refresh cycle.

Another raster method for displaying an animation is to perform motion sequences using block transfers of pixel values. Translations are accomplished by a simple move of a rectangular block of pixel colors from one frame-buffer position to another. And rotations in 90° increments can be performed with combinations of translations and row-column interchanges within the pixel array.

Color-table methods can be used for simple raster animations by storing an image of an object at multiple locations in the frame buffer, using different color-table values. One image is stored in the foreground color, and the copies of the image at the other locations are assigned a background color. By rapidly interchanging the foreground and background color values stored in the color table, we can display the object at various screen positions.

Several developmental stages can be used to produce an animation, starting with the storyboard, object definitions, and specification of key frames. The storyboard is an outline of the action, and the key frames define the details of the object motions for selected positions in the animation. Once the key frames have been established, in-between frames are generated to construct a smooth motion from one key frame to the next. A computer animation can involve motion specifications for the "camera," as well as motion paths for the objects and characters involved in the animation.

Various techniques have been developed for simulating and emphasizing motion effects. Squash and stretch effects are standard methods for stressing