An Introduction to 3D Computer Graphics, Stereoscopic Image, and Animation in OpenGL and C/C++

Fore June

Chapter 11 Animation

11.1 Introduction

Animation is an optical illusion of motion due to the phenomenon of persistence of vision. The following figure shows a sequence of images of a ball at various heights. When the images are shown in sequence repetitively at a rate of about 20 frames (images) in one second, we shall see the ball bouncing on the ground.

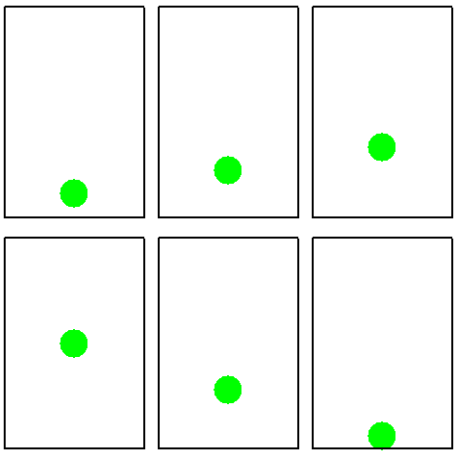

Figure 11-1 Sequence of Images Used in Animation

People had applied traditional animation techniques in traditional movies long before the use of computerized animation in applications. In early animation techniques, started in the 1880s, sequential drawings are employed to produce movement. The technique requires drawing a series of pictures, each picture showing an instantaneous snapshot of objects in motion. Figure 11-2 shows an animation picture of "Fantasmagorie" created by Emile Cohl in 1908. Production of animated short films (Cartoon) became an industry in the 1910s.

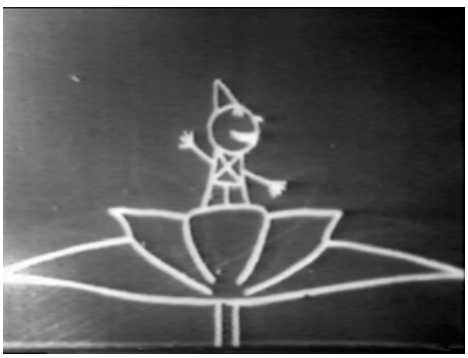

Figure 11-2 Fantasmagorie by Emile Cohl, 1908

Cel animation is a traditional animation technique, where individual frames are photographs of drawings, which are first drawn on paper. Drawings are traced or photocopied onto transparent acetate sheets called **cels**, and then photographed by a rostrum camera.

Stop motion animation is another popular technique used in traditional animation. People take photographs of real physical objects one frame at a time and play them back to create an illusion of motion. This technique consists of several kinds which are listed below:

1. **Clay animation (claymation)** uses clay to create figures. Figure 11-3 shows a claymation scene from a TV show.

Figure 11-3 A Claymation Scene from a TV Show

2. **Cutout animation** uses 2-D pieces of materials such as paper, cloth, card or even photographs to do animation. Nowadays computer software is frequently used to create cutout-style animation with scanned images or vector graphics taking the role of physically cut materials.

3. **Graphic animation** uses non-drawn flat visual graphic material such as photographs, newspaper clippings, and magazines, which are sometimes manipulated frame-by-frame, to create object movement.

4. **Model animation** uses models that interact with live-action world to create the illusion of a real-world fantasy sequence. The technique was used in the the films *The Lost World* (1925), *King Kong* (1933), and *The Son of Kong* (1933).

5. **Object animation** uses regular objects such as LEGOs, dolls, and toys to do animation.

6. **Pixilation** uses live humans to create surreal effects such as disappearances and reappearances, allowing people to appear to slide across the ground.

7. **Puppet animation (pupperty)** manipulates puppets to creation object motion.

11.2 Computer Animation

Nowadays, we use computer and in particular computer graphics to do much of the work of animation. We can consider computer animation as a digital successor to the stop motion techniques discussed above. Modern computer animation usually uses 3D computer graphics, although 2D computer graphics are still used for stylistic, low bandwidth, and

faster real-time renderings. To create the illusion of movement, we display an image on the computer screen and repeatedly replace it by a new image that is similar to the previous image, usually at a rate of 24 or 30 frames/second. This technique is identical to how the illusion of movement is achieved with television and motion pictures.

For 3D animations, we usually build graphical objects using a virtual skeleton of rigid links connected by joints. The joints of a skeleton can be given characteristics that controls their motion. The movement of the rest of the object is synchronized with that of the skeleton. One such technique is called **skinning**, which allows the "skin", or surface, of a creature to move in sync with the skeleton movement. For 2D figure animations, we may use separate objects with or without a virtual skeleton. Other features of the figure such as limbs, eyes, mouth, and clothes are moved by the animator on key frames. The differences in appearance between key frames are interpolated by the computer using the tweening or morphing technique. At the end, the animation is rendered.

In fact, several aspects of computer animation are direct extensions of traditional animation techniques. The following are some prominent features of computer animation.

1. **keyframing**. Keyframing is a simple form of animating an object by specifying the states of objects at keyframes and filling in intermediate frames by interpolation. The technique is based on the notion that an object has a beginning state or condition and will be changing over time, in position, form, color, luminosity, or any other property, to some different final stare. Keyframing takes the stance that we only need to show the "key" frames, or conditions, that describe the transformation of this object, and that all other intermediate positions can be interpolated. In traditional animations of movies, the keyframes (keys) are drawn by senior animators. Other artists, *inbetweeners* and *inkers*, draw intermediate frames. This is important for creating a sequence of frames that are coherent and for division of labour. In computerized keyframing, people make use of spline curves to interpolate points.

2. **Motion Capture**. Motion capture uses real world object movements to guide animated motion. Special sensors, called *trackers* can be used to record the motion of a human. In motion capture sessions, movements of one or more actors are sampled many times per second. These animation data are mapped to a 3D model so that the model performs the same actions as the actor.

3. **Compositing**. Compositing is the process of overlaying visual elements from separate sources to form a single image, often to create the illusion that all those elements are parts of the same scene. It makes use of the alpha channel of the images to achieve the effect. Using compositing we can combine images of real characters and objects with computer graphics to create special scenes and effects.

4. **Procedural Methods**. Procedural method is the process of generating contents algorithmically rather than manually; the computer program follows the steps of an algorithm to generate motions. They can be used to easily generate a family of similar motions. The algorithms can be based on actual physical laws to simulate real physical motions. For example, *flexible dynamics* simulates behaviors of flexible objects, such as clothes and paper. We usually build a model using triangles, with point masses at the triangles' vertices. We join the triangles at edges with hinges, which open and close in resistance to springs that hold the two hinge halves together. Parameters in the model include point masses, positions, velocities, accelerations,

spring constants, and wind force. Procedural methods are particularly useful in modeling particle systems, which model fuzzy objects such as fire, water and smoke.

5. **Computer Animation Software**. Popular computer animation software include the following:

   - 3D Studio MAX (Autodesk)
   - Softimage (Microsoft)
   - Alias/Wavefront (SGI)
   - Lightwave 3D (Newtek)
   - Prisms 3D Animation Software (Side Effects Software)
   - HOUDINI (Side Effects Software)
   - Apple's toolkit for game developers
   - Digimation
   - Blender (Open Source)
   - Maya

   In particular, Blender is a free open-source 3D content creation suite, available for all major operating systems under the GNU General Public License. It is a very powerful tool that can create sophisticated 3D graphical objects and models for animation. For details, one can refer to its official site *http://www.blender.org/*.

11.3 Computer Animation Systems

Most computer animation systems use a model representation of the image, similar to a stick figure to control the movements of the animation. Each segment of the model is controlled by an animation variable. However, these systems are only appropriate in animating rigid bodies. Today, computer animation can model systems beyond rigid bodies. These include the following kinds of systems.

1. **Rigid Bodies**. We may treat a rigid body as a system of particles, where the distance between any two particles is fixed. Therefore, its shape never changes. A rigid body may be controlled directly with positions and orientation or indirectly with velocity and angular velocity.

2. **Flexible Objects**. Flexible objects are squishy or bendable objects such as cloth, rope, and paper. A simple and intuitive technique to simulate or animate flexible objects is the mass-spring model, where differential equations are employed to calculate the force that acts on each mass-point and we can animate the mass-point by numerical integration of the force.

3. **Camera Viewpoint**. The camera is our eye. In computer animation, every scene is rendered from the camera's point of view. We need to move the camera to follow the viewer's viewpoint without having excessive jerkiness, oscillation or other jitter.

4. **Articulated Rigid Bodies** (**multibodies**). An articulated rigid body is an assembly of hierarchically linked rigid bodies; joints may be rotational or translational (e.g. robot arms, skeletons of humans). There are two categories of articulated rigid bodies, kinematics and dynamics, which can be further classified as follows:

- **Forward Kinematics** determines the positions of the links in the structure when the joint settings are known.
- **Inverse Kinematics** determines the joint settings (angles) to achieve a desired position. This is much more difficult than Forward Kinematics.
- **Forward Dynamics** is the same as physical simulation. That is, given the initial physical parameters such as positions, orientations, and velocities, the process computes the movement of the articulated object as a function of time.
- **Inverse Dynamics** determines what forces must be applied to achieve certain motion.

5. **Particle Systems**. Particle systems are for modeling fuzzy objects such as water, cloud, fire, and smoke. The objects are non-rigid, dynamic, and do not have well-defined shapes. They differ from normal image representation in 3 ways:

   - In a particle system, we do **not** represent an object by a set of patches or a polygon mesh but as clouds of primitive particles that define its volume.
   - They are not static. New particles are created and destroyed.
   - Object form may not be specific. Stochastic processes are used to change an object's shape and appearance. However, particle systems can also be used to create well-shaped objects such as a human head or body.

   Figure 11-4 below is an image created using particle system techniques. The image is taken from the film *Star Trek II: Wrath of Khan*, which was one of the first films to extensively use electronic images and computer graphics to expedite the production of shots, to show the effect of *Genesis*, which was a powerful terraforming device. The film contained a one minute computer-generated imagery (CGI) sequence depicting a simulation of the birth of a planet (the *Genesis effect*). The Genesis effect made the first onscreen use of a particle rendering system to achieve its fiery effects.

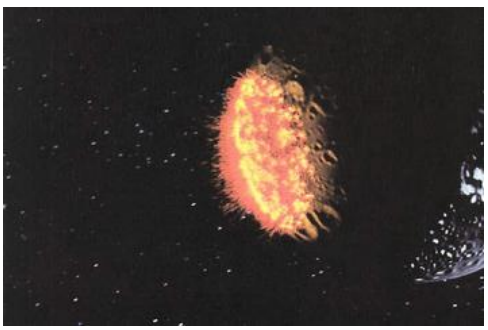

Figure 11-4 Genesis Effect of Star Trek II

11.4 Interpolation

Interpolation is the foundation of all animations. In animation, interpolation refers to in-betweening, or filling in frames between the key frames. It typically calculates the in-between frames through use of piecewise polynomial interpolation to draw images semi-automatically. The simplest case is interpolating the position of a point. An animator has

a list of values associated with a given parameter at specific frames (called key frames or keys). The parameter to be interpolated may be one of the following:

- a coordinate of the position of an object,
- a joint angle of an appendage of a robot,
- the transparency attribute of an object, or
- any other parameter used in the display and manipulation of computer graphic elements.

The simplest kind of interpolation is the linear interpolation, where the points are related by a linear equation. Suppose two polylines Q and R are formed by n points:

$$\begin{aligned} Q &= \{Q_0, Q_1, .., Q_i, ...Q_{n-1}\} \\ R &= \{R_0, R_1, .., R_i, ...R_{n-1}\} \end{aligned} \tag{11.1}$$

We construct polyline P with points $\{P_0, P_1, .., P_i, ...P_{n-1}\}$ by 'interpolating' Q and R. That is,

$$P_i(t) = (1-t)Q_i + tR_i \tag{11.2}$$

Note that (11.2) is an affine combination of points as the sum of the coefficients $(1-t)$ and t is always equal to 1, and thus the result is always a valid point. We also have the special situation:

$$\begin{aligned} &\text{When } t = 0, P_i = Q_i (i.e. P = Q) \\ &\text{When } t = 1, P_i = R_i (i.e. P = R) \end{aligned} \tag{11.3}$$

So when t changes from 0, to 0.1, 0.2, ..., to 1, P begins with the shape Q and gradually morphs to R. Figure 11-5 below shows an example of tweening; figures (a) and (d) are the key frames at $t = 0$ and $t = 1$ respectively; (b) and (c) are obtained by interpolation.

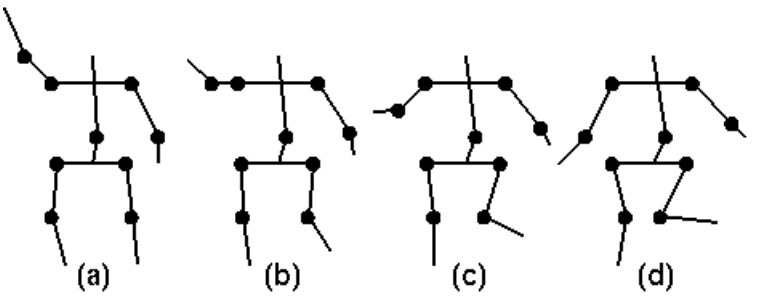

Figure 11-5 Interpolation

By rewriting (11.2) as

$$P_i(t) = Q_i + (R_i - Q_i) \times t \tag{11.4}$$

we can implement the linear interpolation with a code like the following:

```
class Point2{
public:
  float x;
  float y;
};

Point2 tween( const Point2 &Q, const Point2 &R, float t )
{
```

```
    Point2 P;
    P.x = Q.x + ( R.x - Q.x ) * t;
    P.y = Q.y + ( R.y - Q.y ) * t;
    return P;
}
```

The following listing is the complete program of the above example of tweening that generates the figures of Figure 11-5. In the program, the functions **make***() creates three figures from the key frames (i.e. at $t = 0$, and $t = 1$); the three figures together simulates the skeleton of a human body. The function **tween**() does the interpolation of points at t with ($0 \leq t \leq 1$. The function **drawTween**() calls the **tween**() function to do the interpolation and draws the figures. The value of t and the position of a figure can be changed by pressing the key 'a', which animates and moves the skeleton across the screen. Pressing the key 'r' reverses the direction of motion.

Program Listing 11-1: Animation by Interpolation (tween.cpp)

```
/*
 * tween.cpp
 * Demo of in-between principle and animation.
 * Press 'ESC' to quit the program.
*/
#include <stdlib.h>
#include <GL/gl.h>
#include <GL/glut.h>

using namespace std;

class Point2{
public:
  float x;
  float y;
  Point2()
  { x = y = 0; }
  Point2( float x0, float y0 )
  {
    x = x0; y = y0;
  }
  Point2 ( const Point2 &p )
  {
    x = p.x;
    y = p.y;
  }
};

Point2 tween( const Point2 &Q, const Point2 &R, float t )
{
  Point2 P;
  P.x = Q.x + ( R.x - Q.x ) * t;
  P.y = Q.y + ( R.y - Q.y ) * t;
  return P;
}

void drawTween( const Point2 A[], const Point2 B[], int n,
                float t, const Point2 &c )
{
  glColor3f (0, 0, 0 );
```

```
  glEnable ( GL_POINT_SMOOTH );
  glPointSize ( 8 );

  // draw the tween at time t between polylines A and B
  Point2 P0 = tween(A[0], B[0],t);
  Point2 P1 = Point2( P0.x + c.x, P0.y + c.y );
  for(int i = 1; i < n; i++)
  {
    Point2 P, P2;
    if ( i == n )
       P = tween( A[0], B[0],t );
    else
       P = tween( A[i], B[i],t );
    P2 = Point2( P.x + c.x, P.y + c.y );
    if ( i < n - 1 ) {
      glBegin( GL_POINTS );
        glVertex2f ( P2.x, P2.y );
      glEnd();
    }
    glBegin( GL_LINES );
      glVertex2f ( P1.x, P1.y );
      glVertex2f ( P2.x, P2.y );
    glEnd();
    P1 = P2;
  }
}

//create two figures for demo use
void makeHand( Point2 A[], Point2 B[] )
{
  A[0].x = 1.2; A[0].y = 9.8;  A[1].x = 2; A[1].y = 8;
  A[2].x = 3; A[2].y = 7;  A[3].x = 6; A[3].y = 7;
  A[4].x = 7; A[4].y = 5; A[5].x = 7; A[5].y = 4;

  B[0].x = 1; B[0].y = 4;  B[1].x = 2; B[1].y = 5;
  B[2].x = 3; B[2].y = 7;  B[3].x = 6; B[3].y = 7;
  B[4].x = 7.5; B[4].y = 5.5; B[5].x = 8; B[5].y = 5;
}

void makeBody( Point2 A[], Point2 B[] )
{
  A[0].x = 4.5; A[0].y = 8;  A[1].x = 4.7; A[1].y = 5;
  A[2].x = 4.5; A[2].y = 4;

  B[0].x = 4.5; B[0].y = 8;  B[1].x = 5; B[1].y = 5;
  B[2].x = 4.5; B[2].y = 4;
}

void makeLeg( Point2 A[], Point2 B[] )
{
  A[0].x = 3.5; A[0].y = 0.2;  A[1].x = 3; A[1].y = 2;
  A[2].x = 3.2; A[2].y = 4;  A[3].x = 5.8; A[3].y = 4;
  A[4].x = 6; A[4].y = 2; A[5].x = 6.2; A[5].y = 0.2;

  B[0].x = 3.2; B[0].y = 0.2;  B[1].x = 3.5; B[1].y = 2;
  B[2].x = 3; B[2].y = 4;  B[3].x = 5.8; B[3].y = 4;
  B[4].x = 4.8; B[4].y = 2; B[5].x = 7; B[5].y = 1.8;
}

Point2 A[10], B[10];
```

```
Point2 A1[10], B1[10];
Point2 A2[10], B2[10];
Point2 center( 0, 0 );
float t = 0, deltat = 0.1;
float deltax = 2, deltay = 0;

void init ( void )
{
  gluOrtho2D ( 0.0, 30.0, 0.0, 30.0 );
  makeHand( A, B ); //create figure A and B
  makeBody( A1, B1 ); //create figure A1 and B1
  makeLeg( A2, B2 ); //create figure A2 and B2

  glLineWidth ( 2 );
  glClearColor( 1.0, 1.0, 1.0, 0.0 );
}
void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT);

  drawTween( A, B, 6, t, center );
  drawTween( A1, B1, 3, t, center );
  drawTween( A2, B2, 6, t, center );

  glFlush();
  glutSwapBuffers();
}

void animate()
{
  t += deltat;
  //move center for clarity of display
  center.x += deltax; center.y += deltay;
  if ( t > 1 ){
t = 1.0;
deltat = -deltat;  //reverse direction
        deltax = -deltax;
        deltay = -deltay;
  } else if ( t < 0 ) {
t = 0;
deltat = -deltat;  //reverse direction
        deltax = -deltax;
        deltay = -deltay;
  }
  glutPostRedisplay ();
}

void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
      case 27:
        exit(0);
        break;
      case 'a':
        animate();
break;
      case 'r':
  deltat = -deltat; //reverse direction
        deltax = -deltax;
        deltay = -deltay;
```

```
animate();
break;
  }
}

int main( int argc, char *argv[] )
{
  // Set things (glut) up
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB );

   // Create the window and handlers
   glutCreateWindow("Tweening Demo");
   glutReshapeWindow( 500, 500 );
   glutInitWindowPosition(100, 100);
   glutDisplayFunc(display);
   glutKeyboardFunc(keyboard);
   init();
   //perpetual loop
   glutMainLoop();
   return(0);
}
```

Equation (11.2) is for linear interpolation and it is used by the above example. We can easily generalize the idea to do quadratic or cubic interpolation. For instance, we note that

$$1 = 1^2 = ((1-t)+t)^2 = (1-t)^2 + 2t(1-t) + t^2 \tag{11.5}$$

As the sum of $(1-t)^2$, $2t(1-t)$, and t^2 is always equal to 1, we can use them as the coefficients for combining three points so that the result is a valid point. This gives us quadratic interpolation. In this case, we construct polyline $P = \{P_0, P_1, .., P_i, ...P_{n-1}\}$ from the interpolation of three polylines, say, Q, R, and S. The equation corresponding to (11.2) for quadratic interpolation is given by

$$P_i(t) = (1-t)^2 Q_i + 2t(1-t)R_i + t^2 S_i \tag{11.6}$$

Note that when $t = 0$, $P = Q$, and when $t = 1$, $P = S$. So P starts from Q and morphs to S at the end. Similarly, we can do cubic interpolation by noting that

$$1 = 1^3 = ((1-t)+t)^3 = (1-t)^3 + 3(1-t)^2 t + 3(1-t)t^2 + t^3 \tag{11.7}$$

Now, we need four polylines, say, Q, R, S, and W to do the interpolation:

$$P_i(t) = (1-t)^3 Q_i + 3(1-t)^2 t R_i + 3(1-t)t^2 S_i + t^3 W_i \tag{11.8}$$

For more sophisticated graphics, people often use Bezier curves or B-splines to interpolate points.

11.5 Animation in OpenGL

People often use the OpenGL callback functions **glutIdleFunc**() and/or **glutVisibility-Func**() to do animation.

The function **glutIdleFunc**() sets the global idle callback with prototype,

```
void glutIdleFunc ( void (*func)() );
```

It sets the global idle callback to be *func* so that an OpenGL program can perform tasks in the background or continuous animation when window system events are not being received. The callback routine *func* does not take any parameters. We should minimize the amount of computation and rendering done in an idle callback to avoid disturbing the program's interactive response. In general, we should not process more than one frame of rendering in an idle callback. If we want to disable the generation of the idle callback, we can simply pass NULL to **glutIdleFunc**(). To use this function in the program, we can add a statement like the following in the **main**() routine of the program:

```
glutIdleFunc ( idle );
```

The following is a typical callback idle function to animate a window:

```
void idle(void)
{
    time += 0.02;
    glutSetWindow ( window );
    glutPostRedisplay();
}
```

Note in this example that the idle callback function does not do any actual drawing. It only advances the time scene state global variable. The actual drawing is done by the window's display callback function, which is called by the call to **glutPostRedisplay**().

If we use the idle callback for animation, we should make sure that we stop rendering when the window is not visible. This can be easily done by setting up with a visibility callback like the following:

```
void visible ( int vis )
{
   if ( vis == GLUT_VISIBLE )
       glutIdleFunc ( idle );
   else
       glutIdleFunc ( NULL );  //disable idle callback
}
```

Also, for animation applications, it is better **not** to set up the idle callback before calling **glutMainLoop**(), but to use the visibility callback to install idle callback when the window first becomes visible on the screen.

For simplicity, we make slight modifications to the program **tween.cpp** of Listing 11-1 to illustrate the use of **glutIdleFunc**() to animate the tweening figures. To specify the frame rate, the rate at which we want to display a new figure, we need a function that gives us the the time lapse between the execution of statements of a program. We use the a function provided by the Simple Directmedia Layer (SDL) Library (*http://www.libsdl.org/*) to accomplish this. The SDL function **SDL_GetTicks**() is used to give an estimate of the time delay between two frames as shown in the following piece of code where a frame rate of 20 frames per second (fps) is assumed:

```
static unsigned int prev_time = SDL_GetTicks();
static unsigned int current_time=SDL_GetTicks();

current_time = SDL_GetTicks();     //ms since library starts
int diff = current_time - prev_time;
if ( diff < 50 ){      //20 fps = 50 ms / frame
   int delay = 100 - ( current_time - prev_time );
   SDL_Delay ( delay );
}
```

The function **SDL_GetTicks**() returns the number of milliseconds since the SDL library has been initialized. This value wraps around if the program runs for more than 49.7 days. The SDL function **SDL_Delay**() waits a specified number of milliseconds before returning. The delay granularity is at least 10 ms. To use these functions, we have to include the header statement,

```
#include <SDL/SDL.h>
```

and links the SDL library with the linking option "-lSDL". (Alternatively, one may use the standard C function **gettimeofday**() function to obtain the current time by including the header "#include <sys/time.h>".)

Listing 11-2 presents the program that does the tweening animation. It is a modification of **tween.cpp** of Listing 11-1, which moves the skeleton when the user presses the key 'a'. Here, the skeleton moves across the screen automatically. The parts that are identical to **tween.cpp** are not listed.

Program Listing 11-2: Animation using glutIdleFunc (tween1.cpp)

```
#include <SDL/SDL.h>

void idle ()
{
   static unsigned int prev_time = SDL_GetTicks(),
      current_time = SDL_GetTicks();

   current_time = SDL_GetTicks();  //ms since library starts
   int diff = current_time - prev_time;
   if ( diff < 100 ){    //10 fps ~ 100 ms / frame
      int delay = 100 - ( current_time - prev_time );
      SDL_Delay ( delay );
   }
   prev_time = current_time;
   animate();
}

int main( int argc, char *argv[] )
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB );

   glutCreateWindow("Tweening Demo");
   glutReshapeWindow( 500, 500 );
   glutInitWindowPosition(100, 100);
   glutDisplayFunc(display);
   glutIdleFunc ( idle );    //for animation

   glutKeyboardFunc(keyboard);
   init();
   glutMainLoop();
   return 0;
}
```

Another way to do animation in OpenGL is to use the function **glutVisibilityFunc**(), which sets the visibility callback for the current window. It has the following prototype,

void glutVisibilityFunc (void (**func*)(int *state*));

where the parameter *func* is the new visibility callback function. The visibility callback for a window is called when the visibility of a window changes. The *state* callback parameter is either GLUT_NOT_VISIBLE or GLUT_VISIBLE depending on the current visibility of the window. GLUT_VISIBLE does not distinguish a window being totally versus partially visible. GLUT_NOT_VISIBLE indicates that no part of the window is visible, i.e., until the window's visibility changes, all further rendering to the window is discarded. Note that GLUT considers a window visible if any pixel of the window is visible or any pixel of any descendant window is visible on the screen. Passing NULL to **glutVisibilityFunc**() disables the generation of the visibility callback.

Very often, this function works along with the function **glutTimerFunc**() to set the frame rate of animation. The function **glutTimerFunc**() has the following prototype,

```
void glutTimerFunc ( unsigned int  msecs,
                     void (*func)(int value), value);
```

where parameters

msecs specifies the number of milliseconds to pass before calling the callback,
func specifies the timer callback function, and
value specifies the integer value to pass to the timer callback function.

This function registers the timer callback *func* to be triggered in at least *msecs* milliseconds. The *value* parameter to the timer callback will be the value of the *value* parameter to **glutTimerFunc**(). Multiple timer callbacks at same or differing times may be registered simultaneously.

We again slightly modify **tween.cpp** of Listing 11-1 above to make use of these two functions to animate the tweening figures. Listing 11-3 below presents the modified code; functions identical to those of Listing 11-1 are omitted. In this program (**tween2.cpp**), the function **timerHandle**() calls **animate**() to change the skeleton to a new form and move it to a new positon; it then calls **glutPostRedisplay**() to render the new scene. At the end, it calls **glutTimerFunc**() to call itself after 100 ms. Note that in this example, the *value* parameter is not used. The visibility callback function **visHandle**() initiates the call to **timerHandle**() when the window becomes visible.

Program Listing 11-3: Animation using glutTimerFunc (tween2.cpp)

```
void timerHandle ( int value )
{
   animate();
   glutPostRedisplay();
   glutTimerFunc ( 100, timerHandle, 0 );
}

void visHandle( int visible )
{
   if (visible == GLUT_VISIBLE)
      timerHandle ( 0 );  //start animation when visible
}

int main( int argc, char *argv[] )
{
```

```
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB );

    glutCreateWindow("Tweening Demo");
    glutReshapeWindow( 500, 500 );
    glutInitWindowPosition(100, 100);
    glutDisplayFunc(display);
    glutVisibilityFunc( visHandle );

    glutKeyboardFunc(keyboard);
    init();
    glutMainLoop();
    return(0);
}
```

Other books by the same author

Windows Fan, Linux Fan

by *Fore June*

Windws Fan, Linux Fan describes a true story about a spiritual battle between a Linux fan and a Windows fan. You can learn from the successful fan to become a successful Internet Service Provider (ISP) and create your own wealth. See *http://www.forejune.com/*

Second Edition, 2002.
ISBN: 0-595-26355-0 Price: $6.86

An Introduction to Digital Video Data Compression in Java

by *Fore June*

The book describes the the principles of digital video data compression techniques and its implementations in java. Topics covered include RBG-YCbCr conversion, macroblocks, DCT and IDCT, integer arithmetic, quantization, reorder, run-level encoding, entropy encoding, motion estimation, motion compensation and hybrid coding. See

http://www.forejune.com/

January 2011

ISBN-10: 1456570870

ISBN-13: 978-1456570873

An Introduction to Video Compression in C/C++

by *Fore June*

The book describes the the principles of digital video data compression techniques and its implementations in C/C++. Topics covered include RBG-YCbCr conversion, macroblocks, DCT and IDCT, integer arithmetic, quantization, reorder, run-level encoding, entropy encoding, motion estimation, motion compensation and hybrid coding.

January 2010
ISBN: 9781451522273