COMPUTER GRAPHICS

********* 2013 FALL

A Simple Integrated House

in OpenGL

****

Abstract

This paper is composed for “Computer Graphics” course in NYIT Nanjing 2013 Fall. It covers basic theories and programming implementation in OpenGL. A simple but integrated 3D house is created and its procedures have been explicitly explained. Almost all the fundamental OpenGL tools and tricks for modeling and displaying are covered in this paper. It also can been seen as a tutorial for green-hands to set up their first OpenGL simple house program.

Key Words – OpenGL, 3D house.

INTRODUCTION

Since Computer Graphics are applied in more and more fields in today's society, computer technicals for image processing and graphic modeling become mature. The special 3D animation effects in movies and games is an outstanding represent. Most importantly, the rich libraries and good behaviors of OpenGL ensure its dominance in 3D graphics.

After long-time development, OpenGL has robust libraries supporting regular computer graphics operations. For instance, it can display with depth test in a statement. This indicates the good encapsulation of OpenGL. The packages we will include in this project are “glut”, “GL” and “GLU”.

The purpose of this paper is to present the basic concepts and theories in OpenGL. Additionally, implement these abstract ideas by writing a runnable program of a simple integrated house using OpenGL common libraries.

OPENGL INSTALLATION

OpenGL is based on the C/C++. Due to the wide use of those two development environment, no more details will be introduced. The OpenGL installation will be slightly different on different operating systems. In this report, we only consider Linux mainstream distributions.

According to the package management in Linux, only a simple command is required.

$ apt-get install mesa-common-dev libgl1-mesa-dev libglu1-mesa-dev freeglut3-dev libglut-dev

Here, we have finished install the libraries required for OpenGL and simple programs can be run.

OPENGL BASIC PROGRAM STRUCTURE

1. Include header files
   Just like all the other C/C++ files, relative header files need to be imported first. In this project, “gl.h”, “glu.h” and “glut.h” are required.
2. Draw graphics
   Graphics compose of models. Various graphics should be drawn in well-encapsulated methods like “drawBox()” or “drawRoof()”.
3. Initialize all the parameters.
   A method called “init()” is often defined for initialization. Turning on the light, enabling the depth test and setting the point of view are necessary for all the programs to see the objects created before.
4. Display models
   A display function should be created to draw scenes we will see. So clear the drawing buffer first and then draw the objects we want.
5. glutMainLoop()
   Everything goes into the loop after the function is called. It will keep drawing the scene set before and stop until some events happen.

Above is the structure for a OpenGL program and also the common order for the execution. However, after going into the main loop, some call-back functions can be trigged.

A basic call-back function “void glutReshapeFunc(void (* f)(int width, int height))” is called when the window size is changed. Everything can be reset using this function. Another frequently-used function is “glutIdleFunc(void (* f)(void)”. If it is empty in the event queue, the function can be able to execute. Moreover, “void glutPostRedisplay()” is the call-back function for window redisplaying. This will be trigged when the model is transformed or the view point is changed.

3D GRAPHICS

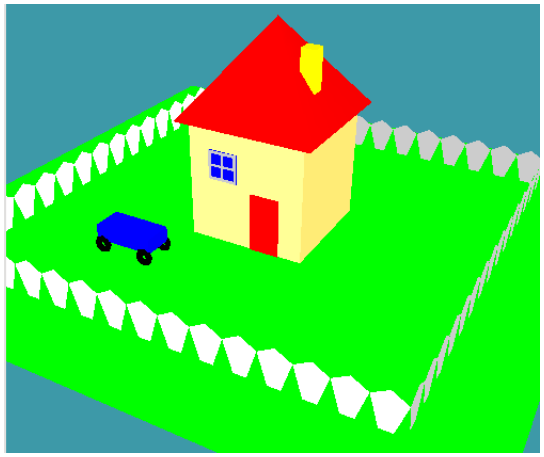
There are plenty of 3D graphics can be used directly from OpenGL libraries. *glutSolidCube(float length)* generates a cube with 'length' for each side. Similarly, solid toruses and cones can be easily set and created.

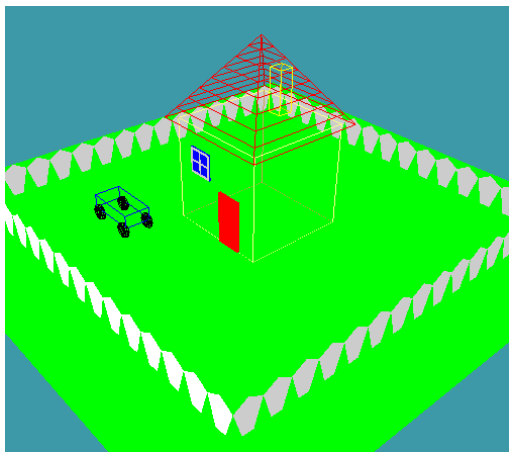
Other than the solid type of objects, there is also a wire type. Objects will be shown with lines and grids. Under this type, the inner part of objects can be seen very clearly, just like the human skulton under the X-ray.

On top of these existing graphics, polygons play an important role in model design. To make a polygon, we only need to specify its vertexes in three-dimension coordinates. And “glBegin(GL_POLYGON); ... glEnd(); ” structure is used to range the statements for the polygon. For example, to make a square on the x-y plat, all we need to do is:

```
glBegin(GL_POLYGON);
glVertex3f(-0.25,-0.25,0);
glVertex3f(-0.25,0.25,0);
glVertex3f(-0.25,0.25,0);
glVertex3f(-0.25,-0.25,0);
glEnd();
```

These graphics can be organized as groups. OpenGL uses *glPopMatrix()* and *glPushMatrix()* to group graphics. More processing operations can be executed to the all graphics in a group rather than to each graphic separately.





LIGHTING

There are many kinds of light in daily life, like spot light and environment light. Because there are no absolute smooth things in real world so almost everything can be seen by human eyes with the light they reflect.

OpenGL simulate the whole process of lighting in real world. But not only the light but also the material of objects need to be considered.

Basically, four factors decide the lightness----diffuse reflection, specular reflection, ambient reflection and emission light. OpenGL divide light to RGB model, which means, the light on any point of objects is turned into three values. With this idea, different kinds of light can be added together by respectively calculating the sum of the three value.

Last but not least, the object material is of great significance. Because absolute smooth can be set in the programs----when the diffusion coefficient equals zero----any light projecting on the object will be completely reflected backwards. Then we will be unable to see it. To prevent from this problem, remember to add the following statement:

glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);

with *diffuse* array is declared first.

MODEL TRANSFORMATION

Model transformation is the way to transform graphics in order to produce the scene you expect. The root theory of the model transformation is the linear algebra which use matrices to calculate and get the new model matrices. This approach also determines the order of program statements.

The changes of model actually happen according to the relative relations between view point and models. When we set the point of view more closed to the object. The object will seem bigger. If we change all the sides of the object in some scale, it will also seem bigger. Therefore, OpenGL provides two different ways to realize this.

Due to the usual custom, we change models more frequently than the point of view. Translation, rotation and scaling are the three main transformation operations in OpenGL.

Translation needs direction vector to decide which direction the object should be tranlated. The method "*void glTranslate<fd>(type x, type y, type z)*" defines a vector (x,y,z) as the translation direction vector.

Rotation needs an axis that the object can rotate around. The function "*void glRotate<fd>(type angle, type x, type y, type z)*" gives the object information about rotating around direction vector (x,y,z) at some angle.

It is similar for scaling. Scaling needs coefficient about each coordinate respectively on x,y,z three axis. The function "*void glScale<fd>(type sx, type sy, type sz)*" is used to scale x-coordinate, y-coordinate and z-coordinate separately.

Due to the three effective tools we have now. We can combine them as we want. Besides, there are two things to mention. First, as we state above, the transformation is realized by matrix calculation. According to the nature of matrix calculation, all matrices will be calculated in a reverse order from the statements. Second, all transformation can work on a group of graphics instead of single one. This brings great convenience to model design. More relative details can be referred to in the part "3D GRAPHICS".

The objects in my project, like a house or a car, are built up by using model transformation. Here is a car example.

```
glPushMatrix();                    // body
```

```
glScalef(2,.5,1);
glutSolidCube(.5);
glPopMatrix();
glTranslatef(0,0,.25);
glPushMatrix();
glTranslatef(-.4,-.2,0);
glutSolidTorus(.05,.1,8,8);      // wheel
glTranslatef(.8,0,0);
glutSolidTorus(.05,.1,8,8);      // wheel
glPopMatrix();
glPopMatrix();
glPopMatrix();
```

We can see from this part of code that *glPopMatrix()* combines well with transformation functions and this can create a cute car.

COLOR

Color is an significant part in visualization. There are two kinds of model in OpenGL. One is RGB and the other is RGBA. Here the 'A' means alpha, the transparency value.

In OpenGL, the default background color is black while the default drawing color is white. And we can set the color using “void glColor3(int r, int g, int b)” with value from 0 to 255. The opposite method is “glClearColor(GLclampf r, GLclampf g, GLclampf b, Glclampf a)” which can cleat the color, but only in RGBA model.

The way to color an object is definitely easy. Keep the matrix calculation order in mind.

```
GlColor3(1,1,0); // yellow
glutSolidCube(.5);
```

Here we just create a yellow cube.

By the way, lighting and material should be set properly to show the color. (refer to “LIGHTING”)

INTERACTIVE PROGRAMMING

Interactive programming relates to the user events and call-back functions (refer OPENGL BASIC PROGRAM STRUCTURE). The objects trigging an event can classify the interactive programming into two kinds, keyboard and mouse response.

In keyboard responding, we have two kinds of key, common-used keys and special keys. Sepcial keys include all the functional keys like direction keys and function keys. So the common-used keys are the other keys on the keyboard. There are two key-register-functions corresponding to each of them. "*glutKeyboardFunc(void (* f) functionName)*" is for common-used keys and "*glutSpecialFunc(void (* f) functionName)*" is for special keys. Both two only need to be declared in the "*int main()*" and they will be automatically invoked when events happen.

In my program, the key responding is for model transformation. Here is the table.

Key Name	Function	Key Name	Function
q	Rotate around x axis in clockwise	w	Rotate around x axis in counter-clockwise
a	Rotate around y axis in clockwise	s	Rotate around x axis in counter-clockwise
z	Rotate around z axis in clockwise	x	Rotate around x axis in counter-clockwise
o	Translate along x axis in positive direction	P	Translate along x axis in negative direction
k	Translate along y axis in positive direction	l	Translate along y axis in negative direction
n	Translate along z axis in positive direction	m	Translate along z axis in negative direction
UP	Scale up	DOWN	Scale down
ESC	Exit		

In mouse responding, we use "*void glutMouseFunc(void (*f)(int button, int state, int x, int y)*" to register mouse events. The call-back function will return that whether it is a left-click or right-click or middle-click, whether it is in state GLUT_UP or

GLUT_DOWN, and the exact pixel position of the cursor.

In the program, several mouse events can be responded.

First, when left-click on the mouse, the idle function will be switch between “spin” and “still”, the “AnimateScene” and “NULL” for real function names.

```
void MouseFunction(int button, int state, int x, int y)
{
   if(state == GLUT_DOWN && button == GLUT_LEFT_BUTTON)
   {
      if(mark)
         glutIdleFunc(NULL);
      else
         glutIdleFunc(AnimateScene);
      mark = !mark;
   }
}
```

Second, a right-click menu is added into the mouse responding. The menu includes six items.

Item Name	Effect
Spin	The house will rotate around y axis.
Still	The house will be shown in still.
Wire	Show the house with wires.
Solid	Show the house composing of solid graphics.
Textured	Show the textured house.
Exit	Exit the program.

All the functions to call when some item is clicked are defined in the method “*void mymenu(int value)*”. The corresponding value is set in the “*glutAddMenuEntry(String itemName, int value)*”. Until now, an integer can be mapped to some functions. Then “*glutCreateMenu(mymenu)*” is used to create a menu in the program. At last, only connection between the menu and right button of the mouse is needed. So we invoke “*glutAttachMenu(GLUT_RIGHT_BUTTON)*”.

TEXTURE MAPPING

There are mainly two kinds of texture mapping, texture maps and environment maps.

Texture maps can happen in 1D, 2D or 3D due to the dimension of texture itself. In 2D texture, with a given picture of texture, we need to bind it with a “texture object” in OpenGL first. The texture may have various properties----different color sets, ways of dealing with inappropriate size----which vary the final effect of texture mapping. Then the point-to-point relations need to be clarified to fix a point in the texture with one on the graphic models. Finally, the texture will be shown on the model like putting up a poster on the wall.

Environment maps is a brilliant implement of our real life. A mental ball always can reflect the scenes around it. And the scene will be stretched in some way on the ball, a curved rather than a flat surface. The complicated implement methods are well encapsulated in “*glTexGeni()*”.

In my program, only the first one is used. The steps can be told as follow:
1. Read the texture picture, usually a bmp or raw file, into an array.
2. Bind the texture object with the array.
3. Set the texture model
4. Fix the texture coordinates with the graphic model coordinates.

```
void setTexture(std::string fileName, int width, int height)
{
   // Step 1
   fd = fopen(fileName.c_str(), "r");
   for(i=0; i<TEX_WIDTH; i++) {
      for(j=0; j<TEX_HEIGHT; j++) {
         for(k=0; k<3; k++) {
            fread(&ch, 1, 1, fd);
            texImage[i][j][k] = (GLubyte) ch;
         }
      }
   }
   fclose(fd);

   // Step 2
   glEnable(GL_TEXTURE_2D);
   glGenTextures( 1, texName);
   glBindTexture( GL_TEXTURE_2D, texName[0] );
   glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
```

```
    // Step 3
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,GL_CLAMP);
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,GL_REPEAT );
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST)
glTexImage2D(GL_TEXTURE_2D,0,GL_RGB8,TEX_WIDTH,TEX_HEIGHT,0,
GL_RGB,GL_UNSIGNED_BYTE,texImage);
}

void drawSomeGraphic()
{
    // Step 4
    glEnable(GL_TEXTURE_2D);
    glBegin(GL_POLYGON);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(0.5,0.5,0.51);
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-0.5,0.5,0.51);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-0.5,-0.5,0.51);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(0.5,-0.5,0.51);
    glEnd();
}
```

Since we only use the three basic OpenGL packages in this program, the texture pictures read in have lost some color information which result in color distortion. OpenCV and “glaux.h” are recommended for image preprocessing.