

COMP3201 – Computer Graphics

3. Module 3: Realism and Performance

3.1 Animation, Double Buffering and Quadrics

3.1.1 Animation

Animation is the process of presenting a sequence of still images to the eye. The differences between images must be small so that the brain decides that the motion is continuous. Each of these still images is usually called a frame.

There are two ways to draw the next frame. The first is to keep track of the changes between one frame and the next. We then "simply" alter the original frame to update it to the next. This can work well for a simple animation of a small object moving around a static background. In general though, this method proves to be too complex. A simpler method is to wipe the entire buffer and redraw the new scene from scratch.

The GLUT framework is a typical event handling system. Thus our display callback function is called whenever the windowing system needs to redraw the screen (e.g. a window on top of ours was moved away, or our window was resized, etc). For animation however we need to redraw the window much more frequently than these events would normally occur.

As you may expect from an event handling framework, there are two ways of achieving more callbacks from the GLUT framework. The first is by using idle time callbacks, the second is by using timer callbacks. Inside these extra events you can generate a display callback request by calling `glutPostRedisplay()`. Note that you should avoid calling the display function explicitly and simply post a request for the display function to be called.

The difference between idle time and timers is as follows. An application is defined to be idle when there are no other messages in the event queue. An idle time message is then posted to the event queue by the operating system. The application is then able to choose to ignore the idle time message or to take advantage of this and do some processing that would otherwise not be done. Idle time processing can lead to CPU thrashing as whenever the application would otherwise be doing nothing, it is now working.

Timers are messages posted by the operating system to the application at times predefined by the application. That is, an application can ask the operating system to send it a timer message at a given time. It should be noted that the operating system will send the timer message as soon after the expiration of the waiting period as possible. If the application can process the timer message before any other messages appear in the event queue (and assuming that there is no idle time processing) then the application can sleep until the next message (possibly a timer message) arrives. This method means that the application only uses resources as necessary, as opposed to idle time processing which tends to consume available resources.

An idle time callback function can be registered with GLUT by calling `glutIdleFunc`. For example,

```
void idle( void )
{
    /* do some stuff */
    /* if you now want to generate a call to the display callback */
    glutPostRedisplay();
}

int main( int argc, char* argv[])
{
    /* other init code */
    glutIdleFunc( idle );
    /* more init code */
    glutMainLoop();
    return 0;
}
```

A timer callback is registered with GLUT by calling `glutTimerFunc` with both the time in milliseconds to pass before the callback, the callback function, and a value to pass to the callback function. (Note that the time until callback is only guaranteed to be greater than that of the requested callback time. However best efforts are made to be close to the callback time.) For example,

```
void timer( int value )
{
    /* do some processing, could be conditional on the value passed in */
    /* if you want to generate a call to the display callback */
    glutPostRedisplay();
    /* now reregister ourselves otherwise we are never called again */
    glutTimerFunc( 1000, timer, 23 );
}

int main( int argc, char* argv[])
{
    /* other init code */
    glutTimerFunc( 2300, timer, 17 );
    /* more init code */
    glutMainLoop();
    return 0;
}
```

It is worthwhile noting that a timer callback is only called once. If the timer callback function needs to be called again, it must be reregistered with GLUT.

As a complete animation example, consider a rectangle in the x-y plane which spins around the z-axis. The generation of the extra display callbacks will be done in idle time.

```

/* anim1.c */

#include <GL/glut.h> /* includes gl.h */

/* Global variables */
GLfloat angle = 0.0;

/* the idle callback function */
void idle(void)
{
    static GLint lastTime=0;
    GLint time = glutGet( GLUT_ELAPSED_TIME );

    /* increment the angle */
    angle += 20.0 * (time -lastTime)/1000; /* 20 degrees per second */
    lastTime = time;

    glutPostRedisplay();
}

GLvoid display( GLvoid )
{
    glClear( GL_COLOR_BUFFER_BIT );
    glLoadIdentity();
    glRotatef( angle, 0.0, 0.0, 1.0 );
    glRectf( 0.0, 0.0, 1.0, 1.0 );
    glFlush();
}

int main( int argc, char *argv[] )
{
    glutInit( &argc, argv );
    glutCreateWindow( argv[0] );

    glutDisplayFunc( display );
    glutIdleFunc( idle ); /* register the idle callback */

    glutMainLoop();
    return 0;
}

```

NOTE: We don't just increment the current modelview matrix with `glRotate(increment, 0, 0, 1)` but each time we clear the transformation and rotate to the desired total angle. This is to stop round off errors from building up over time.

3.1.2 Double Buffering

The above complete example actually looks terrible when executing. The reason for this is that the system is now spending almost all its time doing the display. The first task of the display routine is to clear the colour buffer so that our screen is now blank. The rotated rectangle is then drawn. The process then repeats. This means that about half of the time we are actually looking at a blank screen.

The solution to this is to use two buffers. We draw into the first buffer, while the second is being displayed, then we swap the buffers so that the first buffer is now displayed and we draw into the second. And then swap again. This process means that no half-completed image is ever shown to the viewer.

It is easy to implement this procedure in OpenGL. Firstly the display mode must be set to have the double buffers. This is achieved by a bitwise or of GLUT_DOUBLE with any other modes.

```
glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGBA | ... );
```

Next we must set the drawing to go to the back buffer, since the front buffer is the one that is displayed on the screen. (Note that this is actually the default behaviour in double buffer mode and hence it is actually unnecessary to explicitly make the call.)

```
glDrawBuffer( GL_BACK );
```

The last thing that we must do is swap the buffers when we are finished drawing a frame.

```
glutSwapBuffers();
```

Note that glutSwapBuffers has an implicit glFlush() and thus there is no need to explicitly call glFlush.

The following is the previous program with double buffering enabled. This program now has a much smoother animation.

```
/* anim2.c */

#include <GL/glut.h> /* includes gl.h */

/* Global variables */
GLfloat angle = 0.0;

/* the idle callback function */
void idle(void)
{
    static GLint lastTime=0;
    GLint time = glutGet( GLUT_ELAPSED_TIME );

    /* increment the angle */
    angle += 20.0 * (time - lastTime)/1000; /* 20 degrees per second */
}
```

```

        lastTime = time;

        glutPostRedisplay();
    }

GLvoid display( GLvoid )
{
    glClear( GL_COLOR_BUFFER_BIT );
    glLoadIdentity();
    glRotatef( angle, 0.0, 0.0, 1.0 );
    glRectf( 0.0, 0.0, 1.0, 1.0 );
    glutSwapBuffers(); /* switch the pointers to front and back buffers */
    /* glutSwapBuffers does the glFlush for you */
}

int main( int argc, char *argv[] )
{
    glutInit( &argc, argv );
    glutInitDisplayMode( GLUT_DOUBLE ); /* use a double buffer */
    glutCreateWindow( argv[0] );
    glutDisplayFunc( display );
    glutIdleFunc( idle ); /* register the idle callback */
    glDrawBuffer( GL_BACK ); /* draw to the back buffer */
    glutMainLoop();
    return 0;
}

```

3.1.3 Window Visibility

When working with a windowing environment the window that the program is using will sometimes be visible and other times not be visible. For example if a window is minimised or completely obscured by another window, then the window is not visible. GLUT defines a window to be visible when as little as a single pixel of the window (or any descendant window) can be seen on the screen.

Why is this important? If the application window is currently not visible, then the user must not currently be interested in the output of the program. Thus there is no need to be rendering animations in idle time or timer callbacks and as a consequence using up processor time that could otherwise be used by the application that the user is currently interested in.

This raises the question, how can we know whether or not the application window is visible or not? GLUT has a mechanism to provide this information. It is possible to register with the GLUT framework a function to be called back whenever the visibility state changes. This callback can then take care of stopping and starting any animation.

For example, we can define a visibility callback function by

```

void visibility( int state )
{
    /* start/stop the animation based on the information in state */
    /* one possible implementation (for idle time animation) would be */
    if( GLUT_NOT_VISIBLE == state )
    {
        glutIdleFunc( NULL );
    }
    else
    {
        glutIdleFunc( idle );
    }
}

```

The state parameter will be either GLUT_NOT_VISIBLE or GLUT_VISIBLE. The callback is registered with GLUT by calling

```

int main( int argc, char* argv[] )
{
    /* usual stuff */
    glutVisibilityFunc( visibility );

    /* more stuff */
    glutMainLoop();

    return 0;
}

```

3.1.4 Quadrics

The GL library of OpenGL only provides support for the modelling of points, lines and arbitrary polygons. While this is adequate for all drawing purposes, it would be nice to be able to easily draw standard geometric objects like spheres, cylinders, etc. The GLU library provides a facility to draw an object that is defined by the quadric equation

$$a_1x^2 + a_2y^2 + a_3z^2 + a_4xy + a_5yx + a_6xz + a_7x + a_8y + a_9z + a_{10} = 0$$

i.e., spheres, cylinders, etc.

To generate one of these geometric objects the following steps are undertaken:

- Generate a quadrics object that will in turn generate the geometric object

```
GLUquadricObj* pQuadric = gluNewQuadric();
```

- Specify the rendering attributes that we want the geometric objects to have (unless the defaults are fine)

```
gluQuadricDrawStyle( pQuadric, drawStyle );
```

where drawStyle is either GLU_POINT, GLU_LINE, GLU_SILHOUETTE, or GLU_FILL (default). GLU_POINT simply draws the vertices. GLU_LINE draws the edges between the vertices. GLU_SILHOUETTE is similar to GLU_LINE but edges between coplanar faces are not drawn. (Consider for example drawing a disk with gluDisk.)

gluQuadricOrientation(pQuadric, orientation);

where orientation is either GLU_OUTSIDE (default) or GLU_INSIDE. This controls which way the normals face. These directions are obvious for spheres and cylinders. For disks however, the positive z direction is defined to be the outside.

gluQuadricNormals(pQuadric, normals);

where normals is either GLU_NONE (default), GLU_FLAT, or GLU_SMOOTH. GLU_FLAT generates a normal for each face and applies that normal at the vertices, GLU_SMOOTH generates a normal for every vertex. If the scene is being lit and the shading model is GL_SMOOTH then GLU_SMOOTH gives the most realistic effect.

gluQuadricTexture(pQuadric, textureCoords);

where textureCoords is either GL_FALSE (default) or GL_TRUE. This function controls whether or not texture coordinates are generated for every vertex.

- Register an error handling callback for quadric rendering errors

```
void quadricErrorHandler( void ) { /* do some error handling */ }  
gluQuadricCallback( pQuadric, GLU_ERROR, quadricErrorHandler );
```

- Now render the required geometric object. e.g.

gluSphere(pQuadric, radius, slices, stacks);

where slices are the number of subdivisions *around* the z-axis to make and stacks are the number of subdivisions *along* the z-axis to make.

gluCylinder(pQuadric, baseRadius, topRadius, height, slices, stacks);

gluDisk(pQuadric, innerRadius, outerRadius, slices, rings);

**gluPartialDisk(pQuadric, innerRadius, outerRadius, slices, rings,
startAngle, sweepAngle);**

- Destroy the quadric creation object when you no longer need to create quadric geometric objects.

gluDeleteQuadric(pQuadric);

The following is a complete program using the quadrics facility to draw a tapering cylinder.

```

/* quadrics.c */

#include <GL/glut.h> /* includes gl.h */
#include <stdio.h>

/* Global variables */
GLUquadricObj* pQuadric;

void checkError( char *label )
{
    GLenum error;
    while ( (error = glGetError()) != GL_NO_ERROR )
    {
        printf( "%s: %s\n", label, gluErrorString(error) );
    }
}

/* Note that the CALLBACK is necessary to work under MS Windows */
/* To make this work on unix as well as MS Windows do the following
   #ifndef WIN32
   #define CALLBACK
   #endif
*/
GLvoid CALLBACK checkError2(GLvoid)
{
    checkError("Quadric Error");
}

GLvoid display( GLvoid )
{
    glClear( GL_COLOR_BUFFER_BIT );
    glLoadIdentity();

    /* Draw a tapering cylinder with
       baseRadius = 0.5,
       topRadius = 0.25,
       height = 0.4,
       slices = 12,
       stacks = 6
    */
    glRotatef( 45, 1.0, 1.0, 0.0 );
    gluCylinder( pQuadric, 0.5, 0.25, 0.4, 12, 6 );

    /* Don't check errors every frame - expensive. Only do it
       while debugging. Note that we don't explicitly check for
       quadric errors as the quadric error checking callback will
       be called whenever there is a quadric error.
    */
    checkError( "display" );
    glFlush();
}

```

```
}  
  
int main( int argc, char *argv[] )  
{  
    glutInit( &argc, argv );  
    glutCreateWindow( argv[0] );  
    glutDisplayFunc( display );  
  
    /* Allocate a quadric creation object */  
    pQuadric = gluNewQuadric();  
  
    /* Set up the quadric error callback function */  
    gluQuadricCallback(pQuadric, GLU_ERROR, checkError2);  
  
    glutMainLoop();  
    /* After finishing with a quadric creation object we should delete  
       it, but the execution never gets to here :) */  
    gluDeleteQuadric( pQuadric );  
    return 0;  
}
```