Developing Simple Games with OpenGL

Research · October 2015

DOI: 10.13140/RG.2.1.1685.8324

1 author:

Osama Hosam
Taibah University
34 PUBLICATIONS **69** CITATIONS

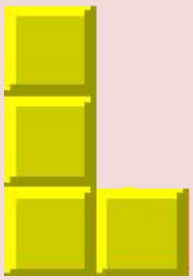
SEE PROFILE

Developing Simple Games with OpenGL

Osama Hosam Eldeen

Assistant Professor, The collage of Computer Science and Engineering in Yanbu, Taibah University, Saudi Arabia.
As with SRTA-City, Alexandria, Egypt.

October 2015



Developing Simple Games with OpenGL

Osama Hosam Eldeen

Assistant Professor, The collage of Computer Science and Engineering in Yanbu, Taibah University, Saudi Arabia; As with SRTA-City, Alexandria, Egypt

October 2015

Developing Simple Games with OpenGL

ISBN(XXXXXXXXXXXXX)

Printed in Saudi Arabia.

To my parents, my wife Mona and children
Mohammed and Abdulrahman

Osama Hosam Eldeen

Table of Contents

Preface

The book is a collection of introductory tutorials for game programming in OpenGL**.** They are collected when I was teaching game development course in Nanjing University of Information Science and Technology, China in 2008. I started to think about introducing the book when I was teaching computer graphics course in Taibah University, Saudi Arabia, Fall 2013. I was keen to let students practice how to use the basic concepts of computer graphics to make simple games. I introduced this book with accompany CD with all source codes used in the book. The source code with some tutorials on the book can be found on www.codersource.net with my name as the author.

Who can read this book?

The book is graduate level for university students who study Computer Graphics and Game development. The book contains the basics of computer graphics and OpenGL for game design and development. It is made self explanatory for those who wish to continue developing more sophisticated games. When reading specific chapters of this book, you are required to start form the first chapter till the chapter you aim to. Jumping to any chapter without reading previous chapters will result in ambiguity and difficulty.

Prerequisites

You are expected to write non-trivial software programs using C++ with object oriented programming concepts. Ability to wirte C++ code with structures and pointers is required. You also need to be able to learn programming library such as GLUT. Finally you need to be familiar with simple matrix algebra and calculus.

The Attached CD

The book has attached CD with all OpenGL codes used inside this book. The code need to be copied to your PC to be able to run them and update them. Visual C++ 6 must be installed on your PC. In addition, OpenGL library must be linked to Visual C++ 6.

The attached CD can be found on the following link

Book CD Link

How to run the attached codes?

To run the attached programs you need to follow the following steps

Step 1- Install Visual Studio 6 or higher version on your machine

Step 2- Add the OpenGL library to the installation folder of Visual Studio 6 this will be done in the following steps

- Download GLUT library from the following link http://www.xmission.com/~nate/glut.html or Google for "Download GLUT Library"
- You will find 3 files glut32.lib, glut32.dll and glut.h files, Copy the glut32.dll file and put it under the C:\Windows\System32 path
- About the glut32.lib you go to the installation folder of the Visual studio and put a copy of glut32.lib the path of libraries folder C:\Program Files\Microsoft Visual Studio\VC98\Lib.
- Copy the glut.h file and send it to the header files in the installation path for Visual Studio C:\Program Files\Microsoft Visual Studio\VC98\Include\GL.

Step 3- Open the attached program in Visual C++, and add the following path to linked libraries as Follow:

Click Project -> settings add the following line in the link tab

OpenGL32.lib GLu32.lib

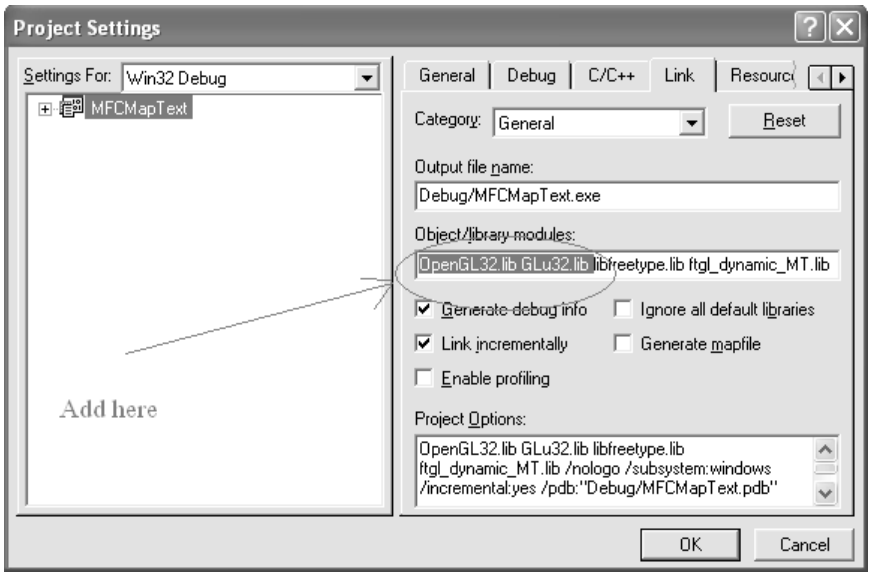


Fig.i.1 Project settings screen, linking OpenGL and Glu libraries

You can skip *Step 3* by adding the following code segment after the "include" part of the code

```
#pragma comment (lib, "opengl32.lib") /* link with Microsoft OpenGL lib */
#pragma comment (lib, "glu32.lib") /* link with OpenGL Utility lib */
```

```
#pragma comment (lib, "glut32.lib") /* link with Win32 GLUT lib */
```

Then you will be able to run OpenGL programs from within Visual C++ 6.

Chapter 1

Drawing Simple Shapes

1.1 Introduction
1.2 The required background and tools
1.3 Creating your first OpenGL project
1.4 OpenGL program structure
1.5 Drawing simple shapes
1.6 Conclusion

1.1 Introduction

In this chapter we introduce the basic building blocks for making your first OpenGL program. We are going to use GLUT library for drawing objects. GLUT library is more advanced than classic GLU, it has much functionality that saved time and made programming in OpenGL more interesting and easy. In this lesson we will introduce how to develop new OpenGL project form scratch. Also we are going to show you how to draw a simple line and triangle.

1.2 The required background and tools

To work smoothly with our lessons, you should learn the basics of C++, as OpenGL library is implemented by using C++ and we are going to use the Visual C++ environment. To be able to run our examples, on your machine you should install Visual C++ and Install GLUT and OpenGL libraries.

1.3 Creating your first OpenGL project

Open a new project in Visual C++ 6 and select the "Win32 Console Application", then select a name and a path for your project. As shown in Fig. 1.1

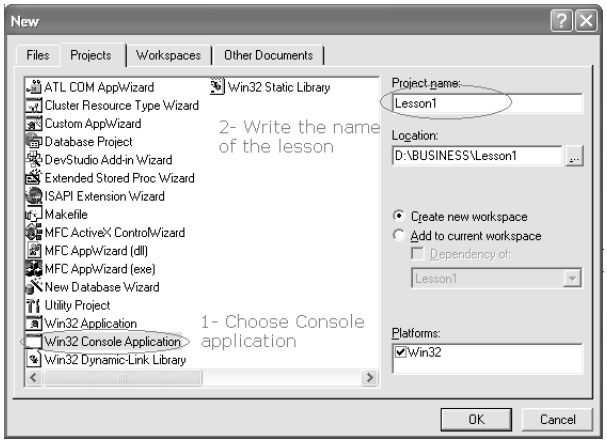


Fig. 1.1 Creating new OpenGL project

Then, select the "A simple application" to select a simple console application, as shown in Fig. 1.2

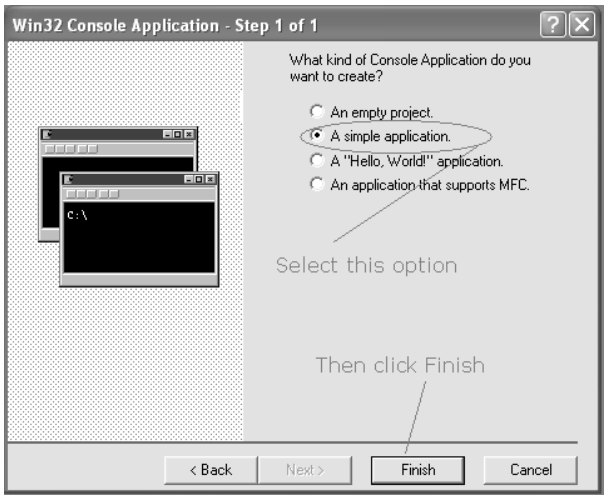


Fig. 1.2 selecting the type of the project.

The created project will contain only the following code segment

```
#include "stdafx.h"
int main(int argc, char* argv[])
{
   return 0;
}
```

We need to include the GLUT library and by using code, and link to OpenGL, Glu, and Glut libraries as shown in the following code

```
#include "stdafx.h"
#include<GL/glut.h>
#pragma comment (lib, "opengl32.lib") /* link with Microsoft OpenGL lib */
#pragma comment (lib, "glu32.lib") /* link with OpenGL Utility lib */
#pragma comment (lib, "glut32.lib") /* link with Win32 GLUT lib */
```

So far, we have done all setup needed to write our OpenGL program.

1.4 OpenGL program structure

Now, we need to know the basic shape of typical OpenGL program, any OpenGL program should have the following structure

Main function: this function contains instructions for creating the OpenGL window and looping for ever. The loop can be controlled by a timer function; we will show how to implement that in the next lessons. This can be shown by the following code segment

Initialization function: this function will initialize the environment of OpenGL to be ready for drawing shapes, this function will be called only once in the beginning of running the program.
Resize function: will be used to reset the OpenGL window to the desired parameters. This function will be activated whenever you resize your window.
Display function: This function is responsible for rendering the OpenGL scene; this function will be repeated endlessly as long as the program is running.

```
int main(int argc, char* argv[])
{
   //The body of the program
    return 0;
}
void init(void)
{
   //initialization code will be written here
}
void resize (int width, int height)
{
   //resizing window code will be written here
}
void display(void)
{
   //rendering the scene
}
```

1.5 Drawing simple shapes

Our work in this lesson will be focused on 2D applications, in the next lessons we will show how to draw objects in 3D environments. We will show what will be included in the above explained functions.

Main: The setup for creating and specifying OpenGL window characteristics is done in the main function. Here is the code of creating new and setting up new OpenGL window

```
int main(int argc, char* argv[])
{
   glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(    screen_width,screen_height);
    glutInitWindowPosition(0,0);
    glutCreateWindow("Lesson1: Drawing simple shapes by Osama Hosam");

    glutDisplayFunc(display);
    glutReshapeFunc (resize);
    init();
    glutMainLoop();
    return 0;
```

```
}
```

Initialization: In this function we will set the background color of the window to black, you can select any color you want.

```
void init(void)
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);

}
```

glClearColor is the function which sets the window's background color. glHint specifies the behavior, GL_PRESPECTIVE_CORRECTION_HINT is the behavior to be controlled. GL_NICEST is the desired behavior, the default values for both of them are GL_DONT_CARE.

Resizing: when the user wants to change the window size, specified calculations should be calculated. The aspect ratio should be changed due to the change in the window size

```
void resize (int w, int h)
{
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);

   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();
   gluOrtho2D (0, w, 0, h);


   glMatrixMode (GL_MODELVIEW);
   glLoadIdentity ();
}
```

glViewport is the function to specify the portion of the window which will display the OpenGL scene. In our example we selected the whole window to be our view port. Many View ports can be selected to display different scenes in the same window; this will be shown in the advanced lessons. The work of the glViewport can be shown in Fig. 1.3

The functions glMatrixmode, glLoadIdentity, and gluOrtho2D define an orthogonal projection to be used to map the contents of the rectangular area (2D) of the world coordinate to the screen. And the range of x-coordinates will be from 0 to w (width) and y-coordinates from 0 to h (height). The origin will be at the lower left corner of the window. Anything in the real world will range from 0 to w and 0 to h will lie inside the window and will be displayed, anything drawn outside this range will not displayed in the window.

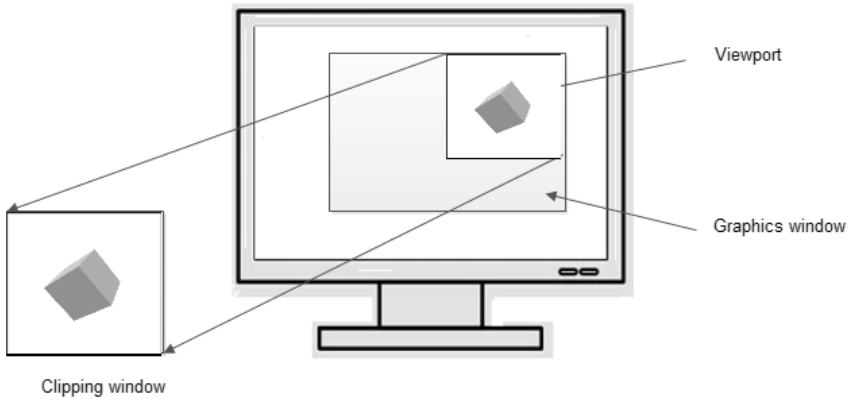


Fig. 1.3 glViewport with respect to the graphics window

Displaying: In this step we need to draw simple shapes, we are going to draw simple line and triangle. As we said in the resizing step, we need to define coordinates inside the range w, h so we will be able to display the object inside the window or the view-port.

```
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT );
   glLoadIdentity();

   glBegin(GL_LINES);
         glVertex2i(180,15);
         glVertex2i(10,145);
   glEnd();

   glBegin(GL_TRIANGLES);
         glVertex2i(200,200);
         glVertex2i(100,75);
         glVertex2i(250,100);
   glEnd();

   glutSwapBuffers();
}
```

glClear clears the buffer bit of the memory, then we drawn a Line by using to end vertices in 2D. For the triangle we have used the three vertices of its corners. See Fig. 1.4 for the results

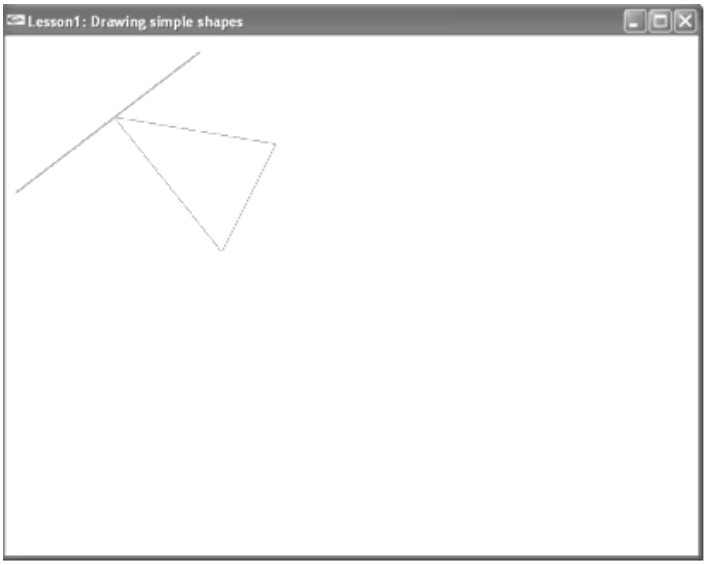


Fig. 1.4 drawing simple shapes (Line and Triangle)

1.6 Conclusion

In this chapter we developed the first example of OpenGL. The basics used in this chapter will be used in the remaining chapters of this book. We have learned how to use OpenGL to draw very simple shapes such as lines and triangles.

Chapter 2

OpenGL Coloring

2.1 Introduction
2.2 Computer Colors
2.3 Colors in OpenGL
2.4 Conclusion

2.1 Introduction

Everything around us is seen colored; when you are going to draw a scene in OpenGL you have to add some colors to the objects to make them similar to the real world objects. Colors have two end points

All colors: represented by the white color (the material of the object reflects all colors)

No colors: represented by the black color (the material of the object reflects no color)

In this chapter we are going to introduce how colors are represented in OpenGL, we will do some color blending to the objects; also we will show how to change the default color of the background of the OpenGL window.

2.2 Computer Colors

The colors in computer are represented by three components of other basic colors, namely Red, Green, Blue colors. The blending of the three colors gives us the desired color, for example

Black: Red=0, Green=0, Blue=0 (means no color)
Green: Red=0 Green=1, Blue=0
Yellow: Red=1, Green=1 Blue=0
White: Red=1, Green=1, Blue=1(all colors)

In Fig. 2.1 we see how the colors are represented by the computer monitor

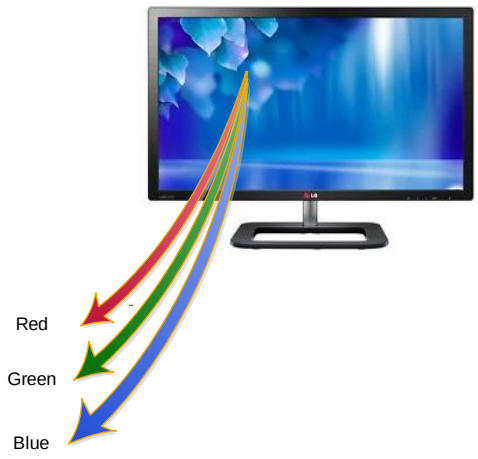


Fig. 2.1 Computer colors

The default color for OpenGL is white. We are going to use the function *glColor3f(redvalue, greenvalue, bluevalue)*to add colors to the objects.

2.3 Colors in OpenGL

Open new project in Visual C++ 6, “You can use the code from chapter 1 and update it to complete the current chapter’s code”. Update the init() function to become as follow

```
void init(void)
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
      glEnable(GL_COLOR_MATERIAL);
}
```

glEnable(GL_COLOR_MATERIAL) enable the coloring in OpenGL.
Also we are going to change the shapes in chapter 1 and use other shapes; this will help us understanding how shapes are drawn in OpenGL. For example to draw a Star you can use the glBegin(GL_LINE_LOOP) and connect the segments to form a Star, let us see how to do that in OpenGL

We will use structures in C++ which will be very helpful in developing large programs like a complete game. We need to define a structure for the point as follow

```
struct POINT
{
   GLfloat x;
   GLfloat y;
};
```

Then we will define the five points of the Star, p1, p2, p3, p4, and p5 (it is preferred to sketch the drawing on a paper and define the points before start coding). Then we will use the glBegin(GL_LINE_STRIP) to draw the Star which will connects each point to the other, for example p1 will be connected to p2, and p2, will be connected to p3, and p3 will be connected to p4 and p4 will be connect to p5 and finally p5 will be connected to p1 to form a line loop. Change the display function to be as follow

```
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT );
   glLoadIdentity();

   glColor3f(0.0f,0.0f,1.0f);
   POINT p1,p2,p3,p4,p5;
   p1.x=300;p1.y=200;
   p2.x=260;p2.y=300;
   p3.x=350;p3.y=240;
```

```
    p4.x=250;p4.y=240;
    p5.x=340;p5.y=300;

    glBegin(GL_LINE_LOOP);
          glVertex2i(p1.x,p1.y);
          glVertex2i(p2.x,p2.y);
          glVertex2i(p3.x,p3.y);
          glVertex2i(p4.x,p4.y);
          glVertex2i(p5.x,p5.y);
    glEnd();

    glutSwapBuffers();
}
```

The result of the above code will be as shown in Fig. 2.2 , as we can see we have used the Yellow color to color the lines. The shape is pretty simple but we will show you how to convert this shape into a more colorful polygon.

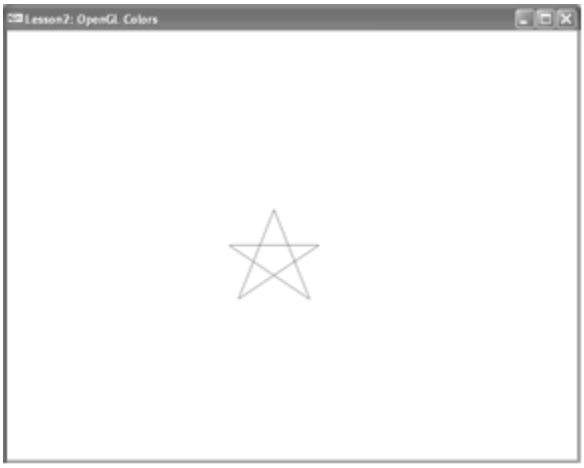


Fig. 2.2 drawing a Star

If you just change the glBegin(GL_LINE_LOOP) to glBegin(GL_POLYGON). The shape will not be a Pentagon as you may expect. Instead it will take non-uniform shape (try to do this as a practice). The reason of that is the order of the points. We have to change the order the points to start from point one p1 and rotating counter-clockwise, like rotating around a circle. The order of the points should be as follow p1, p4, p2, p5, and p3. This way we will get the desired Pentagon shape. As shown by the following code.

```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT );
    glLoadIdentity();
```

```
    glColor3f(0.0f,0.0f,1.0f);
    POINT p1,p2,p3,p4,p5;
    p1.x=300;p1.y=200;
    p2.x=260;p2.y=300;
    p3.x=350;p3.y=240;
    p4.x=250;p4.y=240;
    p5.x=340;p5.y=300;

    glBegin(GL_POLYGON);
          glVertex2i(p1.x,p1.y);
          glVertex2i(p4.x,p4.y);
          glVertex2i(p2.x,p2.y);
          glVertex2i(p5.x,p5.y);
          glVertex2i(p3.x,p3.y);
    glEnd();

    glutSwapBuffers();
}
```

The result of the above code is shown in Fig. 2.3, it is important to mention that if you draw other shapes after the Pentagon, they will all take the same color (Yellow in our example)

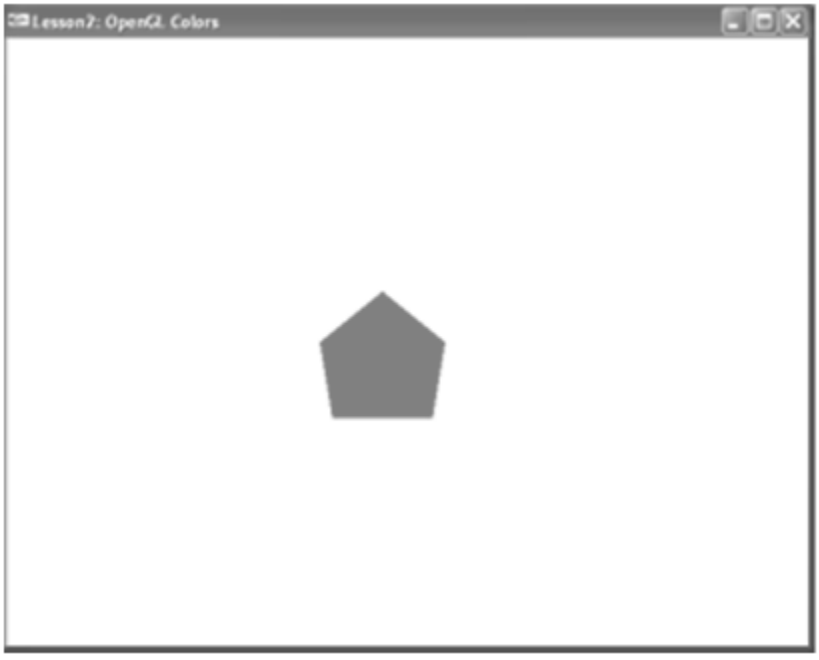


Fig.2.3 Drawing a Pentagon by using the same points in Fig.2.2

Now we will do some tweaks to our pentagon, so as to have the effect of blending colors. This can be done by specifying a different color for each vertex. Let us try using the following code for coloring the pentagon

```
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT );
   glLoadIdentity();

   POINT p1,p2,p3,p4,p5;
   p1.x=300;p1.y=200;
   p2.x=260;p2.y=300;
   p3.x=350;p3.y=240;
   p4.x=250;p4.y=240;
   p5.x=340;p5.y=300;

   glBegin(GL_POLYGON);
         glColor3f(1.0f,1.0f,0.0f);
         glVertex2i(p1.x,p1.y);
         glColor3f(0.0f,1.0f,0.5f);
         glVertex2i(p4.x,p4.y);
         glColor3f(0.4f,0.0f,0.5f);
         glVertex2i(p2.x,p2.y);
         glColor3f(0.2f,0.3f,1.0f);
         glVertex2i(p5.x,p5.y);
         glColor3f(0.7f,0.1f,0.0f);
         glVertex2i(p3.x,p3.y);
   glEnd();

   glutSwapBuffers();
}
```

.The result will be as shown in Fig. 2.4

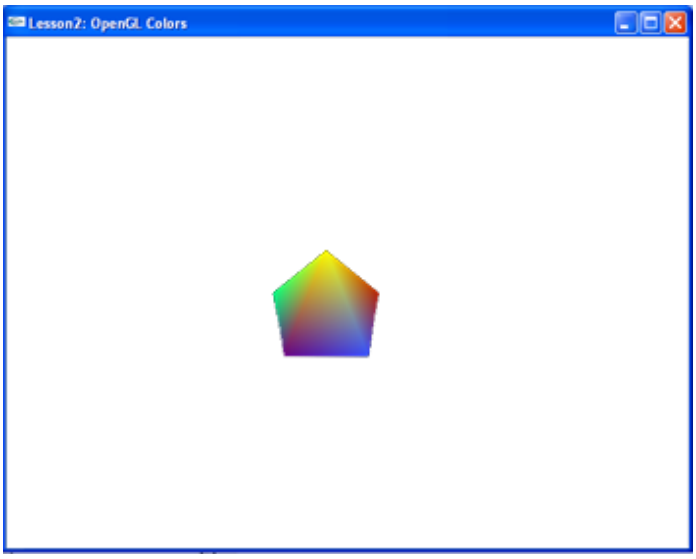


Fig. 2.4 The pentagon with blended colors

The last thing we will do here is to change the background color of our OpenGL window. The function which is responsible for this is

```
glClearColor (0.0, 0.0, 0.0, 0.0);
```

which is defined in the init() function. We will notice that the RGB colors are all zeros, means black. If we want to change this color to Sky blue color we can use the following RGB values

```
glClearColor (0.7, 0.9, 1.0, 0.0);
```

The result of the above change will be as shown in Fig. 2.5

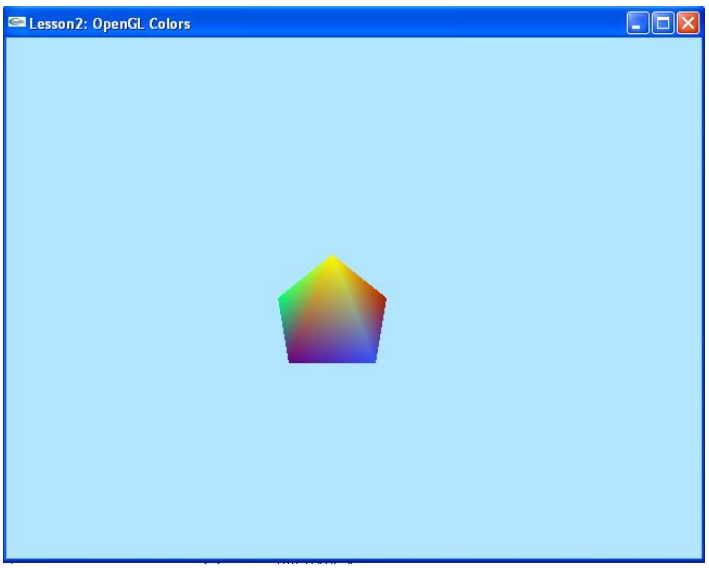


Fig. 2.5 The background colored with sky blue color

2.4 Conclusion

In this chapter we have developed simple program to show how to use colors in OpenGL. We have seen that the basic colors are Red, Green and Blue. Any color is a blend of the basic thre+e colors. OpenGL uses the RGB color system.

Chapter 3

Working with 3D Environment

3.1 Introduction
3.2 Object Transformations
3.3 Object Transformations in OpenGL
3.4 Conclusion

3.1 Introduction

Most of current games in market are built in the 3D environment, so it is clear that we find the look and feel of the real world environment. We will build a simple application that helps us to understand how to use OpenGL to build a real world 3D game. We are going to draw shapes in 3D, rotate and move them. First we will show the difference between the previous chapters which were 2D and this chapter which is 3D. Then we will explain in detail how to translate rotate and scale objects.

3.2 Object Transformations

The combination of rotation and translation is always done in OpenGL. With combining the two operations, you should take care of the order, since if you do the translation first and then do the rotation it will lead to another results than doing the opposite. Fig.3.1 shows an example of a cube, in the left part it is Rotated and then translated (the cub's initial position is at origin and the final position is on x-axis), in the right part it is translated and then rotated (the cube's final position is different and not on x-axis as the previous case)

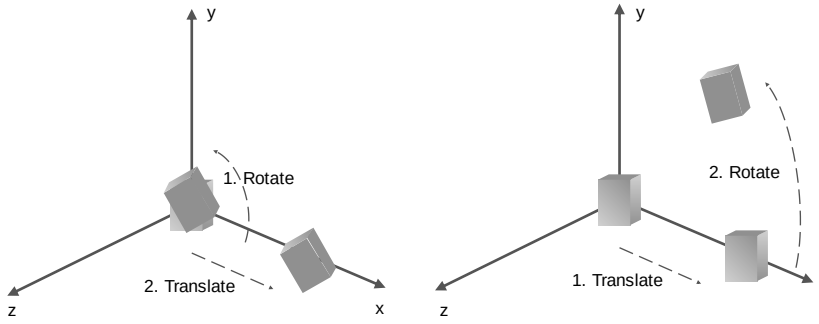


Fig. 3.1 Rotate-Translate combination

The object transformations are made in OpenGL with the following three modules

- glTranslate()
- glRotate()
- glScale()

glTranslate() is used to move the object from place to another, in other words you can say that we move the coordinate system by the amount of translation. Fig.3.2 shows the translation of a sample cube, it is done also by moving the coordinate system. The feel of translation can be seen as "You are moving" state this can occur if you are inside an object and you move the object so you feel you are moving, while the second state is "The object is moving" state in which the

object is small and you see it moving and rotating. “You are moving state” can be done by using the gluLookAt() function in OpenGL which changes the camera position or “Your” position.
In OpenGL if you are implementing a game in which you driving a car, If you move the car in the street and you see the car is becoming smaller when it goes into the screen, this reflects the “Object is moving” state while if the car is fixed and the platform and the road is moving, this is the “You are moving” state. The first state is done by translating the car object while the second state is done by translating the whole scene except the car.

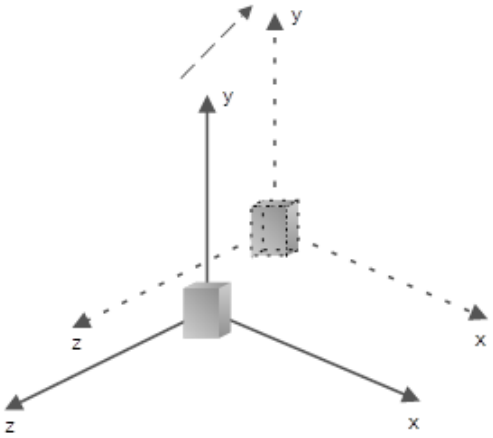


Fig.3.2 translating in 3D space

glRotate() is used to rotate an object counterclockwise around an axis. It is very important to notice that is the angle of rotation is zero; this means the object will not be rotated. Also if the object is near the rotating axis, it will move by small amount rather than an object which is positioned away from the rotating axis. glRotate is shown in Fig.3.3

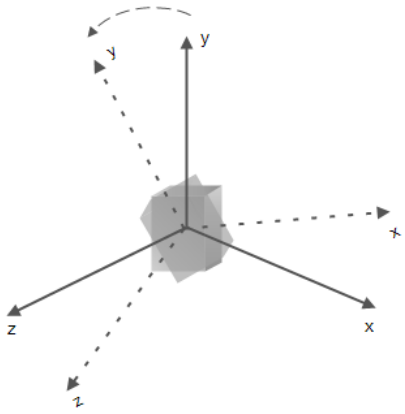


Fig.3.3 rotating in 3D space.

glScale() is used to shrink or stretch the object. glScale() is used also in zooming in or out the whole scene.

3.3 Object Transformations in OpenGL

The program from previous chapters can be used but with some changes. In the previous chapters we have used the orthogonal projection, while in this chapter we are going to use the perspective projection. The orthogonal projection is suitable to design simple 2D games while the perspective projection can be used to implement real world 3D games.

The OpenGL program structure should be as follow

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

#pragma comment (lib,"opengl32.lib")
#pragma comment (lib,"glu32.lib")
#pragma comment (lib,"glut32.lib")

void init()
{
   glClearColor(0.0,0.0,0.0,0.0);
   glEnable(GL_DEPTH_TEST);
   glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
   //glPolygonMode (GL_FRONT_AND_BACK, GL_LINE);

}

void render()
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity();
   glTranslatef(0,0,-3.0);

   //Draw your objects here
   glutSwapBuffers();

}
void reshape(int w,int h)
{
   glViewport(0,0,w,h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();

   gluPerspective(45.0f,(float)w/(float)h,1,1000);
```

```
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
int main(int argc, char* argv[])
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);

    //window settings
    glutInitWindowPosition(100,100);
    glutInitWindowSize(320,230);
    glutCreateWindow("Chapter 3: Working with 3D environment");

    glutDisplayFunc(render);
    glutIdleFunc(render);
    glutReshapeFunc(reshape);
    init();
    glutMainLoop();
    return 0;
}
```

gluPerspective() take the following form:

void **gluPerspective**(GLdouble *fovy*, GLdouble *aspect*,GLdouble *near*, GLdouble *far*);

fovy Is the field of view angle, aspect is the aspect ratio of the viewed scene. Near and far are the distances between the view point and the clipping window and they must be both positive. gluPerspective() is described in Fig.3.4

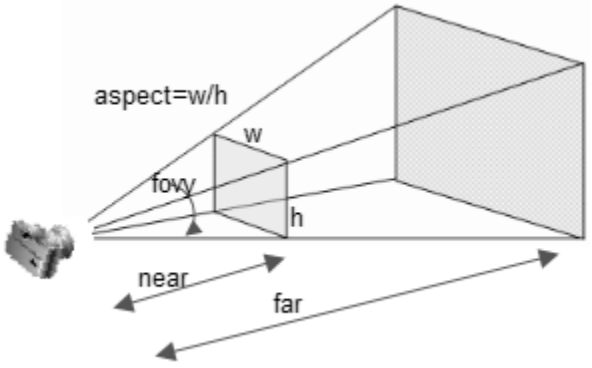


Fig.3.4 the perspective volume in OpenGL

glutIdleFunc(render), is very important, without it the scene will not be changed. It is telling OpenGL to execute the function render even in the Idle state.

An example of using gluPerspective, we will show how to render a cube in OpenGL and translate scale and rotate it. The code of the render function will be changed as follow:

```
void render()
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity();

   glTranslatef(-1.5f,0.0f,-10.0f);

   glRotatef(45,0,1,0);

   //Draw your objects here
   glBegin(GL_QUADS);
         glColor3f(0.0f,1.0f,0.0f);
         glVertex3f( 1.0f, 1.0f,-1.0f);
         glVertex3f(-1.0f, 1.0f,-1.0f);
         glVertex3f(-1.0f, 1.0f, 1.0f);
         glVertex3f( 1.0f, 1.0f, 1.0f);

         glColor3f(1.0f,0.5f,0.0f);
         glVertex3f( 1.0f,-1.0f, 1.0f);
         glVertex3f(-1.0f,-1.0f, 1.0f);
         glVertex3f(-1.0f,-1.0f,-1.0f);
         glVertex3f( 1.0f,-1.0f,-1.0f);


         glColor3f(1.0f,0.0f,0.0f);
         glVertex3f( 1.0f, 1.0f, 1.0f);
         glVertex3f(-1.0f, 1.0f, 1.0f);
         glVertex3f(-1.0f,-1.0f, 1.0f);
         glVertex3f( 1.0f,-1.0f, 1.0f);

         glColor3f(1.0f,1.0f,0.0f);
         glVertex3f( 1.0f,-1.0f,-1.0f);
         glVertex3f(-1.0f,-1.0f,-1.0f);
         glVertex3f(-1.0f, 1.0f,-1.0f);
         glVertex3f( 1.0f, 1.0f,-1.0f);

         glColor3f(0.0f,0.0f,1.0f);
         glVertex3f(-1.0f, 1.0f, 1.0f);
         glVertex3f(-1.0f, 1.0f,-1.0f);
         glVertex3f(-1.0f,-1.0f,-1.0f);
         glVertex3f(-1.0f,-1.0f, 1.0f);
```

```
            glColor3f(1.0f,0.0f,1.0f);
            glVertex3f( 1.0f, 1.0f,-1.0f);
            glVertex3f( 1.0f, 1.0f, 1.0f);
            glVertex3f( 1.0f,-1.0f, 1.0f);
            glVertex3f( 1.0f,-1.0f,-1.0f);

      glEnd();
      glutSwapBuffers();

}
```

We have changed the view point to be 1.5 units to the left and 10 units deep into the screen by using the glTranslate() function, then the cube is rotated 45 degrees around the y axis (notice the order of glRotate and glTranslate, if you changed the order the result will be completely different, you are free to play with the code to see the difference). glBegin(GL_QUADS) draws the cube with its six faces by specifying its vertex coordinates.

If we need to make the cube rotating around a specified axis, we need to define a global variable for holding the angle of rotation and then changing the value of the angle every time the scene is rendered. To do that we have changed the render function to be as follow:

```
float rot=0.01;
void render()
{
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      glLoadIdentity();

      glTranslatef(-1.5f,0.0f,-10.0f);

      glRotatef(rot++,0,1,0);
      rot=rot>=360?0:rot;


      //Draw your objects here
      glBegin(GL_QUADS);
            glColor3f(0.0f,1.0f,0.0f);
            glVertex3f( 1.0f, 1.0f,-1.0f);
            glVertex3f(-1.0f, 1.0f,-1.0f);
            glVertex3f(-1.0f, 1.0f, 1.0f);
            glVertex3f( 1.0f, 1.0f, 1.0f);

            glColor3f(1.0f,0.5f,0.0f);
            glVertex3f( 1.0f,-1.0f, 1.0f);
```

```
            glVertex3f(-1.0f,-1.0f, 1.0f);
            glVertex3f(-1.0f,-1.0f,-1.0f);
            glVertex3f( 1.0f,-1.0f,-1.0f);


            glColor3f(1.0f,0.0f,0.0f);
            glVertex3f( 1.0f, 1.0f, 1.0f);
            glVertex3f(-1.0f, 1.0f, 1.0f);
            glVertex3f(-1.0f,-1.0f, 1.0f);
            glVertex3f( 1.0f,-1.0f, 1.0f);

            glColor3f(1.0f,1.0f,0.0f);
            glVertex3f( 1.0f,-1.0f,-1.0f);
            glVertex3f(-1.0f,-1.0f,-1.0f);
            glVertex3f(-1.0f, 1.0f,-1.0f);
            glVertex3f( 1.0f, 1.0f,-1.0f);

            glColor3f(0.0f,0.0f,1.0f);
            glVertex3f(-1.0f, 1.0f, 1.0f);
            glVertex3f(-1.0f, 1.0f,-1.0f);
            glVertex3f(-1.0f,-1.0f,-1.0f);
            glVertex3f(-1.0f,-1.0f, 1.0f);

            glColor3f(1.0f,0.0f,1.0f);
            glVertex3f( 1.0f, 1.0f,-1.0f);
            glVertex3f( 1.0f, 1.0f, 1.0f);
            glVertex3f( 1.0f,-1.0f, 1.0f);
            glVertex3f( 1.0f,-1.0f,-1.0f);

    glEnd();
    glutSwapBuffers();

}
```

We have defined a variable called *rot* and we initialize its value to be 0.01 degrees. *rot* value will be changed in ***glRotatef(rot++,0,1,0);*** by increasing its value with 1, and then we check if *rot* reached the value 360 we reset it to 0 by the statement ***rot=rot>=360?0:rot;*** which literally means if (rot >= 360) rot=0; The result of the above code will be a rotating cube colored with different color in each of its sides. Fig. 3.5 shows the output of the OpenGL code.

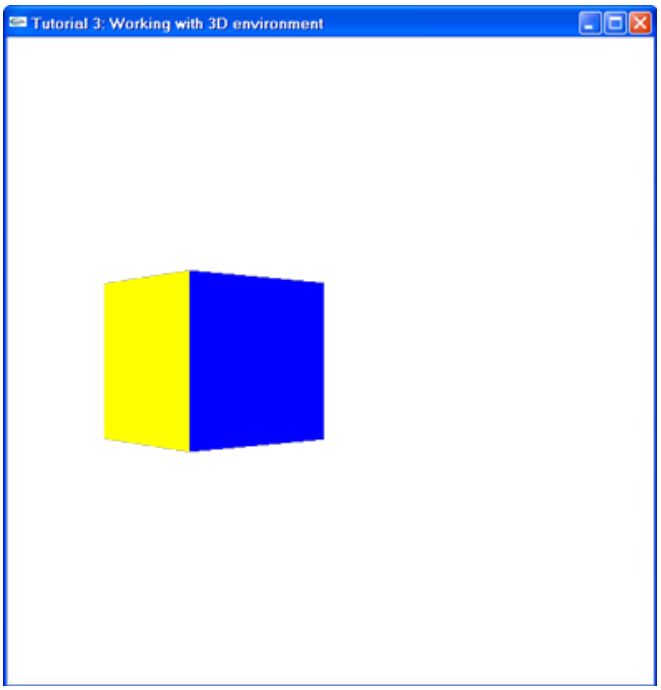


Fig. 3.5 The rotating cube output of the OpenGL program

3.4 Conclusion

In this chapter we have seen how to do transformations in OpenGL. Rotating an object is done with glRotate command, translating and object is done with glTranslate command, scaling an object is done with glScale command. Transformations can be combined to do complex movements and animation of an object. Object transformations are not commutative, so rotating and then translating of an object is different from translating and then rotating of the same object.

Chapter 4

Texturing

4.1 Introduction
4.2 Texturing in OpenGL
4.3 Loading the Texture and defining filters
4.4 Covering the model with the loaded texture
4.5 Defining Texture Coordinates
4.6 Conclusion

4.1 Introduction

Texturing is the way of covering objects with textures. In game design sometimes it is needed to have walls with the exact appearance of the real walls, to accomplish that we have two ways, the first way is to design a model for a wall and build the detailed bricks and tiles, the other way is to just build a simple rectangle which represents a face of the wall and cover it with an image containing the wall detailed graphics. The second way is our approach in this chapter, because it saves time for the programmer and let him do roads, grass fields, walls, sky, etc. with a few lines of code. In this chapter we are going to build a small cube and cover it with textures.

4.2 Texturing in OpenGL

Texturing takes time in coding since it needs long processes and calculations, especially if the image is of compact formats like jpg, gif. However, texturing in OpenGL takes few lines of code, texturing is done in the following stages

- Loading the image, loading the image from the file on disk.
- Creating a texture from the image, in this stage we will define the filters which will be applied on the texture.
- Covering the model with texture, in this phase we will define the coordinates of the model and the texture, then the texture will be mapped to the model.

4.3 Loading the Texture and defining filters

The image file will be opened then it will be assigned as a texture. The previous lesson's code will be used (it is a rotating cube in 3D) and we are going to load a texture to the faces of the cube. See the following code segment.

```
#include "stdafx.h"
# pragma comment (lib, "glaux.lib")
#include<GL/glut.h>
#include <stdlib.h>
#include <gl\glaux.h>

#pragma comment (lib,"opengl32.lib")
#pragma comment (lib,"glu32.lib")
#pragma comment (lib,"glut32.lib")

GLuint         ID_1;
void LoadTexture(char *FileName ,  GLuint *ID)
{
   FILE *File=NULL;
   File=fopen(FileName,"r");
   if (File)
```

```
    {
          fclose(File);
    }
    AUX_RGBImageRec *TextureImage=new(AUX_RGBImageRec) ;
    memset(TextureImage,0,sizeof(void *)*1);
    TextureImage = auxDIBImageLoad(FileName) ;

    glGenTextures(1,  ID);
    glBindTexture(GL_TEXTURE_2D, *ID);

    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
    glTexImage2D(GL_TEXTURE_2D,       0,       3,       TextureImage->sizeX,
    TextureImage->sizeY, 0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage->data);

    free(TextureImage->data);
    free(TextureImage);
}
```

In the above code we have added the glaux library which will help us loading the texture. Also the **`#include <stdlib.h>`**for the "free" function to work properly.
First we opened the file, and then we have defined a pointer point to an object of type AUX_RGBImageRec

```
typedef struct _AUX_RGBImageRec {
GLint sizeX, sizeY;
 unsigned char *data;
} AUX_RGBImageRec;
```

This structure contains the length and the height of the image that we are going to load. Notice that the image must be of type BMP for this structure to work properly also it needs to by of size 2 powered (256x256 or 512x512 or 1024x1024 etc.) in this chapter's code we have used an image of size 256x256 . Now TextureImage->sizeX is the width of the image and TextureImage->sizeY is the length of the image.
The function glGenTextures tells OpenGL to allocate space for number of 1 texture with its ID. Suppose we have two textures as follow

```
glGenTextures(1, ID);
glGenTextures(1, ID2);
```

Now if we want to update the first texture, we use

```
glBindTexture(GL_TEXTURE_2D,*ID);
```

For the second texture we use

```
glBindTexture(GL_TEXTURE_2D,*ID2);
```

For filters we have two types of filters`GL_TEXTURE_MAG_FILTER,GL_NEAREST,GL_TEXTURE_MIN_FILTER,GL_NEAREST` Which has lower quality, and`GL_TEXTURE_MAG_FILTER,GL_LINEAR, GL_TEXTURE_MIN_FILTER,GL_LINEAR` which has higher quality.

The function glTexImage2D loads the image into the variable ID, so the variable ID holds the position of all the data of the image. So TextureImage is no more needed so we delete it from the memory by using.

```
free(TextureImage->data);
free(TextureImage);
```

We will talk about the function glTexImage2D in more details, its general structure is

```
glTexImage2D (GLenum target, GLint level, GLint internalformat, GLsizei
width, GLsizei height, GLint border, GLenum format, GLenum type, const GLvoid
*pixels);
```

The first parameter will be always GL_TEXTURE_2D, the second parameter will be zero, it is related to some kind of texture called MIPMAPPING, The third parameter refers to the color system of the image (RED, GREEN, BLUE, ALPHA). 1 means RED. 2 means RED and ALPHA. 3 means RGB, 4 means RGBA. The fourth and fifth parameters are the width and height of the image. The sixth parameter is the border of the image we put zero as default. The seventh parameter takes a value from the following list

```
GL_COLOR_INDEX
GL_STENCIL_INDEX
GL_DEPTH_COMPONENT
GL_RED
GL_GREEN
GL_BLUE
GL_ALPHA
GL_RGB
GL_RGBA
GL_LUMINANCE
GL_LUMINANCE_ALPHA
```

We choose GL_RGB since this is the current format of our image. The eighth parameter is the type of data which will be loaded. We choose GL_UNSIGNED_BYTE.
So far, we have successfully loaded the image into memory, it is time to use it and bind it to the specified object.

4.4 Covering the model with the loaded texture

In the above procedures we just loaded the texture into memory, we need to enable texturing in OpenGL by updating the initialization function "init()" to be as

```
void init()
{
   glClearColor(0.0,0.0,0.0,0.0);

   glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
   LoadTexture("texture_bricks.bmp" , &ID_1);
   glEnable(GL_TEXTURE_2D);
   //glPolygonMode (GL_FRONT_AND_BACK, GL_LINE);

}
```

We are going to use the image file named "texture_bricks.bmp" saved on the same directory of the application, then we tell OpenGL to enable texture by using the function glEnable.
In the render() function we created a 3D cube then we covered only five sides of the cube, we left one side to see the internal structure of the cube. Now we can cover the texture with the texture as shown in the following code segment.

```
void render()
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity();

   glTranslatef(-1.5f,0.0f,-10.0f);

   glRotatef(rot+=0.1f,0,1,0);
   rot=rot>=360?0:rot;

   glBindTexture(GL_TEXTURE_2D,  ID_1);
   //Draw your objects here
   glBegin(GL_QUADS);
         //glColor3f(0.0f,1.0f,0.0f);
         glTexCoord2f(0.0f, 0.0f);
```

```
        glVertex3f( 1.0f, 1.0f,-1.0f);
        glTexCoord2f(1.0f, 0.0f);
        glVertex3f(-1.0f, 1.0f,-1.0f);
        glTexCoord2f(1.0f, 1.0f);
        glVertex3f(-1.0f, 1.0f, 1.0f);
        glTexCoord2f(0.0f, 1.0f);
        glVertex3f( 1.0f, 1.0f, 1.0f);

        //glColor3f(1.0f,0.5f,0.0f);
        glTexCoord2f(0.0f, 0.0f);
        glVertex3f( 1.0f,-1.0f, 1.0f);
        glTexCoord2f(1.0f, 0.0f);
        glVertex3f(-1.0f,-1.0f, 1.0f);
        glTexCoord2f(1.0f, 1.0f);
        glVertex3f(-1.0f,-1.0f,-1.0f);
        glTexCoord2f(0.0f, 1.0f);
        glVertex3f( 1.0f,-1.0f,-1.0f);


        //glColor3f(1.0f,0.0f,0.0f);
        glTexCoord2f(0.0f, 0.0f);
        glVertex3f( 1.0f, 1.0f, 1.0f);
        glTexCoord2f(1.0f, 0.0f);
        glVertex3f(-1.0f, 1.0f, 1.0f);
        glTexCoord2f(1.0f, 1.0f);
        glVertex3f(-1.0f,-1.0f, 1.0f);
        glTexCoord2f(0.0f, 1.0f);
        glVertex3f( 1.0f,-1.0f, 1.0f);

//      glColor3f(1.0f,1.0f,0.0f);
        glTexCoord2f(0.0f, 0.0f);
        glVertex3f( 1.0f,-1.0f,-1.0f);
        glTexCoord2f(1.0f, 0.0f);
        glVertex3f(-1.0f,-1.0f,-1.0f);
        glTexCoord2f(1.0f, 1.0f);
        glVertex3f(-1.0f, 1.0f,-1.0f);
        glTexCoord2f(0.0f, 1.0f);
        glVertex3f( 1.0f, 1.0f,-1.0f);

        //glColor3f(0.0f,0.0f,1.0f);
        glTexCoord2f(0.0f, 0.0f);
        glVertex3f(-1.0f, 1.0f, 1.0f);
        glTexCoord2f(1.0f, 0.0f);
        glVertex3f(-1.0f, 1.0f,-1.0f);
        glTexCoord2f(1.0f, 1.0f);
```

```
        glVertex3f(-1.0f,-1.0f,-1.0f);
        glTexCoord2f(0.0f, 1.0f);
        glVertex3f(-1.0f,-1.0f, 1.0f);

    //  glColor3f(1.0f,0.0f,1.0f);
        glTexCoord2f(0.0f, 0.0f);
        glVertex3f( 1.0f, 1.0f,-1.0f);
        glTexCoord2f(1.0f, 0.0f);
        glVertex3f( 1.0f, 1.0f, 1.0f);
        glTexCoord2f(1.0f, 1.0f);
        glVertex3f( 1.0f,-1.0f, 1.0f);
        glTexCoord2f(0.0f, 1.0f);
        glVertex3f( 1.0f,-1.0f,-1.0f);

    glEnd();
    glutSwapBuffers();

}
```

We have used the function glVertext3f which defines the coordinates of the texture, we will explain in the next section how to use this function to define the coordinates of the texture relative to the coordinates of each face of the cube. The output of our program should be as shown in Fig.1

Fig.1 Texturing a cube

We notice from the figure that the cube rotates around y. Imagine that you need to build a wall from scratch; it will need to be done in days if you will build all the detailed graphics from scratch.

4.5 Defining Texture Coordinates

Defining the coordinates is very important when covering the model with the texture, because if it is not known well this will lead to unexpected results, if you look carefully in the above example we notice the image contains a word “Texture Sample” written in blue on the texture. The word is flipped because we just do texturing without really understanding how the coordinates is defined, the coordinates of the texture is defined as shown in Fig.2

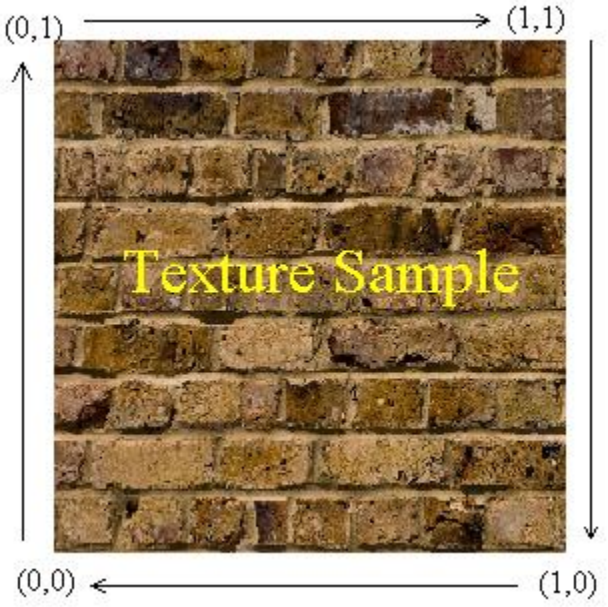


Fig.2 The texture coordinates

We will take one face of the rectangle to cover it with the above texture, suppose the rectangle has the following coordinates as shown Fig.3.

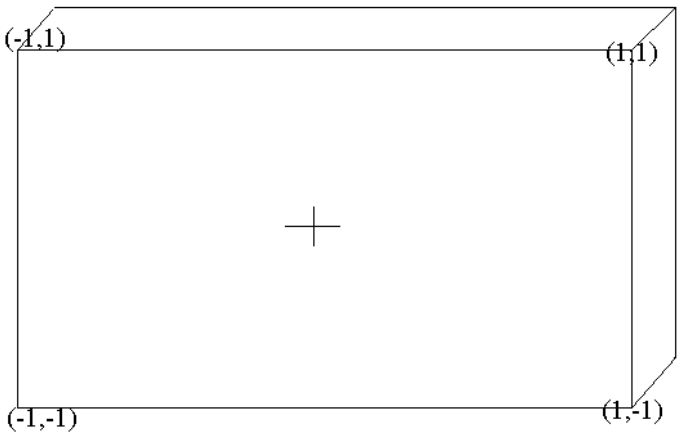


Fig.3 the coordinates of the cube's face which will be textured

The true texturing is to map the texture coordinate (1,1) to the cubes coordinate (1,1) then map the texture coordinate (1,0) to the cubes coordinate (1,-1), then map the texture coordinate (0,0) to the cubes coordinate (-1,-1), finally map the texture coordinate (0,1) to the cubes coordinate (-1,1).

Previously we have used the following code which has a wrong arrangement and mapping of the texture coordinates to the cube's coordinate.

```
glTexCoord2f(0.0f, 0.0f);
glVertex3f( 1.0f, 1.0f, 10.0f);
glTexCoord2f(1.0f, 0.0f);
glVertex3f(-1.0f, 1.0f, 10.0f);
glTexCoord2f(1.0f, 1.0f);
glVertex3f(-1.0f,-1.0f, 10.0f);
glTexCoord2f(0.0f, 1.0f);
glVertex3f( 1.0f,-1.0f, 10.0f);
```

As a result the image appeared flipped when rendered on the cube. The correct order should be

```
glTexCoord2f(1.0f, 1.0f);
glVertex3f( 1.0f, 1.0f, 10.0f);
glTexCoord2f(1.0f, 0.0f);
glVertex3f(1.0f, -1.0f, 10.0f);
glTexCoord2f(0.0f, 0.0f);
glVertex3f(-1.0f,-1.0f, 10.0f);
glTexCoord2f(0.0f, 1.0f);
glVertex3f( -1.0f, 1.0f, 10.0f);
```

As an exercise try to change the order of the texture coordinates for the remaining faces of the cube

4.6 Conclusion

We have introduced texture basics. In OpenGL texturing is done by loading the image from image file on the disk, then creating a texture from the image- in this stage the filters are defined, filters are applied on the texture, and finally covering the model with texture, in this phase we defined the coordinates of the model and the texture, then the texture is mapped to the model.

Chapter 5

Displaying Text

5.1 Introduction
5.2 Bitmap Fonts
5.3 Stroke fonts
5.4 Conclusion

5.1 Introduction

Text is the way of delivering messages. It is a way of interactivity between the user and the game. It can be used to post messages in multiplayer environments game. Also when programming, text can be used as a tool of debugging. For example, suppose there is a function to create a line rotation. The output of the function can be displayed at the top of the screen, so it will be easy to make sure if the output is true or not. In OpenGL's GLUT library we have two types of fonts

- Bitmap fonts, and
- Stroke fonts.

In this article we are going to show the difference between Bitmap fonts and Stroke fonts, we will demonstrate by example how to use both of them.

5.2 Bitmap Fonts

Bitmap fonts are basically 2D fonts. They can't be rotated or scaled, they can only be translated. This type of fonts is suitable for the 2D games to display for example the current score or level of the game. The function to draw bitmap fonts is

```
void drawBitmapText(char *string,float x,float y,float z)
{
   char *c;
   glRasterPos3f(x, y,z);

   for (c=string; *c != '\0'; c++)
   {
         glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_10, *c);
   }
}
```

When you draw a text you have to define "Where" it will be drawn, the following functions are responsible for this role

```
void glRasterPos2f(float x, float y);
void glRasterPos3f(float x, float y, float z);
```

The function takes the x, y, z coordinates of the text which will be displayed. The suitable one here is glRasterPos2f() because the z coordinate will not be needed in the bitmap fonts since it is basically 2D.

We iterate through the string character by character to display it by using the function

```
glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_10, *c);
```

which takes as an argument the font type and the character to be displayed. The font type can be one of the following fonts

```
GLUT_BITMAP_8_BY_13
GLUT_BITMAP_9_BY_15
GLUT_BITMAP_TIMES_ROMAN_10
GLUT_BITMAP_TIMES_ROMAN_24
GLUT_BITMAP_HELVETICA_10
GLUT_BITMAP_HELVETICA_12
GLUT_BITMAP_HELVETICA_18
```

You are invited to try all of them, also you can create a menu in your application and try to select from a font from the menu and see each font separately.
The function glutBitmapCharacter() renders the character at the required position and advances the current raster position by the width of the character. Therefore, to render a string, successive calls to glutBitmapCharacter() will suffice to achieve the desired output.
After coding the function to draw a bitmap font, it is time to use it in our program, we will do that in the render() function, here is a code showing how to display the text “Osama Hosam’s OpenGL Tutorials” at the raster position 200,200 of the window

```
void render(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glLoadIdentity();

   drawBitmapText("Osama Hosam's OpenGL Tutorials",200,200,0);
   glutSwapBuffers();
}
```

The output is shown in Fig. 5.1

Fig. 5.1 Displaying bitmap fonts in OpenGL

5.3 Stroke fonts

Stroke fonts are basically 3D fonts. OpenGL deals with them as any other object in the scene. They can be rotated, scaled, or translated. The basic function which is used in displaying stroke fonts is

```
void drawStrokeText(char*string,int x,int y,int z)
{
      char *c;
      glPushMatrix();
      glTranslatef(x, y+8,z);
      glScalef(0.09f,-0.08f,z);

      for (c=string; *c != '\0'; c++)
      {
          glutStrokeCharacter(GLUT_STROKE_ROMAN , *c);
      }
      glPopMatrix();
}
```

The function iterates on all the characters and displays them. The function which display each character is

```
glutStrokeCharacter(GLUT_STROKE_ROMAN , *c);
```

The first argument is the font's name; a font from the following list can be used

```
GLUT_STROKE_ROMAN
GLUT_STROKE_MONO_ROMAN (fixed width font: 104.76 units wide).
```

OpenGL renders stroke fonts with lines, the line width can be specified by using the function glLineWidth() which takes only one parameter represents the width of the line. Also the stroke width can be specified by using the function

```
int glutStrokeWidth(void *font, int character)
```

which takes the font name (one of the above mentioned list) and the character.
Now, we have implemented the function to display the text, we will use it in the render() function to display the text "Osama Hosam's OpenGL Tutorials" as follow

```
void render(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glLoadIdentity();

   glColor3f(0,1,0);
   drawStrokeText("Osama Hosam's OpenGL Tutorials",200,200,0);

   glutSwapBuffers();
}
```

We have colored the font with green, the output will be as shown in Fig.5.2. The width and height of the font can also be done with the function

```
glScalef(0.09f,-0.08f,z);
```

We have scaled the font with negative value in the y direction since the origin of our coordinate system is located at the top left corner of the screen. If we don't use the glScalef() function the output will be the text flipped as shown in Fig.5.3.

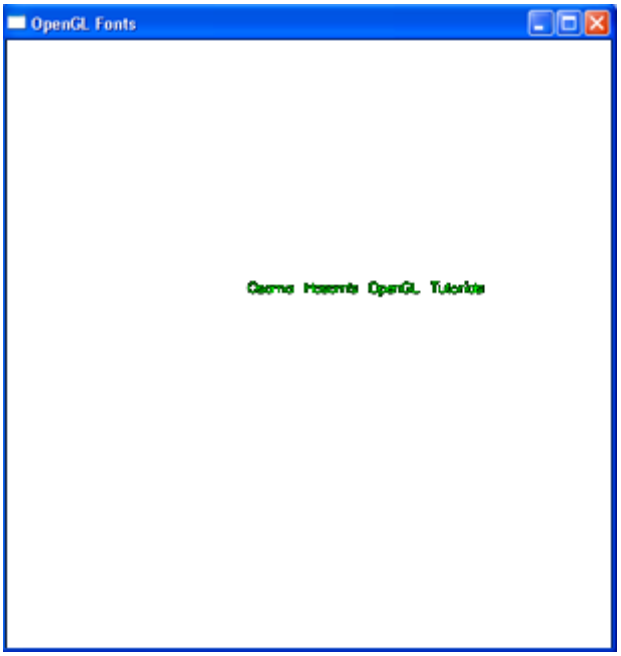


Fig.5.2 An example of stroke fonts

Fig. 5.3 Stroke font without using the glScale function

5.4 Conclusion

Screen text is important in games, it used as a communication tool between players. We have seen two types of text fonts, Bitmap fonts and Stroke fonts. Bitmap fonts are basically an image for each letter displayed on the screen as 2D image, it can be only translated, it can't be rotated or scaled. Stroke fonts are 3D fonts, it can be rotated scaled and translated.

Chapter 6

OpenGL Primitives

6.1 Introduction
6.2 Vectors
6.3 The bouncing balls with vectors
6.4 Finding the parallel line
6.5 Drawing the circles
6.6 Line-Circle collision
6.7 Circle-Circle collision
6.8 The collision by using vectors
6.9 Conclusion

6.1 Introduction

Vectors are very important in implementing games. They are used to define the direction of an object also defines the velocity which needs direction and speed. We are going to introduce the algebra of vectors, and then we will show how to use vectors in our programs by applying a simple example. An example of two bouncing balls inside a box will be introduced. The vectors will be used to define the direction of the circles (balls) while moving, also it will define the direction after bouncing. Two types of bouncing (collision) will be introduced; the first is line-circle collision and the second is circle-circle collision.

6.2 Vectors

The vector specifies a direction and magnitude and has no location. It can be represented by 3D coordinate. The addition of two vectors is represented in Fig.6.1 To add two vectors we use the parallelogram rule.

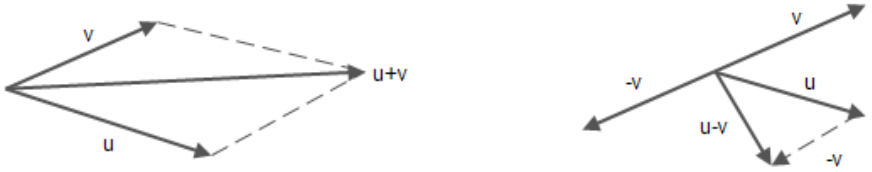


Fig.6.1 Vector addition and subtraction

The dot product of two vectors or more generally the inner product of two vectors is a scalar

$v1 \bullet v2 = x1x2 + y1y2 + z1z2$

the dot product is useful in defining the magnitude of the vector, also is can be used to define the angle between the two vectors.

The cross product or the vector product of two vectors is a vector; the direction of the resulting vector is perpendicular on both the original vectors. The right-hand rule is used to dictate the direction of the cross product.

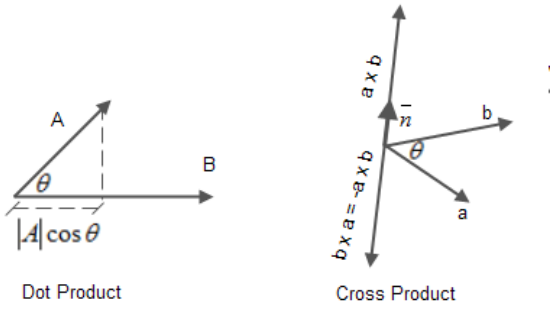


Fig.6.2 Dot product and cross product of two vectors

Combining the cross product and the dot product we can get the angle between two lines. The dot product of two vectors a,b is given by

$$a.b = |a| \ |b| \ \cos\theta \quad \text{..... (1)}$$

The cross product is given by

$$|a \times b| = |a| \ |b| \ \sin\theta \quad \text{.....(2)}$$

The dot product and cross product is shown in Figure 6.2. The cross product is a vector perpendicular to both a,b. solving equations (1) and (2) together gives

$$\tan\theta = \frac{|a \times b|}{a.b} \quad \text{.....(3)}$$

From equation (3) we can get the angle between the two lines.

We have implemented the Vector3D class, we defined the most useful features of vectors the implementation of the Vector3D is shown in the following code segment

```
class Vector3D
{
public:
   float x, y, z;
   public:
         virtual ~Vector3D();
    // three-dimensional Cartesian coordinates


        // constructors
        Vector3D (void) {x = y = z = 0;}
        Vector3D (float X, float Y, float Z) {x = X; y = Y; z = Z;}

        // vector addition
          Vector3D operator+ (const Vector3D& v) const {return Vector3D
         (x+v.x, y+v.y, z+v.z);}

        // vector subtraction
          Vector3D operator- (const Vector3D& v) const {return Vector3D
         (x-v.x, y-v.y, z-v.z);}

        // unary minus
        Vector3D operator- (void) const {return Vector3D (-x, -y, -z);}
```

```
 // vector times scalar product (scale length of vector times
 //argument)
Vector3D operator* (const float s) const
 {return Vector3D (x * s, y * s, z * s);}

// vector divided by a scalar (divide length of vector by
argument)
Vector3D operator/ (const float s) const
 {return Vector3D (x / s, y / s, z / s);}

// dot product
 float dot (const Vector3D& v) const {return (x * v.x) + (y * v.y)
 + (z * v.z);}

// length
float length (void) const {return (float)sqrt(lengthSquared ());}

// length squared
float lengthSquared (void) const {return this->dot (*this);}

// normalize: returns normalized version (parallel to this, length
= 1)
Vector3D normalize (void) const
{
    // skip divide if length is zero
    const float len = length ();
    return (len>0) ? (*this)/len : (*this);
}

// cross product
void cross(const Vector3D& a, const Vector3D& b)
{
    *this = Vector3D ((a.y * b.z) - (a.z * b.y),
                  (a.z * b.x) - (a.x * b.z),
                  (a.x * b.y) - (a.y * b.x));
}

// assignment
 Vector3D operator= (const Vector3D& v) {x=v.x; y=v.y; z=v.z;
 return *this;}

// set XYZ coordinates to given three floats
Vector3D set (const float _x, const float _y, const float _z)
{x = _x; y = _y; z = _z; return *this;}
```

```
        // +=
         Vector3D operator+= (const Vector3D& v) {return *this = (*this +
         v);}

        // -=
         Vector3D operator-= (const Vector3D& v) {return *this = (*this -
         v);}

        // *=
        Vector3D operator*= (const float& s) {return *this = (*this * s);}

        // equality/inequality
         bool operator== (const Vector3D& v) const {return x==v.x &&
         y==v.y && z==v.z;}
        bool operator!= (const Vector3D& v) const {return !(*this == v);}

};
```

The operator overloading is used extensively to define the addition, subtraction and multiplication operations. The result of the dot product is a scalar (float), and the cross product produces a vector.
Normalizing vector means truncating its length to the unity vector.

6.3The bouncing balls with vectors

The example shows two bouncing balls, colliding in a bounding box, first we will draw the bounding box. Each edge will be a parallel line to the corresponding window edge. The circles will be drawn by the use of GL_POINTS; the polar representation of the circles will be used. The output of our program should be as shown in Fig.6.3.
We will consider the top edge of the window (not the white line) as a line and we will draw another line parallel to that edge and separated from it with defined offset. This edge is the white line in Fig.3, this procedure will be applied to the remaining edges to create the bouncing box.

6.4Finding the parallel line

Suppose we have the following equation for a line

$a.x + b.y + c = 0 \qquad (with \quad a^2 + b^2 = 1)$ .....(4)

The offset line is given by

$a.x(t) + b.y(t) + c + k.t = 0$ .....(5)

Where *a, b, c* are the line's normalized coefficients, we assume that (*a, b*) standsfor the inwards normal vector of the line. The sliding direction is given by k = ±1. k indicates whether the offset line is situated to theleft (-1) or to the right of the line (+1).

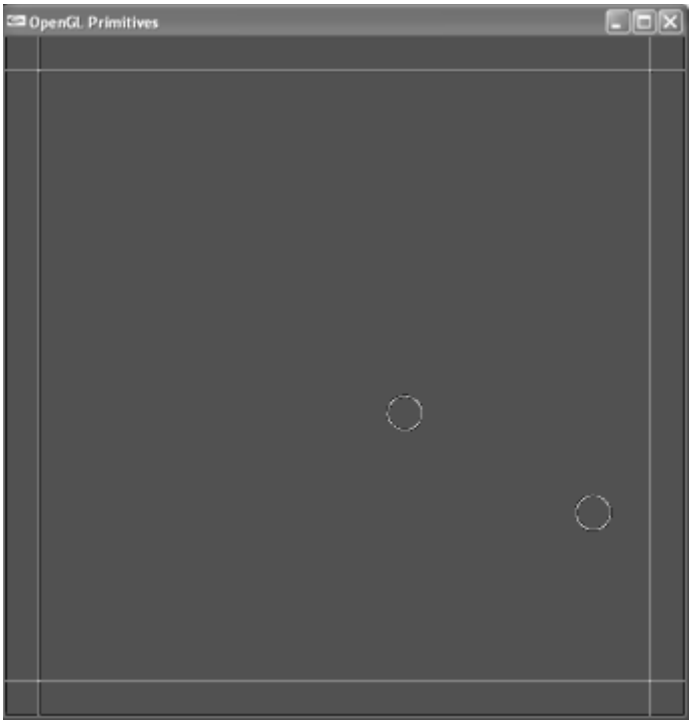


Fig.6.3 The output of our example program

The implementation of the above procedure of finding the parallel line is shown by the following code segment

```
inline CLine getParallelLine(int sliding_direction,float offset)
{
	float dx,dy;
	Vector3D parallelLineStartPoint3D,parallelLineEndPoint3D;
	CLine parallelLine;
	dx=endPoint3D.x-startPoint3D.x;
	dy=endPoint3D.y-startPoint3D.y;

	//The perpendicular direction will be
	//k is the sliding direction

	normalDirection3D.x=dy*sliding_direction;
	normalDirection3D.y=-dx*sliding_direction;

	//normalize the direction
	normalDirection3D=normalDirection3D.normalize();

	parallelLine.setNormal(normalDirection3D);

	//multiply by the desired offset
```

```
        parallelLineStartPoint3D = startPoint3D + normalDirection3D *
        offset;
        parallelLineEndPoint3D = endPoint3D + normalDirection3D * offset;

        parallelLine.setStartPoint(parallelLineStartPoint3D);
        parallelLine.setEndPoint(parallelLineEndPoint3D);

        return parallelLine;

    }
```

The was defined in the class CLine which defines the line by a starting point and ending point and a normal direction.

6.5Drawing the circles

The circles will be drawn by using the polar representation of the circle which is defined by

$$x = a + r\cos\theta,$$
$$y = b + r\sin\theta \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (6)$$

We will start from θ=0 to θ=359 or generally the circle will be defined for a point in each angle of its 360 degrees. The code for drawing a circle is

```
 inline void Draw()
 {
    glBegin(GL_POINTS);

          for(int angle=0;angle<360;angle++)
          {
                glPointSize(border_size);
                float x,y,z;
                x=position3D.x+radius*cos(angle*(PI/180));
                y=position3D.y+radius*sin(angle*(PI/180));
                z=0;
                glVertex3f(x, y, z);
          }

    glEnd( );
 }
```

The code is self explanatory, the loop starts with angle = 0 and ends with angle = 359. since we are working in the 2D environment we will set z=0. glPointSize defines the size of the point and glVertext3f defines the coordinates of the point.

6.6Line-Circle collision

To find the points of intersection between a line and a circle, the code considers the line as generated by the equations:

$$X(t) = x_1 + (x_2 - x_1).t$$
$$Y(t) = y_1 + (y_2 - y_1).t \quad \text{..... (7)}$$

where *t* ranges from 0 to 1 to draw the line segment.The code plugs these equations into the equation for a circle:

$$(X - C_x)^2 + (Y - C_y)^2 = r^2 \quad \text{..... (8)}$$

We then solve for *t*by using the quadratic formula as follow,

Plugging (8) into (7) we get

$$(x_1 + (x_2 - x_1).t - C_x)^2 + (y_1 + (y_2 - y_1).t - C_y)^2 = r^2 \quad \text{.....(9)}$$

Then (9) will be reduced into the following quadratic form

$$At^2 + Bt + C = 0 \quad \text{.....(10)}$$

Where

$A = (x_2 - x_1)^2 + (y_2 - y_1)^2$,

$B = 2((x_2 - x_1).(x_1 - C_x) + (y_2 - y_1).(y_1 - C_y))$ and

$C = (x_1 - C_x)^2 + (y_1 - C_y)^2 - r^2$

Then we solve (10) for *t* by using the following formula

$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A} \quad \text{.....(11)}$$

The result is 0, 1, or 2 values for *t*. those values are plugged back into the equations (7) to get the points of intersection.

The code for finding the line circle collision is

```
inline bool IsIntersectedWithLine(CLine myLine)
      {
            //Line circle intersection
            Vector3D intersectP1,intersectP2;
            Vector3D startPoint3D=myLine.getStartPoint();
            Vector3D endPoint3D=myLine.getEndPoint();
            float x1,y1,Cx,Cy;
            //the center of the circle
            Cx=position3D.x;
            Cy=position3D.y;

            x1=startPoint3D.x;
            y1=startPoint3D.y;

            float dx,dy,A,B,C;
```

```
        dx = endPoint3D.x-startPoint3D.x;
        dy = endPoint3D.y-startPoint3D.y;

        A = dx * dx + dy * dy;
        B = 2 * (dx * (x1 - Cx) + dy * (y1 - Cy));
              C = (x1 - Cx) * (x1 - Cx) + (y1 - Cy) * (y1 - Cy) -
              radius * radius;

        float det = B * B - 4 * A * C;
        float t;

        if(A <= 0.0000001 || det <0)
              return false;
        else if(det==0)
        {
              //one solution
              t=-B/(2*A);
              intersectP1.x=x1+t*dx;
              intersectP1.y=y1+t*dy;
              return true;
        }
        else
        {
              //two solutions

              t=(-B+sqrt(det))/(2*A);
              intersectP1.x=x1+t*dx;
              intersectP1.y=y1+t*dy;

                                  //one solution
              t=(-B-sqrt(det))/(2*A);
              intersectP2.x=x1+t*dx;
              intersectP2.y=y1+t*dy;

              return true;
        }
  }
```

We have defined another class called CCircle which is defined by a position, direction and radius. The direction vector is very important since it will be used to define the direction of the ball after collision.

6.7Circle-Circle collision

Consider Fig.6.4 showing two circles with radii r_0 and r_1. The points p_0, p_1, p_2, and p_3 have coordinates (x_0, y_0) and so forth.

Let d = the distance between the circles' centers so

$$d = \sqrt{(x_1 - x_0) + (y_1 - y_0)} = a + b \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{(12)}$$

Solving for *a* gives

$$a = d - b \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{(13)}$$

Now there are three cases:

- If $d > r_0 + r_1$: The circles are too far apart to intersect.
- If $d < |r_0 - r_1|$: One circle is inside the other so there is no intersection.
- If $d = 0$ and $r_0 = r_1$: The circles are the same.
- If $d = r_0 + r_1$: The circles touch at a single point.
- Otherwise: The circles touch at two points.

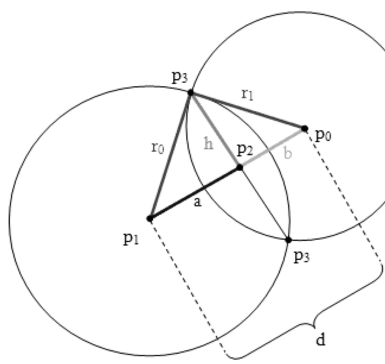


Fig.6.4 Circle-Circle intersections

The Pythagorean Theorem gives:

$$h^2 = r_0^{\,2} - a^2 \quad and \quad h^2 = r_1^{\,2} - b^2 \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{(14)}$$

So:

$$r_0^{\,2} - a^2 = r_1^{\,2} - b^2 \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{(15)}$$

Substituting *a* from equation (13) and multiplying this out gives:

$$r_0^{\,2} - d^2 + 2.d.b - b^2 = r_1^{\,2} - b^2 \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{(16)}$$

The $-b^2$ terms on each side cancel out. We can then solve for b to get:

$$b = (r_1^{\,2} - r_0^{\,2} + d^2)/(2.d) \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{(17)}$$

Similarly:

$$a = (r_0^{\,2} - r_1^{\,2} + d^2)/(2.d) \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{(18)}$$

All of these values are known so you can solve for *a* and *b*. All that remains is using those distances to find the point p_3.

If a line points in direction, then two perpendicular lines point in the directions <dy, -dx> and <-dy, dx>. Scaling the result gives the following coordinates for the points p_3:

$$x_3 = x_2 \pm h(y_1 - y_0),\ y_3 = y_2 \mp h(x_1 - x_0) \quad \text{(19)}$$

Be careful to notice the ± and ∓ symbols.

The implementation of the above procedure is shown in the following code (it is simplified for clarity).

```
inline bool IsIntersectedWithCircle(CCircle myCircle)
{
	//circle circle intersection
	float cx0,cy0,radius0;
	cx0=this->position3D.x;
	cy0=this->position3D.y;
	radius0=this->radius;

	float cx1,cy1,radius1;
	cx1=myCircle.getPosition().x;
	cy1=myCircle.getPosition().y;
	radius1=myCircle.getRadius();

	float dx,dy;
	float dist;
	//float a, h, cx2, cy2;

	dx = cx0 - cx1;
	dy = cy0 - cy1;
	dist = sqrt(dx * dx + dy * dy);

	if(dist > (radius0 + radius1))
		//no solutions the circles are too far apart
		return false;
	else if(dist < abs(radius0 - radius1))
		//no solution one circle contatins the other
		return false;
	else if((dist == 0) && (radius0 == radius1))
		//the circles coincide
		return false;
	else
	{
		return true;
	}
```

```
	}
```

6.8The collision by using vectors

In our program, we have used two CCircle objects (firstCircle, secondCircle) and four CLine objects (upLine, downLine, rightLine, leftLine), first we need to draw the four lines, we will draw the four lines in the reshape function, this will force the lines to have the new coordinates relative to the new size of the window. Drawing lines is shown in the following code

```
void reshape(int w,int h)
{
	win_width=w;
	win_height=h;

    glViewport(0,0,w,h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0,w,h,0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

                //initialize the for sides of the screen
      upLine.setStartPoint(Vector3D(0,0,0));upLine.setEndPoint(Vector3D(win_w
      idth,0,0));
      .setStartPoint(Vector3D(0,win_height,0));downLine.setEndPoint(Vector3D(
      win_width,win_height,0));
      rightLine.setStartPoint(Vector3D(win_width,0,0));rightLine.setEndPoint(
      Vector3D(win_width,win_height,0));
      leftLine.setStartPoint(Vector3D(0,0,0));leftLine.setEndPoint(Vector3D(0
      ,win_height,0));

      upLine=upLine.getParallelLine(-1,30);
      downLine=downLine.getParallelLine(1,30);
      rightLine=rightLine.getParallelLine(-1,30);
      leftLine=leftLine.getParallelLine(1,30);

}
```

We set the startPoint and endPoint of each line as the borders of the window. The lines will be shifted by 30 and drawn inward (by using the sliding direction +1, -1). The getParallelLine will get the parallel components of the edges of the window.

When a ball is intersected with line it will be diverted (reflected) back to the bounding box, the direction of the ball after colliding with a line of the box will be the summation of the normal vector of the line and the current direction of the ball. In Fig.6.5 (a) we can conclude that

-U + V = N

where U is the original direction of the ball and V is the direction after collision, we have used the vector addition to get V, solving for V we get

V= N+U

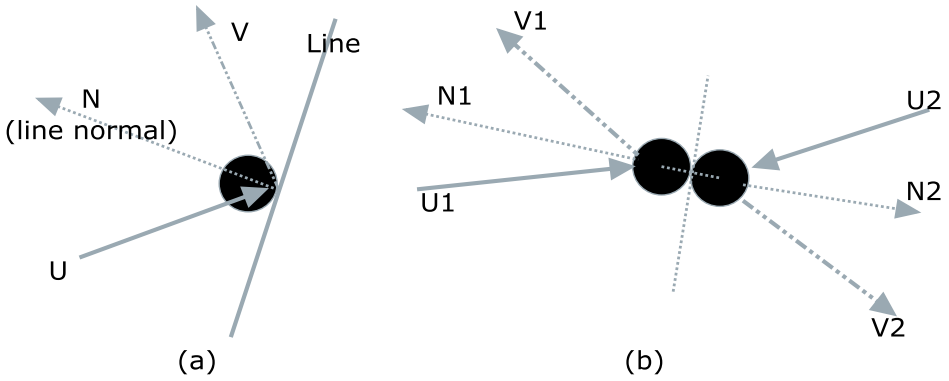


Fig.6.5 collision detection by using vectors (a) ball line collision (b) circle-circle collision

We introduced the code for detecting line-circle collision in the render function, as an example we take the upLine and detect the collision between it and the firstCircle as follow

```
if(firstCircle.IsIntersectedWithLine(upLine) )
{
      //change the direction of the circle (reflect it)
      firstCircle.setDirection((upLine.getNormal()+firstCircle.getDirection()
).normalize());
}
```

We changed the firstCircle direction to the summation of the normal of the line and the current direction of the firstCircle.

For the two circles, when colliding we need to get the tangent line's normal, this direction is obtained by subtracting the first circle position from the second circle position. We then use the vector addition to get the new direction for each ball as follow

V1= N1 + U1

V2= -N1 + U2

The implementation of the collision between the two balls will is shown in the following code

```
if(firstCircle.IsIntersectedWithCircle(secondCircle))
{
            //get the direction of the circles toward each other
```

```
      //the direction is from firstCircle center to secondCircle center
      Vector3D circleCircleDirection =secondCircle.getPosition()-
firstCircle.getPosition();

      circleCircleDirection=circleCircleDirection.normalize();

      firstCircle.setDirection((firstCircle.getDirection()-
      circleCircleDirection).normalize());
      secondCircle.setDirection((secondCircle.getDirection()+circleCircleDire
ction).normalize());
}
```

We define the direction between the two centers of the circles then we change the direction of the two balls to the new direction which is obtained from vector addition.

6.9 Conclusion

We have introduced the concept of vectors and how to use them in object motion. As an example a bouncing ball example is introduced. The ball is bounced against the wall and a collision is done between pair of balls. The collision detection is studied through the bouncing balls example. Finally, we obtained the general framework of using vectors in collision calculations.

Chapter 7

Loading 3D Max files

7.1 Introduction

7.2 3D Loading concepts

7.3 3DS file structure

7.4 The program structure

7.5 Loading the object to OpenGL

7.1 Introduction

When you work with OpenGL you will find the environment without modeling tools. Actually, OpenGL was not designed basically for creating models; instead it is for animation and rendering. However you still able to create models with OpenGL but this will lead to long time developing and wasting time which can be reduced by other techniques. One of these techniques – which is the subject of this chapter- is to create the model with modeling software like 3D max and load it to OpenGL.

7.2 3D Loading concepts

Models will be created in 3D Max and then saved with the extension .3DS This extension defines a type of files with a specific structure. Our objective is to get into this file and extract two types of information

- Vertices
- Polygons

The models will be defined by triangular mesh. The basic polygon of the mesh is a triangular which has three vertices. See Fig.7.1 to understand this concept

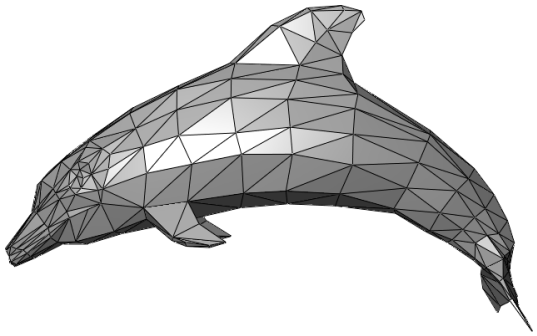


Fig.7.1 Dolphin represented with triangular mesh

In the figure, the Dolphin is represented entirely by a collection of rectangles. Each rectangle has three vertices (x, y, z) our objective is to read the rectangle collection from the file and draw it in OpenGL

7.3 3DS file structure

To read the 3DS files you need to understand the structure of that type of files. The files contain "Chunks". Every chunk is defined by a hexadecimal number. Each chunk type is different from the other, the differentiation is done by the hexadecimal number or the ID of the chunk, the following are the list for the chunks available:

- Chunk ID: 4d4d
  Description: Main chunk, contains all the other chunks

- Chunk ID: 3d3d
  Description: 3D Editor chunk, objects layout info
- Chunk ID: 4000
  Description: Object block, info for each object
- Chunk ID: 4100
  Description: Triangular mesh, contains chunks for 3d mesh info
- Chunk ID: 4110
  Description: Vertices list
- Chunk ID: 4120
  Description: Polygons (faces) list

The last two types are the most important chunks, since they hold the whole structure information of our model. However there are many other types of chunks which are not listed here, like the chunk represents the model shading, lights and materials. We removed them from this chapter for the sake of simplicity.

7.4 The program structure

We have implemented a class called CThreeMaxLoader, the class contains the following structures and global constants which are implemented into the same file assembly "ThreeMaxLoader.h"

```
#define MAX_VERTICES 80000 // Max number of vertices (for each object)
#define MAX_POLYGONS 80000 // Max number of polygons (for each object)
```

The above two variables put the limit of the number of vertices to 80000 and for the number of polygons to 80000, extra vertices or polygons will not be considered, if you have a huge model you need to change the above two variables.

```
// Our vertex type
typedef struct{
    float x,y,z;
}vertex_type;
```

The vertex_type holds the vertex coordinates, since we will work in the 3D environment we need to define x, y, and z.

```
// The polygon (triangle), 3 numbers that aim 3 vertices
typedef struct{
    int a,b,c;
}polygon_type;
```

The polygon_type is basically a triangle with three vertices, a, b, and c.

```
// The object type
typedef struct {
   char name[20];

   int vertices_qty;
    int polygons_qty;

    vertex_type vertex[MAX_VERTICES];
    polygon_type polygon[MAX_POLYGONS];

} obj_type, *obj_type_ptr;
```

Name, holds the model name. This will be extracted from the file. Also we defined a collection of vertices and polygons which will be extracted from the file. Obj_type and *obj_type_ptr are the model and its pointer after loading them from the file to the memory.

CThreeMaxLoader contains the following function which responsible for loading the 3DS file:

```
static char CThreeMaxLoader::Load3DS (obj_type_ptr p_object, char *p_filename)
{
       int i; //Index variable

       FILE *l_file; //File pointer

       unsigned short l_chunk_id; //Chunk identifier
       unsigned int l_chunk_lenght; //Chunk lenght

       unsigned char l_char; //Char variable
       unsigned short l_qty; //Number of elements in each chunk

       unsigned short l_face_flags; //Flag that stores some face information

       if ((l_file=fopen (p_filename, "rb"))== NULL) return 0; //Open the file

       while (ftell (l_file)< filelength (fileno (l_file)))        {

              fread (&l_chunk_id, 2, 1, l_file); //Read the chunk header
              //printf("ChunkID: %x\n",l_chunk_id);
              fread (&l_chunk_lenght, 4, 1, l_file); //Read the lenght of the chunk
              //printf("ChunkLenght: %x\n",l_chunk_lenght);

              switch (l_chunk_id)
              {
                     //----------------- MAIN3DS -----------------
                     // Description: Main chunk, contains all the other chunks
                     // Chunk ID: 4d4d
```

```
// Chunk Lenght: 0 + sub chunks
//------------------------------------------
case 0x4d4d:
break;

//----------------- EDIT3DS -----------------
// Description: 3D Editor chunk, objects layout info
// Chunk ID: 3d3d (hex)
// Chunk Lenght: 0 + sub chunks
//------------------------------------------
case 0x3d3d:
break;

//--------------- EDIT_OBJECT ---------------
// Description: Object block, info for each object
// Chunk ID: 4000 (hex)
// Chunk Lenght: len(object name) + sub chunks
//------------------------------------------
case 0x4000:
       i=0;
       do
       {
              fread (&l_char, 1, 1, l_file);
              p_object->name[i]=l_char;
              i++;
       }while(l_char != '\0' && i<20);
break;

//--------------- OBJ_TRIMESH ---------------
// Description: Triangular mesh, chunks for 3d mesh info
// Chunk ID: 4100 (hex)
// Chunk Lenght: 0 + sub chunks
//------------------------------------------
case 0x4100:
break;

//--------------- TRI_VERTEXL ---------------
// Description: Vertices list
// Chunk ID: 4110 (hex)
// Chunk Lenght: 1 x unsigned short (number of vertices)
//             + 3 x float x (number of vertices)
//             + sub chunks
//------------------------------------------
case 0x4110:
       fread (&l_qty, sizeof (unsigned short), 1, l_file);
       p_object->vertices_qty = l_qty;
       //printf("Number of vertices: %d\n",l_qty);
       for (i=0; i<l_qty; i++)
       {
```

```
                fread   (&p_object->vertex[i].x,   sizeof(float),   1,
                l_file);
                //printf("Vertices    list    x:    %f\n",p_object-
                >vertex[i].x);

                fread   (&p_object->vertex[i].y,   sizeof(float),   1,
                l_file);
                //printf("Vertices    list    y:    %f\n",p_object-
                >vertex[i].y);

                fread   (&p_object->vertex[i].z,   sizeof(float),   1,
                l_file);
                //printf("Vertices    list    z:    %f\n",p_object-
                >vertex[i].z);

                //Insert into the database

        }
        break;

//--------------- TRI_FACEL1 ----------------
// Description: Polygons (faces) list
// Chunk ID: 4120 (hex)
// Chunk Lenght: 1 x unsigned short (number of polygons)
//             + 3 x unsigned short (polygon points) x (number of
polygons)
//              + sub chunks
//------------------------------------------
case 0x4120:
        fread (&l_qty, sizeof (unsigned short), 1, l_file);
        p_object->polygons_qty = l_qty;
        //printf("Number of polygons: %d\n",l_qty);
        for (i=0; i<l_qty; i++)
        {
                fread   (&p_object->polygon[i].a,   sizeof   (unsigned
                short), 1, l_file);
                //printf("Polygon    point    a:    %d\n",p_object-
                >polygon[i].a);
                fread   (&p_object->polygon[i].b,   sizeof   (unsigned
                short), 1, l_file);
                //printf("Polygon    point    b:    %d\n",p_object-
                >polygon[i].b);
                fread   (&p_object->polygon[i].c,   sizeof   (unsigned
                short), 1, l_file);
                //printf("Polygon    point    c:    %d\n",p_object-
                >polygon[i].c);
                fread  (&l_face_flags,  sizeof  (unsigned  short),  1,
                l_file);
                //printf("Face flags: %x\n",l_face_flags);
        }
        break;
```

```
                    //------------- TRI_MAPPINGCOORS ------------
                    // Description: Vertices list
                    // Chunk ID: 4140 (hex)
                    // Chunk Lenght: 1 x unsigned short (number of mapping points)
                    //                + 2 x float (mapping coordinates) x (number of
                    mapping points)
                    //             + sub chunks
                    //----------------------------------------
                    //----------- Skip unknow chunks ------------
                    //We need to skip all the chunks that currently we don't use
                    //We use the chunk lenght information to set the file pointer
                    //to the same level next chunk
                    //----------------------------------------
                    default:
                            fseek(l_file, l_chunk_lenght-6, SEEK_CUR);
            }
      }

      fclose (l_file); // Closes the file stream

      return (1); // Returns ok
}
```

The function is a static member of CThreeMaxLoader class, so it can be called directly without instantiating an object from the class.
The function simply loops through all the available chunks and check for the type of chunks needed. Then it saves the vertices and polygons in the object referred by an `obj_type_ptr`.

7.5 Loading the object to OpenGL

In OpenGL we have used the function in the previous section to load a file named “chesspawn.3ds” to OpenGL. First we defined rotating variables which will be used to rotate the object the variables are

```
float rotation_x=0; float rotation_x_increment=0.06f;
float rotation_y=0; float rotation_y_increment=0.1f;
float rotation_z=0; float rotation_z_increment=0.03f;
```

Then we defined a variable of type obj_type to hold the vertices and polygon of our vertices

```
obj_type object;
```

In the init() function we called the Load3DS function to load the file “chesspawn.3ds” to the object variables. Here is the full implementation of the init() function

```
void init()
```

```
{
    glClearColor(0.0, 0.0, 0.0, 0.0); // This clear the background color to black
    glShadeModel(GL_SMOOTH); // Type of shading for the polygons

     // Projection transformation
    glMatrixMode(GL_PROJECTION); // Specifies which matrix stack is the target for
    matrix operations
    glLoadIdentity(); // We initialize the projection matrix as identity

    gluPerspective(45.0f,(GLfloat)screen_width/(GLfloat)screen_height,10.0f,10000.0
    f);

    glEnable(GL_DEPTH_TEST); // We enable the depth test (also called z buffer)
    glPolygonMode (GL_FRONT_AND_BACK, GL_LINE);

    CThreeMaxLoader::Load3DS(&object,"chesspawn.3ds");

}
```

Notice how we called the function Load3DS directly without taking an instance of the class CThreeMaxLoader. This is because Load3DS is defined as static member function.

We have loaded successfully the model from the file to the "object" variable. We need to go through all the vertices in the "object" variable to draw the polygons and the entire model. This will be done in the render function, see the following code segment

```
void render()
{
    int l_index;

    glClear(GL_COLOR_BUFFER_BIT |  GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW); // Modeling transformation
    glLoadIdentity();

    glTranslatef(0.0,0.0,-500.0);
    glColor3d(1,1,0);
     glRotatef(rotation_x,1.0,0.0,0.0); // Rotations of the object (the model matrix
    is multiplied by the rotation matrices)
     glRotatef(rotation_y,0.0,1.0,0.0);
     glRotatef(rotation_z,0.0,0.0,1.0);

           // TODO: Add your message handler code here
    rotation_x = rotation_x + rotation_x_increment;
     rotation_y = rotation_y + rotation_y_increment;
     rotation_z = rotation_z + rotation_z_increment;

     if (rotation_x > 359) rotation_x = 0;
     if (rotation_y > 359) rotation_y = 0;
     if (rotation_z > 359) rotation_z = 0;
```

```
    glBegin(GL_TRIANGLES); // glBegin and glEnd delimit the vertices that define a
    primitive (in our case triangles)
     for (l_index=0;l_index<object.polygons_qty;l_index++)
     {
         //----------------- FIRST VERTEX -----------------
         // Coordinates of the first vertex
         glVertex3f( object.vertex[ object.polygon[l_index].a ].x,
                     object.vertex[ object.polygon[l_index].a ].y,
                  object.vertex[ object.polygon[l_index].a ].z); //Vertex definition

         //----------------- SECOND VERTEX -----------------
         // Coordinates of the second vertex
           //float x= object.vertex[ object.polygon[l_index].b ].x;

         glVertex3f( object.vertex[ object.polygon[l_index].b ].x,
                     object.vertex[ object.polygon[l_index].b ].y,
                     object.vertex[ object.polygon[l_index].b ].z);

         //----------------- THIRD VERTEX -----------------
         // Coordinates of the Third vertex
         glVertex3f( object.vertex[ object.polygon[l_index].c ].x,
                     object.vertex[ object.polygon[l_index].c ].y,
                     object.vertex[ object.polygon[l_index].c ].z);
     }
     glEnd();
    glutSwapBuffers();
}
```

First we rotated the model by using the rotation variables, it is self-explanatory. Then we loop through the entire model to get the detailed vertices and use the 3 coordinates to draw a Triangle. The loop will load the entire vertices; OpenGL will connect every consecutive three vertices together to form a triangle. This will be done until the entire model is finished. The resulting model is shown in Fig.7.2

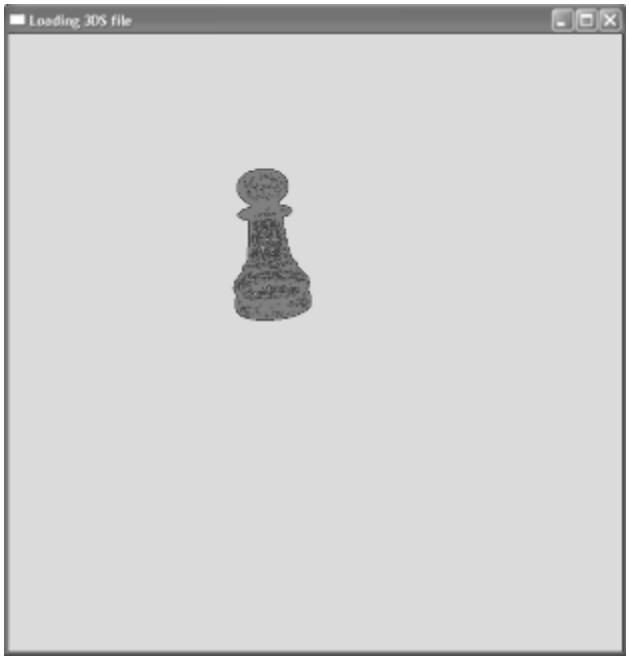


Fig. 7.2 Chesspawn object made by 3D Max and loaded to OpenGL

7.6 Conclusion

Since OpenGL is not used mainly in Modeling, so we introduced method for uploading models modeld by 3D Max software to our OpenGL window. The model consists of a collection of triangles rendered with lights and materils, the objective is to search for each triangle and get its vertices and texture vertices, then upload the model and align it with OpenGL coordinate system.

Chapter 8

Map Tiles

8.1 Introduction
8.2 Map tiles
8.3 Creating the map tiles
8.4 Loading the map tiles to OpenGL
8.5 Displaying the map tiles
8.6 Navigating through the map
8.7 Rendering the Minimap
8.8 Conclusion

8.1 Introduction

Have you ever played a strategy game like “Warcraft” or “Command & Conquer”?. Did you notice the map navigation? Do you know how did they (game developers) build such type of maps?.

These questions and more will be answered in this chapter.

We will explain the concept of map tiles Fig. 8.1, then we will build a map taken from image slices, finally we will show how to navigate through the map.

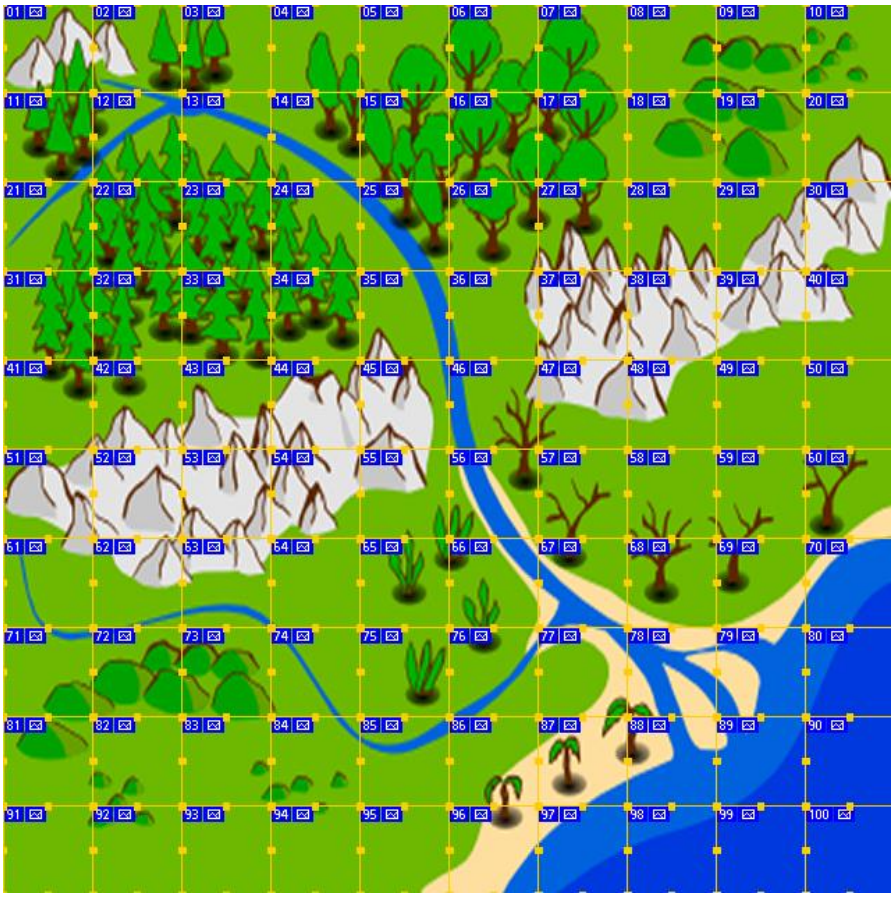


Fig.8.1 Map of size 1280x1280 sliced to 10x10 tiles 128x128 pixels each.

8.2 Map tiles

Strategy games are built on maps, without maps the game will not be interesting. The map can be loaded to OpenGL in tiles or slices. Take Fig.8.1 as an example we have an image for a map of size 1280x1280 sliced to 100 tiles. The map will be loaded to OpenGL by taking each tile and put it on the corresponding screen position. The map contains the following types of tiles

- Grass, example tiles 49, 50
- Rock, example tiles 38, 53.
- Water, example tiles 99, 100.

- Tree, example tiles 22, 23.

So, why we did divide the map into tiles, why don't we load it to OpenGL as one big tile?
We do that for many reasons, the most important reasons are that

1. We can minimize the memory space needed for storing the tiles, example see water tiles 99,100 they are typical, so we can remove one of them and use one tile instead of two also tile 90 can be removed. Another example look at tile number 49, 93 isn't they typical? We can remove one of them too. In Fig.9.1 the tile size is 128, it can be reduced to 32x32 or 64x64, the repeated tiles will be much more then 128x128 size.
2. We will use an obstruction map (we will do that in further chapters) , this map will be used to do the path finding algorithms. So for example, the avatar of the game will not walk on Rock tiles but it can walk on grass tiles.

8.3 Creating the map tiles

Suppose we have the map in Fig.8.1 and we need to slice it down to 400 tiles, We need to cut the image into 400 slices, 64x64 size each. This can be done with Adobe Photoshop or Adobe ImageReady.

Open Photoshop and then open the map image, use the slice tool to do one big slice holds the entire map, then divide this slice down into 20 horizontal slices 64 pixels each, this can be done by right clicking onto the map and select "Divide slice" from the context window. The output will be as shown in Fig.8.2. Then use the same procedure to divide the vertical slices.

To save the created slices, click File-> Save for web from the menu bar. Then use the JPG extension for the created slices. Finally define the path in which you will save your slices. The program will save 400 image files named sequentially from 1 to 400.

After the slices are saved into the hard disk with the .JPG extension, we need to change the image slices extension from .JPG to .BMP so it can be used in OpenGL.

To do that, we need to use third party software, which will convert the entire set of images together, instead of converting them one by one by paint brush or Photoshop. The software to do that named Imageicon and can be downloaded for free from the following website

http://www.bestfreewaredownload.com/freeware/t-free-imagicon-freeware-eignnsns.html

Drag and drop the files to the Imageicon and convert them to .BMP extension then they will be ready to be used in OpenGL.

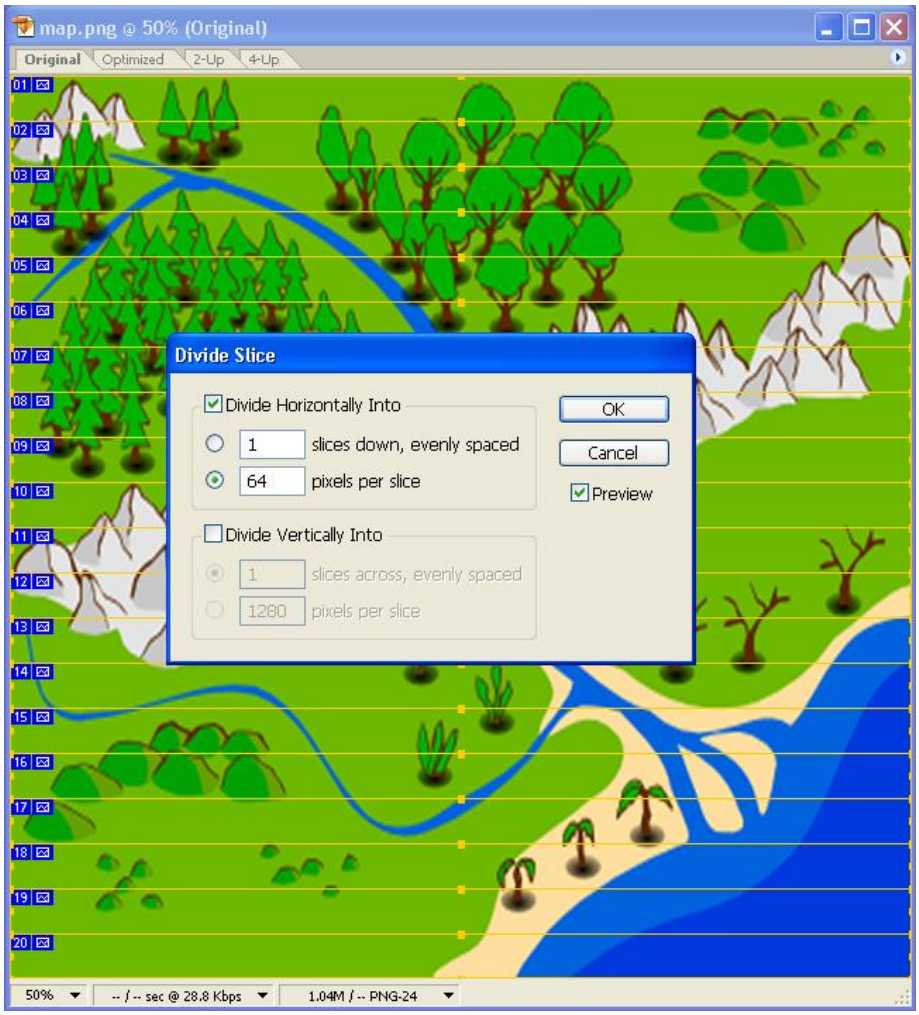


Fig.8.2 Slicing the map horizontally into 64 pixels for each slice

8.4 Loading the map tiles to OpenGL

Now, we will get into the core of implementing maps in OpenGL. In previous chapters, we explained the concept of textures. The map tiles will be loaded to OpenGL as textures, we will create gird of 400 squares 20x20 squares the width and height of the square will be 64 pixels, then we will cover each square with its corresponding tile. We will define a matrix of size 400 it will hold the entire collection of tiles

```
int g_map_width=20;
int g_map_height=20;

GLuint map_tiles[400];
```

Then we will use a function to load the tiles (all tiles are saved into a folder named map_slices located in the same program directory, the first file named map_1.bmp, the second file named

map_2.bmp, ….. the last one named map_400.bmp), the structure of this function is constructed as follow

```
void LoadAllSlices()
{

    for(int i=0;i<g_map_height*g_map_width;i++)
    {
           std::stringstream str_number;
           str_number<<i+1;
           string fileName="map_slices\\map_"+str_number.str()+".bmp";
                  map_tiles[i]=LoadTexture((char*)fileName.c_str());

    }
}
```

The function loops 20x20 or 400 times, and loads the tiles with the use of their names. We will convert the number i to string then we will concatenate the string with the remaining file name string to structure the entire file name. The LoadTexture function will take the current file name as an argument and load it to the proper place in the map_tiles matrix.

8.5 Displaying the map tiles

To display the map tiles, we need to define some concepts. The tiles will not be loaded entirely instead we will just render 10x10 tiles with 64 pixels wide for each tile, the OpenGL window size is 640x640, so it fits only 10x10 tiles and no need to load the entire 20x20 tiles. Fig.8.3 shows this concept.

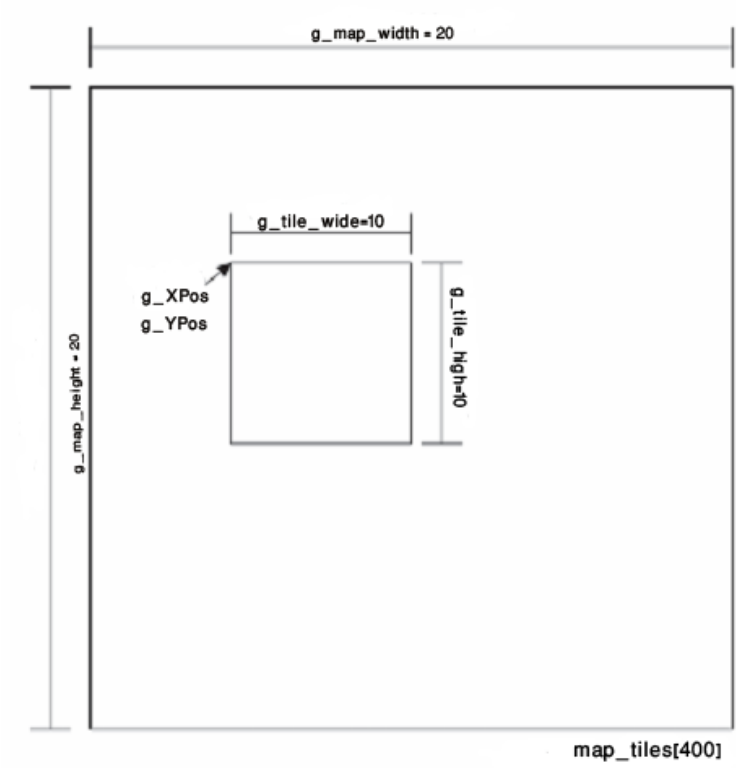


Fig.8.3 Display only 10x10 tiles out of 20x20 tiles.

To display a tile we use the following pseudo-code

```
int x,y;
// Display from top to bottom
for(y = 0; y < 10; y++)
{
        // Display from left to right
        for(x = 0; x < 10; x++) {
                // Your display function here
                DisplayTile(x, y);
        }
}
```

Since the tile array is not defined by x, y but defined by one dimension from 0 to 399. so how to get the corresponding element in the “map_tiles” array by using x, y? we can use the following formula

*DisplayTile(map_tiles[x+y*10])*

To display only 10x10 tiles we need to define some global variables

```
int g_map_width=20;
int g_map_height=20;

int g_tile_wide=10;
int g_tile_high=10;

int g_XPos=0;
int g_YPos=0;
```

The g_map_width, g_map_height holds the width and height of the entire tiles. The g_tile_wide and g_tile_high holds the size of the tiles to be displayed currently on the screen. The g_XPos and g_YPos holds the position of the first tile to be displayed. See Fig 8.3 for more details.
Now we understood the main concepts behind displaying a map, it is time for navigating the code. The code for displaying tiles is shown by the following code segment

```
void RenderMapSlices()
{
    for(int y=0;y<g_tile_high;y++)
    {
           for(int x=0;x<g_tile_wide;x++)
           {
                  glBindTexture(GL_TEXTURE_2D,
                  map_tiles[(x+g_XPos)+(y+g_YPos)*g_map_width]);

                  int pixel_x=x*tile_size;
                  int pixel_y=y*tile_size;

                  glBegin(GL_QUADS);
                         glTexCoord2f(0.0f,1.0f);
                         glVertex2f(pixel_x,pixel_y);
                         glTexCoord2f(1.0f,1.0f);
                         glVertex2f(pixel_x+tile_size,pixel_y);
                         glTexCoord2f(1.0f,0.0f);
                         glVertex2f(pixel_x+tile_size,pixel_y+tile_size);
                         glTexCoord2f(0.0f,0.0f);
                         glVertex2f(pixel_x,pixel_y+tile_size);
                  glEnd();
           }
    }
}
```

We loop through the tiles to be displayed. The texture will be bind at the position related to the current g_XPos and g_Ypos. The pixel position of the tile will be defined by multiply the current

position of tile by the tile size. Finally an imaginary triangle will be drawn and the texture will be mapped to it.

8.6 Navigating through the map

If the cursor hits the top of the map, it will be explored form the top until the top of the map is reached. The same procedure is applied to the right, left, and the bottom of the map. When the cursor hits the right of the map g_XPos will be increased, when it hits the left of the map g_XPos will be decreased, when it hits the top of the map, g_YPos will be decreased, when it hits the bottom of the map, g_YPos will be increased, we just need to make sure that g_XPos and g_YPos are within the boundary of the map. The following code does the above function

```
void MouseMotion(int x,int y)
{

    int mouse_tile_XPos=x/tile_size;
    int mouse_tile_YPos=y/tile_size;
    if(mouse_tile_XPos == g_tile_wide-1)
    {
           g_XPos++;
           if(g_XPos>=g_map_width-g_tile_wide)
                  g_XPos=g_map_width-g_tile_wide;
    }
    else if(mouse_tile_XPos ==0)
    {
           g_XPos--;
           if(g_XPos<0)
                  g_XPos=0;
    }
    else if(mouse_tile_YPos == g_tile_high-1)
    {
           g_YPos++;
           if(g_YPos >= g_map_height-g_tile_high)
                  g_YPos=g_map_height-g_tile_high;
    }
    else if(mouse_tile_YPos == 0)
    {
           g_YPos--;
           if(g_YPos <0)
                  g_YPos=0;
    }
}
```

The code takes the position of the mouse cursor and maps it to the tile scale. The code increases/decreased g_XPos, g_YPos, and then it makes sure that they are within the boundary of the displaying window.

8.7 Rendering the Minimap

The minimap is rendered in the strategy games to show the current position of the player on the map. Two ways to render the minimap

- Resizing each tile to a very small size, and displaying the minimap with the same concept shown in displaying the entire map.
- Use proxy code, this is done by assigning a color for each tile, for example green dots means a grass tile, black dots means a rock tile, and so on.

In this chapter, the first approach will be used. A rectangle showing the current position of the player will be drawn to show the current position of the player on the map.

The following function displays the minimap

```
void RenderMiniMap()
{
    int y_offset=650;
    int dot_size=8;

    for(int y=y_offset;y<y_offset+g_map_height;y++)
    {
           for(int x=0;x<g_map_width;x++)
           {
                  int pixel_x=x*dot_size;
                  int pixel_y= y_offset + (y-y_offset)*dot_size;
                  glBindTexture(GL_TEXTURE_2D,map_tiles[x+(y-
                  y_offset)*g_map_width]);

                  glBegin(GL_QUADS);
                         glTexCoord2f(0.0f,1.0f);
                         glVertex2f(pixel_x,pixel_y);
                         glTexCoord2f(1.0f,1.0f);
                         glVertex2f(pixel_x+dot_size,pixel_y);
                         glTexCoord2f(1.0f,0.0f);
                         glVertex2f(pixel_x+dot_size,pixel_y+dot_size);
                         glTexCoord2f(0.0f,0.0f);
                         glVertex2f(pixel_x,pixel_y+dot_size);
                  glEnd();
           }
    }
           //render the small rectangle to show your pos on the map
    int rect_pixel_x=g_XPos*dot_size;
    int rect_pixel_y=y_offset+g_YPos*dot_size;

    int rect_pixel_width=g_tile_wide*dot_size;
    int rect_pixel_height=g_tile_high*dot_size;
    glPolygonMode (GL_FRONT_AND_BACK, GL_LINE);
    glBegin(GL_QUADS);
```

```
                        glVertex2f(rect_pixel_x,rect_pixel_y);
                        glVertex2f(rect_pixel_x+rect_pixel_width,rect_pixel_y);
    glVertex2f(rect_pixel_x+rect_pixel_width,rect_pixel_y+rect_pixel_height);
                        glVertex2f(rect_pixel_x,rect_pixel_y+rect_pixel_height);
    glEnd();
}
```

Two new variables are used, y_offset is the y position of the map. X offset will be zero, so the map position will be to the bottom-left of the map.
The dot_size=8 is the size of the tile after resizing. This means every tile is resized from 64 pixels to 8.

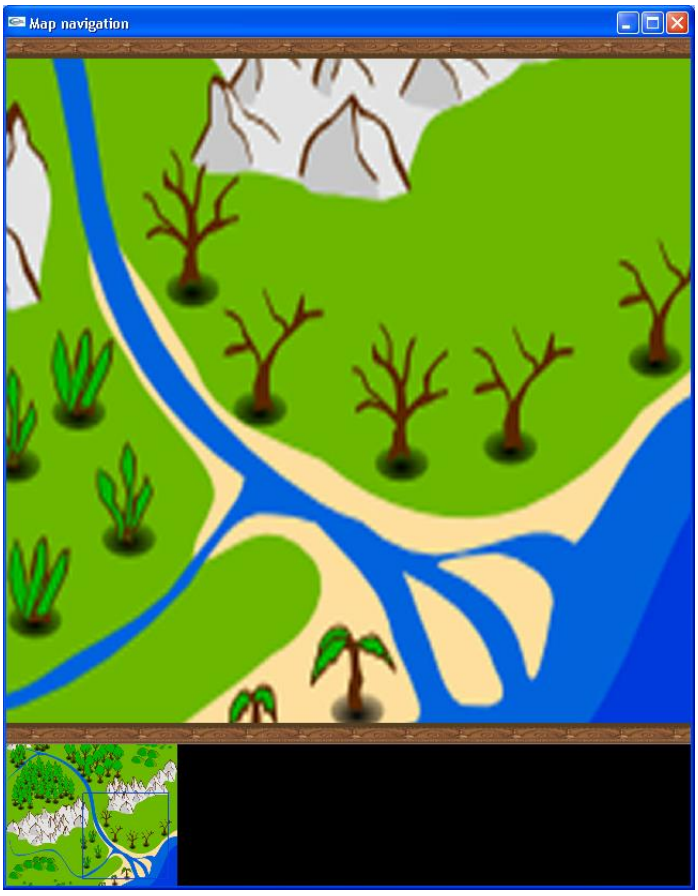


Fig.8.4 The map rendered with a small map in the bottom-left corner.

We loop through the tiles as explained before, and draw the tiles in the specified x, y positions. Finally a rectangle will be drawn to show the current position of the player in the game. The final result of our program should be as shown in Fig. 8.4

8.8 Conclusion

In this chapter the concept of Mapt tiles in games is explained. The idea is make a grid of tiles, the tile can be a small image. Tiles can be repeated to represent wide ares of the same shape, for example field grass, walls and floors.

Chapter 9

Path Finding

9.1 Introduction
9.2 A* Algorithm
9.3 The program Structure
9.4 The CPath class
9.5 Using CPath classM
9.6 Conclusion

9.1 Introduction

In strategy games, the avatar is moving from a place to another, without hitting the wall or walk through a mountain. How this is done? In computer science there is a branch of computer science called “Obstacle avoidance” this branch studies how the character can move on the map and avoid obstacle while it is walking. In this chapter we will show how to do the best obstacle avoidance practice by examining one of the most important path finding algorithms, namely A*(A Star) algorithm. Its importance came from that it not only find the path but it finds the best path. It is extensively used in strategy games.

9.2 A* Algorithm

Suppose we have a map and an object, we want to move this object from its current position to the target position. The intuitive way is to find a straight line connects the object to the needed target. But what happen if the line passes through an obstacle. In real world this is not permitted, since the object can’t walk through walls for example. See Fig 9.1

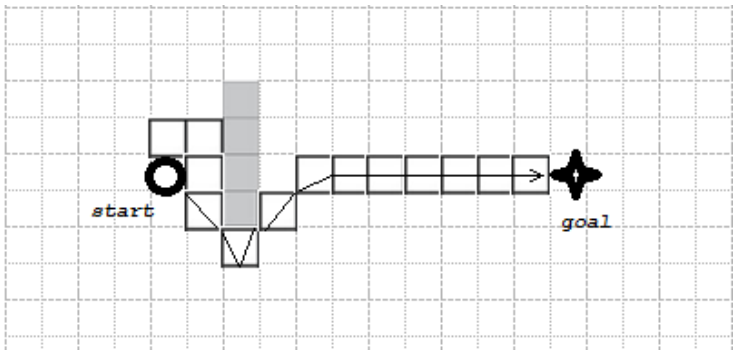


Fig.9.1 Obstacle avoidance in games

A-star solves this problem by examining the neighbor nodes, and finds the best one suitable to be the next step to the target. A start uses two types of lists; the first list is called the Open List which contains all the neighboring nodes to the current node, we call this current node the Parent node of all nodes in its neighborhood. The second list is the Closed list which contains the current node. The node structure can be defined as

```
 struct NODE { int StartCost; int TotalCost; int X,Y; int ParentX,ParentY; //parent x,y };
```

Where start cost is the cost from the current node to get back to the first node, it calculates how hard the object needs to walk to get to the original position. The total cost can be calculated as the following

Total cost=Base cost + Start Cost + Goal Cost

The base cost is the terrain cost for each node, for example if the node is a wall it will take cost higher than the grass. The grass can take higher base cost than road nodes. In our program we will used two types of base cost 0 for grass nodes and 1 for any other node. So the object can only walk on the grass nodes.

The Goal cost calculates how much distance is needed to reach the target or the goal. If the object is at x1, y1 and the goal is at xg, yg the goal cost will be calculated as

Goal cost = |xg – x1| + |yg – y1|

Fig 9.2 shows the costs and the total cost for every open node in the Open List

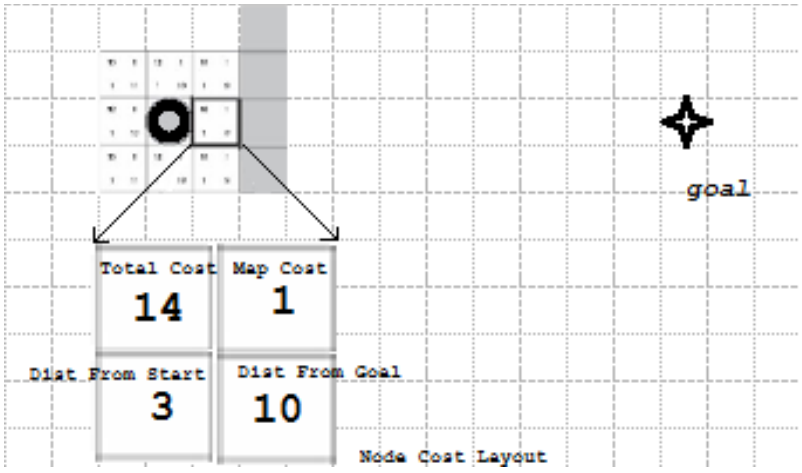


Fig.9.2The costs and the total cost for every open node in the Open List

The AStar algorithm contains the following steps

Step 1: Add the start node to the Closed List.

Step 2: Find all the neighboring nodes to the node in step 1. Add them all to the Open List

Step 3: Find the minimum node with the lowest total cost and add it to the closed list.

Step 4: Remove the node with the lowest total cost from the Open List.

After building the Open List and the Closed List (reach the goal) we back track them to find the path. The path can be examined by examining the goal node's parent and then examining the parent of the goal node's parent and so on until we reach the start node. This way we will get the right path.

9.3 The program Structure

The program contains a map, the map is reconstructed by 2D tiles the main tiles of the map are

Grass tiles, this tile can be easy for the object to walk on.

Tree tile, an obstacle

Stones tile, an obstacle

Bush tile, also an obstacle.

In addition to the start tile (rabbit image), goal tile (red pin) and the track tile (feet track) The map is built by using the above tiles, no more. We constructed a 20 by 20 tiles map as shown in Fig. 9.3

We have reconstructed the tile map as 2D array; we have shown in previous lessons how to reconstruct the map with 1D array. We have changed little bit to the 2D array to show the flexibility of having more than one solution for the same problem. We defined a Node structure as shown previously. Also we defined a CPath class in which we create the path needed for the rabbit to move from its current position to the goal. In our article we will concentrate only on making the path, we will leave the construction of the map as a practice.

9.4 The CPath class

The CPath class member functions are shown in the following code

```
class CPath
{
 private:
  vector &lt NODE >
  OpenList,ClosedList;
  int obstruction[20][20];
  int start_x,start_y;
  int goal_x,goal_y;
 public:
  CPath();
  virtual ~CPath();
      SetObstructionMatrix(unsigned  int  map[20][20],unsigned  int  ground,unsigned
    int start,unsigned int goal);
  InsertIntoClosedList(int total_cost,int start_cost,int x, int y);
  InsertIntoOpenList(int x,int y);
  vector &lt NODE >
  DeleteElement(vector &lt NODE > nodesList,int x,int y);
  bool IsElementExits(int x,int y);
  NODE MinNode(vector &lt NODE > );
  Create();
  ClearAll();
  vector &lt NODE >
  BackTrack();
  NODE GetNodeAt(int x,int y);
 };
```

Fig. 9.3 the costs and the total cost for every open node in the Open List

Two lists are defined, the OpenList and the ClosedList, we defined them as c++ vectors. You need to understand what vectors are so you can continue with this article. The obstruction matrix is needed to define the places in which the rabbit can't walk. It is reconstructed inside the following function

```
CPath::SetObstructionMatrix(unsigned int map[20][20],unsigned int grnd,unsigned int
strt,unsigned int gol)
{
 //we set the ground tiles to be zero
 //we also set the tree,bushes, stones tiles to be 1
 //the start is set to 8 and goal is set to 9
 for(int i=0;i < 20;i++)
 {
   for(int j=0;j < 20;j++)
   {
    if(map[i][j]==grnd) obstruction[i][j]=0;
    else if(map[i][j]==strt)
    {
     obstruction[i][j]=8;
     start_x=i;
```

```
     start_y=j;
    }
    else if(map[i][j]==gol)
    {
     obstruction[i][j]=9;
     goal_x=i;
     goal_y=j;
  }
  else obstruction[i][j]=1;
 }
}
}
```

The function takes the map_tiles and search for the start and goal, it sets a specified number corresponding to each type of tiles. It sets 0 to the grass or ground tiles, 8 to the start tile, 9 to the goal tile, and 1 to any else tile. The InsertIntoOpenList function inserts a node into the Open List the implementation of this function is shown in the following code

```
CPath::InsertIntoOpenList(int x, int y)
{
 //node must be inside the boundary of the map
 if(x < 20 && y < 20 && ClosedList.size() > 0 && !IsElementExits(x,y))
 {
  NODE tempNode;
  //the tile must be ground tile, so it will be passable
  if(obstruction[x][y]==0)
  {
   int base_cost,cost_to_start,cost_to_goal;
   base_cost=obstruction[x][y];
   cost_to_goal=abs(x - goal_x)+abs(y-goal_y);
   cost_to_start= ClosedList.back().StartCost + base_cost;
   tempNode.StartCost=cost_to_start;
   tempNode.TotalCost=base_cost+cost_to_goal+cost_to_start;
   tempNode.X=x;
   tempNode.Y=y;
   tempNode.ParentX=ClosedList.back().X;
   tempNode.ParentY=ClosedList.back().Y;
   OpenList.push_back(tempNode);
  }
  else if(obstruction[x][y]==9)
  {
    tempNode.StartCost=0;
    tempNode.TotalCost=0;
    tempNode.X=x;
    tempNode.Y=y;
    tempNode.ParentX=ClosedList.back().X;
    tempNode.ParentY=ClosedList.back().Y;
    OpenList.push_back(tempNode);
   }
  }
```

```
}
```

It calculates the node's info and adds the node to the Open List, First it calculates the base cost which is taken from the obstruction map, the cost to goal and cost to start will be added to the base cost to construct the total cost. The parent node of the nodes in the Open List will be the last node in the closed list, which is presented as ClosedList.back();. The DeleteElement function deletes a node at position x, y from the given list. The MinNode function iterate through the list to get the node with the lowest total cost. Here is the implementation of the MinNode function

```
NODE CPath::MinNode(vector &lt NODE > nodesList)
{
 NODE minNode;
 if(nodesList.size() > 0)
 {
  minNode=nodesList[0];
  for(int i=1;i < nodesList.size();i++)
  {
   if(nodesList[i].TotalCost < minNode.TotalCost) minNode=nodesList[i];
  }
  return minNode;
 }
 else
 {
  //return an error
  minNode.StartCost=-1;
  minNode.TotalCost=-1;
  minNode.X=0;
  minNode.Y=0;
  return minNode;
 }
}
```

If the list is empty it will return an empty node to show that there is an error. The Create() function is to create a path by using the A-Star algorithm here is the implementation of it

```
CPath::Create()
{
 //Clear previous lists
 ClearAll();
 //add the start node to the closed list
 InsertIntoClosedList(0,0,start_x,start_y);
 int next_x=start_x;
 int next_y=start_y;
 while(obstruction[next_x][next_y] != 9 && next_x < 20 && next_x > =0 && next_y <
20 && next_y > =0)
 {
  //take all the neighboring passable nodes and add them to the OpenList
  InsertIntoOpenList(next_x+1,next_y);
  InsertIntoOpenList(next_x+1,next_y+1);
```

```
  InsertIntoOpenList(next_x,next_y+1);
  InsertIntoOpenList(next_x-1,next_y+1);
  InsertIntoOpenList(next_x-1,next_y);
  InsertIntoOpenList(next_x-1,next_y-1);
  InsertIntoOpenList(next_x,next_y-1);
  InsertIntoOpenList(next_x+1,next_y-1);
  //find the min node
  if(MinNode(OpenList).StartCost != -1)
  {
   NODE minNode=MinNode(OpenList);
   ClosedList.push_back(minNode);
   next_x=minNode.X;
   next_y=minNode.Y;
   //delete the minimum node from the open list
   OpenList=DeleteElement(OpenList,next_x,next_y);
   }
 }
}
```

The two lists are cleared, next

1- The first node is added to the Closed List, the first node is the place which the rabbit stands on.

2- The while loop iterate until it finds the goal node which is known by 9. Then it goes through all the passable neighboring nodes and adds them to the Open List.

3- The min node in the Open List is added to the Closed List, and then the next node will be the min node so we set the next_x to be min node's x and next_y to be min node's y.

4- Last step, the min node is deleted from the Open List.

Now the Closed list contains the path elements, In order to get the right path we need to back track the path. This is done by using the function BackTrack it has the following code

```
vector &lt NODE >
CPath::BackTrack()
{
 vector &lt
 NODE > pathL;
 int k; k=0;
 for(int i=ClosedList.size()-1;i > -1;i--)
 {
  NODE tempNode;
  tempNode=GetNodeAt(ClosedList[i].ParentX,ClosedList[i].ParentY);
  if(tempNode.TotalCost != -1)
   pathL.push_back(tempNode);
  }
  return pathL;
}
```

The function iterates through the closed list nodes from its end back to its beginning. It gets the parent node of the current node.

9.5 Using CPath classM

The CPath class will be used in our program with mouse clicks. So the left mouse click will set the start position or the position of the rabbit and the right click will set the goal and create the path.

```
vector &lt
NODE > pathL;
CPath pathCreator;
void Mouse(int button, int state,int x,int y)
{
 int mouse_tile_XPos=x/tile_size+g_XPos;
 int mouse_tile_YPos=y/tile_size+g_YPos;
 switch (button)
 {
   case GLUT_LEFT_BUTTON:
    if (state == GLUT_DOWN)
    {
     //left clicked
     ClearTrack(true);
     map_tiles[mouse_tile_XPos][mouse_tile_YPos]=start;
     }
    break;
   case GLUT_RIGHT_BUTTON:
    if (state == GLUT_DOWN )
    {
      //right clicked
      ClearTrack(false);
      //set the goal position
      map_tiles[mouse_tile_XPos][mouse_tile_YPos]=goal;
      //create the path
      pathCreator.SetObstructionMatrix(map_tiles,ground,start,goal);
      pathCreator.Create();
      pathL=pathCreator.BackTrack();
      //set the path to the map
      for(int i=0;i < pathL.size()-1;i++)
      {
        map_tiles[pathL[i].X][pathL[i].Y]=track;
      }
      map_tiles[pathL[i].X][pathL[i].Y]=start;
      //clean the path list
      pathL.erase(pathL.begin(),pathL.begin()+pathL.size());
      }
      break;
      default:
```

```
            break;
        }
    }
```

Two variables have been defined, pathL, and pathCreator. pathL holds the path nodes. And pathCreator is an instance of the CPath list, which will create the path by AStar algorithm.

In the left click GLUT_LEFT_BUTTON event the start node will be set to the click location of the mouse on the map. In the right click GLUT_RIGHT_BUTTON event, goal node is set to the goal position which is the current mouse click location on the map. The obstruction matrix is created and then the path is created and back tracked. The path is “printed” on the map by looping through all nodes in the path and set them on the map.

9.6 Conclusion

Obstacle avoidance is important in strategy games. A* Algorithm is introduced with an example of a Rabbit avoids obstals while walking.

Chapter 10

Developing simple 2D game (Ball with Bat)

10.1 Introduction

Game development is really interesting work, actually when you look at a computer game and the AI implemented in it and its graphics and complexity, you then feel it is impossible to build such type of games. But if you know that all these games are depending on graphics libraries which made developing games very easy task, you will be interested in designing and developing games. We are going to prove that here by developing a simple 2D game depending on OpenGL library, we will introduce the concept of scene rendering and object collision, and also we will show how to write a text on the screen to get the score of the player. Finally we will show how to use the keyboard and mouse to interact with the game.

10.2 The Game Basics

When you develop a game you have to fully understand how the game is played, so you can implement its logic. Our game is a simple ball with bat game. The bat will be moved according to the movement of the mouse. And the ball will move randomly in the created window. When the ball hits the right, left, or top wall – we will refer to the window border as a wall - it will return back. When it hits the bottom wall it will not only return back but it will increase the score of the computer, but if the player can hold it by the bat, his score will be increased. Let us take a look at the interface of the game. The interface of the game is depicted in Fig. 10.1

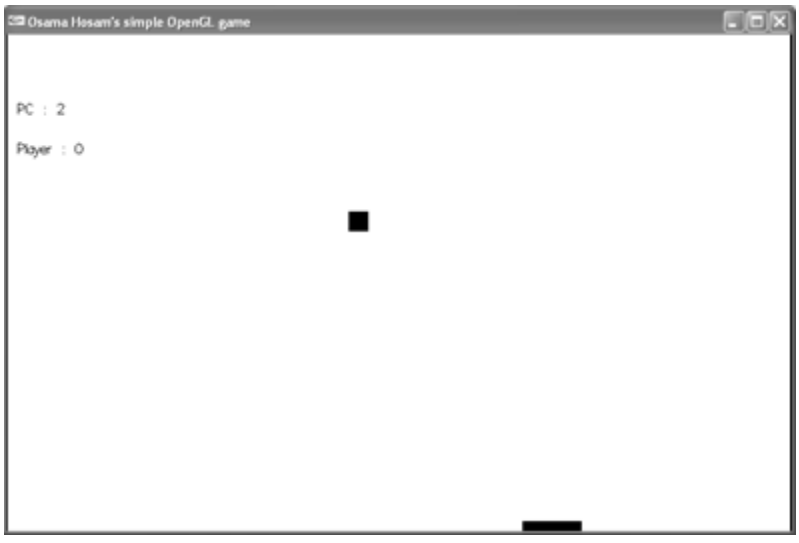


Fig.10.1 The interface of our game, the square in the middle represents the ball and the rectangle at the bottom represents the bat.

The window contains two counters, the first is PC: it is for the computer, it increases only if the ball hits the bottom border of the window (bottom wall). Player: it is a counter increases only if the ball hits your bat (the bat is represented by the rectangle at the bottom of the screen).

10.3 The Game Implementation

We will show step by step how to implement the game. First of all the concept of motion in OpenGL should be clear, the motion is done by drawing and looping. In every loop, the position of the object is adjusted so you fell it is moving. Motion in OpenGL has the same concept as the Cartoon films, every cycle the drawing is adjusted little bit and then all the images are displayed together which results in “moving” characters.

10.4 Drawing the window

The window in OpenGL is implemented simply by the following code segment

```
glutInit(&argc, argv);
glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB);
glutInitWindowSize (795, 500);
glutInitWindowPosition (0, 0);
glutCreateWindow ("Osama Hosam's simple OpenGL game");
```

First, the window initialization is done, and the color mode is set to RGB. Then a size of the window is defined as 795 for width and 500 for the height. Then the position of the window will be at 0,0 which is in the upper left corner of the computer screen. Finally a title is assigned to the window; this title will appear in the title bar or our window.

10.5 Drawing the Ball and the Bat and Displaying the Score Text

As we said the ball will be represented by a rectangle. For that we will define a new structure called RECTA, which is implemented by defining the left, top, right, and bottom coordinates. We are going to define three variables of this structure to be used as Ball, Wall, and Bat respectively

```
struct RECTA
{
        float left,top,right,bottom;
};
RECTA ball={100,100,120,120};
RECTA wall;
RECTA player_1 ={0,490,40,500};
```

Then we will use the DrawRectangle function to draw the RECTA instances on OpenGL window.

```
void DrawRectangle(RECTA  rect)
{
```

```
        glBegin(GL_QUADS);
        glVertex2f(rect.left,rect.bottom);      //Left - Bottom
        glVertex2f(rect.right,rect.bottom);
        glVertex2f(rect.right,rect.top);
        glVertex2f(rect.left,rect.top);
        glEnd();
}
```

The rectangle is drawn in OpenGL by using its corners coordinates. We start by the left-bottom corner and rotating counter-clockwise. glBegin(GL_QUADS) and glEnd() functions will encapsulate the rectangle coordinates. For the text, we need to write two words

PC: (the score of the computer)

Player: (The score of the player or the user). To do that we have implemented a function called drawText, it is shown in the following code segment

```
void drawText(char*string,int x,int y)
{
      char *c;
      glPushMatrix();
      glTranslatef(x, y,0);
      glScalef(0.1,-0.1,1);

      for (c=string; *c != '\0'; c++)
      {
           glutStrokeCharacter(GLUT_STROKE_ROMAN , *c);
      }
      glPopMatrix();

}
```

The function draws a string of type char* at position x,y. First we need to Push the matrix into the stack – so the next functions will not be affected by previous processing – then we translate the OpenGL cursor to position x, y then we scale the text to control its size (you are free to play with the values in glScalef function and see the effect of each value). Then we loop through the string (char* array) and draw every character separately by using the function glutStrokeCharacter. GLUT_STROKE_ROMAN is the name of the used font. Finally we pop the matrix from the stack so the previous settings are returned back.

10.6 Moving the Ball and the Bat

As we said before, the motion of the Ball (the square) is done by adjusting the square position in each loop. The Ball has a speed (this will be defined by a Timer Function together with the length of the Ball movement steps- in our program we call this steps “delta”). For the timer function it always contains a code that will be executed in a regular basis every specified

period of time (every 1 millisecond in our program). Our timer function is depicted in the following code segment

```
static float Xspeed=1,Yspeed=1;
static float delta=1;

void Timer(int v)
{

        ball.left+=Xspeed;
        ball.right+=Xspeed;
        ball.top+=Yspeed;
        ball.bottom+=Yspeed;

        glutTimerFunc(1,Timer,1);
}
```

We have defined two global variables Xspeed and Yspeed, which has the value of delta=1. This can be shown graphically in Fig. 10.2the value of delta will be changed (+1/-1) according to the collision of the ball with the walls. If the ball hits the right wall Xspeed will be –delta (this will make the ball return back), if the ball hits the left wall the Xspeed will be changed to delta and so on. For Yspeed also if the ball hits the top wall it will be equal delta, however if it hits the bottom wall or the bat, it will be –delta. The most important point here is that the Xspeed, Yspead will be used to increase the position of the ball, which was clearly done in the Timer function (refer to the above code segment). For changing the values of Xspeed and Yspeed, it is shown in the following code segment

```
if(Test_Ball_Wall(ball,wall)== FROM_RIGHT)
    Xspeed=-delta;

    if(Test_Ball_Wall(ball,wall)== FROM_LEFT)
    Xspeed=delta;

    if(Test_Ball_Wall(ball,wall)== FROM_TOP)
    Yspeed=delta;

    if(Test_Ball_Wall(ball,wall)== FROM_BOTTOM)
   {
          Yspeed=-delta;
          pcResult +=1;
   }
```

In the above code segment, the variable pcResult holds the counter of the computer score. Also we noticed the existence of a new function called Test_Ball_Wall(ball,wall). This function is used to detect the collision between the ball and the walls. The collision detection will be explained in the next section.

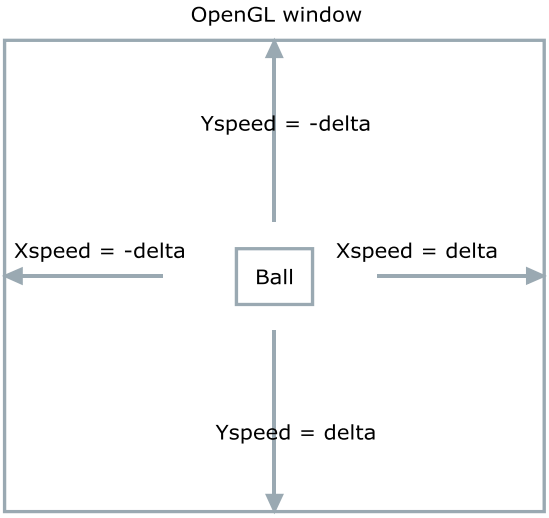


Fig. 10.2 The movement of the ball

Moving the bat will be done by moving the mouse cursor, when the mouse cursor is moved, only the x coordinate of the bat will be changed, the movement of the bat is shown in the following code segment

```
static mouse_x=0;
void MouseMotion(int x,int y)
{

    mouse_x=x;
}
```

We have defined a global variable called mouse_x. it will be used in drawing the bat when we render the whole scene. You can stop the game by pressing the Esc. key from the keyboard this is programmed as follow

```
void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
      case 27:
         exit(0);
         break;
   }
}
```

10.7 Ball Collision Detection

As we described in the previous section, when the ball hits the wall it will reflect back. The detection of the collision between the ball and wall (taking an example the right wall) is done simply by comparing the right coordinate of the wall with the ball's right coordinate, the collision will occur if they are the same or the ball's right is greater. For the bat/ball collision is done by comparing the top of the bat with the bottom of the ball. A collision occurs if they are the same or the balls bottom is greater and the x-coordinate lie inside the bat's x coordinates. The following code shows the above procedure

```
int Test_Ball_Wall(RECTA  ball , RECTA wall)
{
        if(ball.right >=wall.right)
        return FROM_RIGHT;
        if(ball.left <=wall.left)
        return FROM_LEFT;
        if(ball.top <=wall.top)
        return FROM_TOP;
        if(ball.bottom >=wall.bottom)
        return FROM_BOTTOM;

        else return 0 ;
}

bool Test_Ball_Player(RECTA ball,RECTA player)
{
   if(ball.bottom  >=  player.top  &&  ball.left>=  player.left  &&  ball.right
<=player.right )
        {
                 playerResult++;

                return true;
    }

        return false;
}
```

As we can expect playerResult is a global variable holds the score of the player.

10.8 Putting all Things Together

We are going to show how the above code will be used in rendering the whole scene of the game. First take a look at the Render function

```
void Render()
{
   glClear(GL_COLOR_BUFFER_BIT );
   glLoadIdentity();
```

```
    sprintf(string,"PC : %d ",pcResult);
    drawText(string,10,80);
    sprintf(string,"Player : %d ",playerResult);
    drawText(string,10,120);

    wall.left=wall.top=0;
    wall.right=WINDOW_WIDTH;
    wall.bottom=WINDOW_HEIGHT;

    DrawRectangle(ball);

     if(Test_Ball_Wall(ball,wall)== FROM_RIGHT)
     Xspeed=-delta;

     if(Test_Ball_Wall(ball,wall)== FROM_LEFT)
     Xspeed=delta;

     if(Test_Ball_Wall(ball,wall)== FROM_TOP)
     Yspeed=delta;

     if(Test_Ball_Wall(ball,wall)== FROM_BOTTOM)
    {
           Yspeed=-delta;
           pcResult +=1;
    }

    DrawRectangle(player_1);
     player_1.left=mouse_x-20;
     player_1.right=mouse_x+40;


     if(Test_Ball_Player(ball,player_1)==true)
             Yspeed=-delta;

    glutSwapBuffers();
}
```

As we mentioned before the Render function will run every OpenGL loop. So, first we need to clear and load the identity matrix to the window, and then we will use the drawText function to draw the text for the PC and Player scores. Then we define the walls coordinates (it will have the same size as the created OpenGL window) and then draw the ball. Then we detect the collision between the ball and the wall and between the ball and the bat, and accordingly increase the scores of the PC and the player. Finally we move the player according to the movement of the mouse.

10.9 Conlusion

In this chapter, basic concepts of implementing real 2D games are introduced. The game idea is to have a bouncing ball with bat to prevent it from virually hitting the bottom wall. The ball must be detected for collision with the basic four walls and the bat. The fonts are used to display text at the upper left corner scoring the points of both the player and the PC.

Chapter 11

Game Interface Design
Tetris Game

11.1 Introduction
11.2 Game analysis
11.3 Interface design
11.4 Game States
11.5 The Paint Class
11.6 Conclusion

11.1 Introduction

Tetris game is famous and easy to play. Also its rules are few, although it has a strong game play. We will cover the implementation of the game from scratch. We will start by game design, then game interface and then game code. The game has blocks that get down from the screen top with a specified speed. The player has to put the blocks over each other in a way not leaving holes between layers. If too much holes between layers the player will loose because the screen will be filled with blocks, if the player completed one row his score will be increased and he will have space to complete other rows. The game interface is shown in Fig11.1

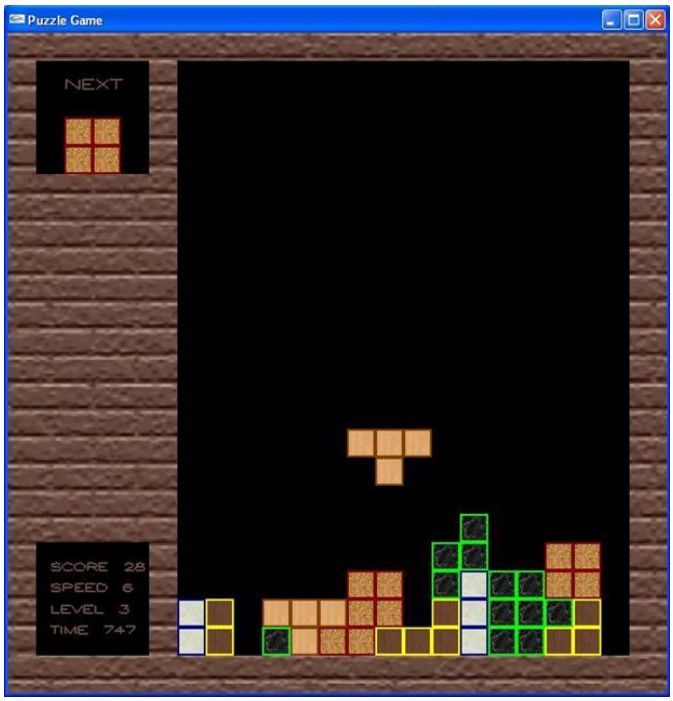


Fig.11.1 The interface of Tetris game

11.2 Game analysis

When you want to code a game, initially you have to understand every tiny detail in the game play. Imagine everything about the game, how the game will be played, the interface of the game, the game sound, etc. Generally speaking you have to do the brainstorm. This technique must be applied to small games, because if done well, it will be easily applied on large scale games. So let's do brainstorm for our game.

We have entities called block, every block has specified shape. Every block can move right, left or bottom, it can't move up. The movements of the blocks will be done by keyboard arrows. The left arrow will make the block move left, the right arrow will move the block to the right, the down arrow will move the block quickly to the bottom, the up arrow will rotate or flip the block.

The movements will be inside a border. We need a matrix to hold the border dimensions and hold the block collection data and positions; we called this matrix (Game_Matrix). The movements of the block will be limited to be inside that border. So we will move right until the block reach the last column in the Game_Matrix or collide with an existing block, the block also will move left until it reaches the column number 0 in the Game_Matrix or hit another block. The block will move down until it reaches the last row of the Game_Matrix or hit another block.

If one row of the matrix filled with block items, it will be removed form the matrix and all the above rows will shifted down one row, then the score will be increased.

The speed of the game will be controlled by OpenGL timers, the speed, level and elapsed time will be stored into another entity called Game_settings. See Fig.11.2 for more details.

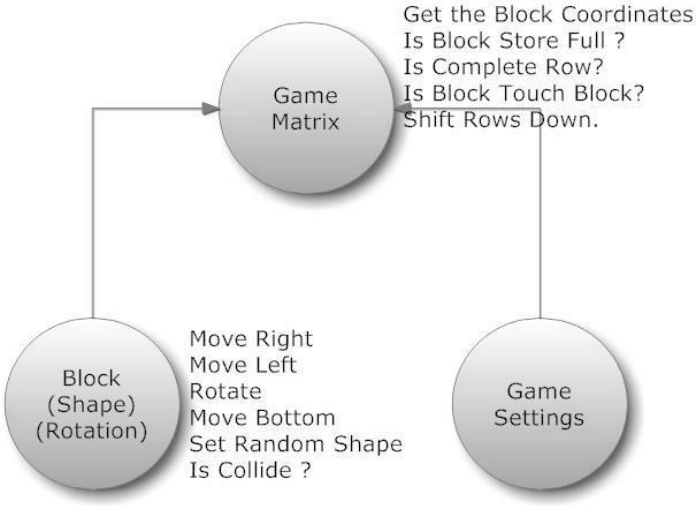


Fig.11.2 Game Entities

Fig.2 shows the three main entities of our game, The Block position will be changed with time and its position will be changed accordingly in the Game_Matrix.

The last thing we want to mention is that the game displays the next block shape which will come from the top of the screen after you settle down the current block. We will assign a small window for this purpose.

11.3 Interface design

First we need to create a main menu. The main menu contains, New Game, Settings, Top Score and Exit items, the main menu is shown in Fig.11.3. To simplify the game design we will just implement the "New Game" screen. The menu is implemented by assigning two types of images for each item, the first image without border and the second image is with border. When

the menu item is selected we assign the image with border to the menu item otherwise the image without border will be assigned.

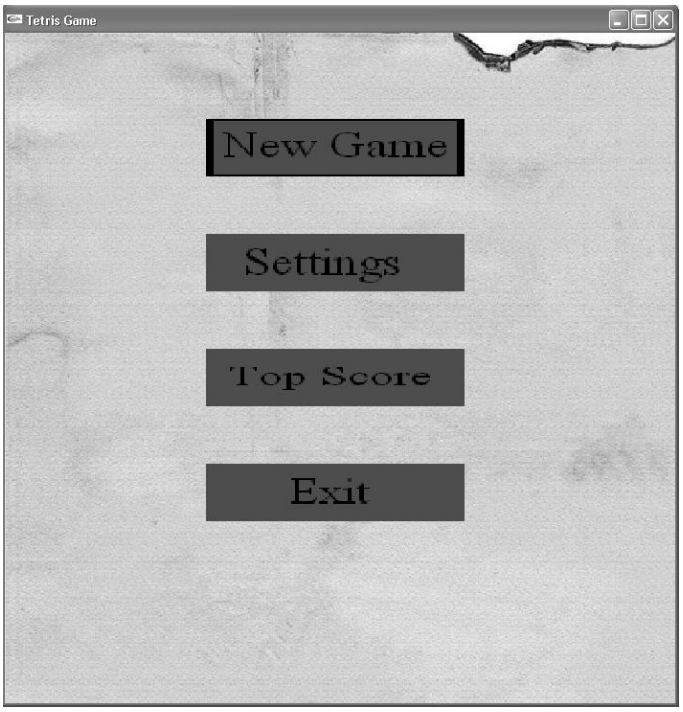


Fig.11.3 The main menu of the game.

When the new game menu item is selected by using the keyboard arrows and pressing enter, the game screen will be displayed. The game screen is 700x700 pixels, it will be divided into a grid of rectangles every rectangle is 30 pixels wide and 30 pixels high. Fig.11.4 shows this grid. Also the following code presents the default game settings

```
GameSettings()
{
	level=0;
	steps_time=300;
	length_unit=30;
	score=0;
	window_width=700;
	window_height=700;
	elapsed_time=0;
	setSpeed();
}
```

The rectangle in red representing the Game_Matrix which is 21 rows x 16 columns.

```
#define MATRIX_ROWS 21
```

```
#define MATRIX_COLS 16
```

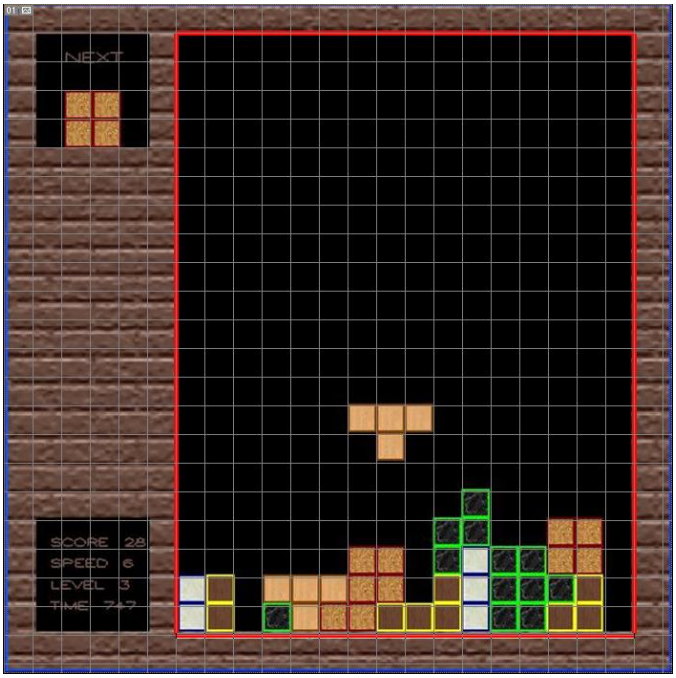


Fig.11.4 The interface of the tetris game.

The Game_Matrix entity contains tow C++ matrices (2 dimensional matrices)

```
int single_block[MATRIX_ROWS][MATRIX_COLS];
int block_store[MATRIX_ROWS][MATRIX_COLS];
```

- Single_Block matrix this matrix will hold the currently moving block position.
- Block_Store matrix which holds the entire collection of blocks

The idea is to combine the two matrices to form the overall matrix which will be displayed on the screen. I separated them to easily deal with both matrices, if it was one matrix managing it will be difficult.

We assigned numbers to each type of block,

- 1 is the T shape block,
- 2 is the L shape block.
- 3 is the Z shape block
- 4 is the I shape block
- 5 is the O shape block.

Every block will be represented by its corresponding number in the single block matrix and block store matrix. The place with no block will be represented by 0. Fig.11.5 shows both single block and block store matrices.

At any time, the pixel coordinates of every item can be get by multiply the length_unit variable of the GameSettings by the items row, column coordinates of the item in the Game_Matrix.

Item's pixel x = item's column * length_unit

Item's pixel y= item's row * length_unit.

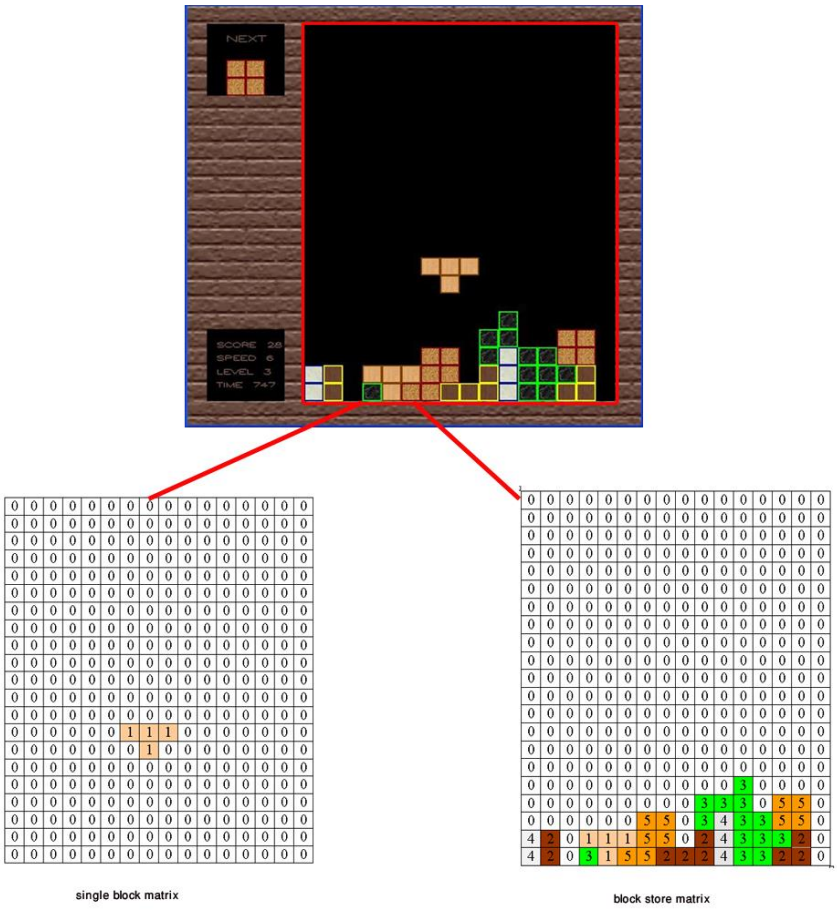


Fig. 11.5 The single block and block store matrices

11.4 Game States

The game state represents which state the game in. Which screen to be displayed? The main menu screen? The game main screen? This question is answered by the game state class. We have assigned the following values for the game state class:

- 1 refers to the main menu screen

- 2 refers to the game screen.
- 3 refers to the Settings screen.
- 4 refers to the Top Score screen.

We check to see which state we are in, and then we display the appropriate interface textures. The States class is shown by the following code segment

```
class States
{
private:
    int state;
public:
    States()
    {
           state=1;
    }
    void setState(int s)
    {
           state=s;
    }
    int getState()
    {
           return state;
    }
}current_state;
```

The default state of the game is 1 or the main menu. This value can be set and get by the function setState() and getState() respectively.

A question arises here. Why there is no global variable instead of creating a class for that variable? I prefer this way of programming to be easy in the future to update the game States class and add new features if any.

11.5 The Paint Class

We have implemented a single class responsible for all the activities of interface design. Let see the game class's member functions

```
class Paint
{
private:
    char string [100];
    int menu_number;
    int menu_items_count;
    GLuint texture_id[20];
public:
    Paint();
```

```
    void setNextMenuItem();
    void setPrevMenuItem();
    int getMenuItemPos();
    GLuint LoadTexture(char *FileName );
    void DrawBackgroundImage();
    void LoadAllTextures();
    void DrawRectangle(RECTANGLE rect);
    void drawText(char*string,int x,int y, float font_size);
    void printMessageEsc();
    void printMessageGameOver();
    void DrawTexturedRectangle(RECTANGLE rectangle,int item_number);
    void DrawGameMenu();
    void DrawGameSubwindows();

} paint_background;
```

I'll explain the class by the order of the execution of its member functions. First the class loads all textures by using LoadAllTextures() function which uses the function LoadTexture() to load a single file . The function takes the path of every image file on the hard disk and load it as a texture to the texture array textur_id[20]. Fig. 11.6 shows the texture file names and their corresponding positions in the texture_id matrix.

The functions setNextMenuItem and setPrevMenuItem change the current menu item in the menu screen. getMenuItemPos gets the current position of menu item. The implementation is as follow

```
Paint()
    {
           menu_number=1;
           menu_items_count=4;
    }
    void setNextMenuItem()
    {
           if(menu_number==menu_items_count)
                  menu_number =1;
           else
                  menu_number++;
    }
    void setPrevMenuItem()
    {
           if(menu_number==1)
                  menu_number=menu_items_count;
           else
                  menu_number--;
    }
    int getMenuItemPos()
    {
           return menu_number;
```

```
}
```

Paint() function is the constructor of our class and it sets the menu items to 4 menu items, and set the menu number to 1 means the default menu item is “New Game” menu item, so

- menu_number = 1 refers to the “New Game” menu item.
- menu_number=2 refers to the “Settings” item.
- menu_number=3 refers to the “Top Score” item.
- menu_number=4 refers to the “Exit” item.

When the menu_number is 1 and we need to get the previous menu item we set the menu_number to be 4, also when the menu_number is 4 and we want to get the next menu item we set the menu_number to be 1. So it acts like a circle relating the collection of menu items.

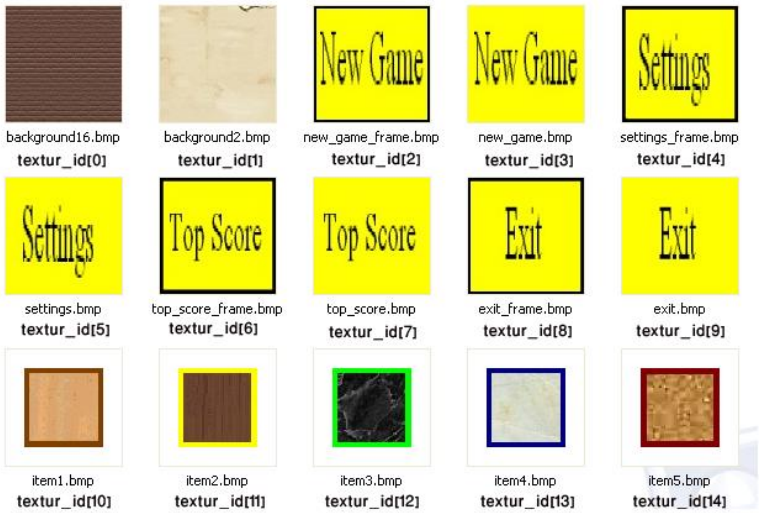


Fig. 11.6 The textures needed for Tetris game.

The function DrawTexturedRectangle takes a rectangle with its positions on the screen and cover it with a texture according to the item_number variables here is the implementation of the DrawTexturedRectangle function

```
void DrawTexturedRectangle(RECTANGLE rectangle,int item_number)
   {
          switch(item_number)
          {
          case 1:
                 if(menu_number == 1)
                        glBindTexture(GL_TEXTURE_2D,  texture_id[2]);
                 else
```

```
                glBindTexture(GL_TEXTURE_2D,  texture_id[3]);
        break;
case 2:
        if(menu_number == 2)
                glBindTexture(GL_TEXTURE_2D,  texture_id[4]);
        else
                glBindTexture(GL_TEXTURE_2D,  texture_id[5]);
        break;
case 3:
        if(menu_number == 3)
                glBindTexture(GL_TEXTURE_2D,  texture_id[6]);
        else
                glBindTexture(GL_TEXTURE_2D,  texture_id[7]);
        break;
case 4:
        if(menu_number == 4)
                glBindTexture(GL_TEXTURE_2D,  texture_id[8]);
        else
                glBindTexture(GL_TEXTURE_2D,  texture_id[9]);
        break;
case 5:
        glBindTexture(GL_TEXTURE_2D,  texture_id[10]);
        break;
case 6:
        glBindTexture(GL_TEXTURE_2D,  texture_id[11]);
        break;
case 7:
        glBindTexture(GL_TEXTURE_2D,  texture_id[12]);
        break;
case 8:
        glBindTexture(GL_TEXTURE_2D,  texture_id[13]);
        break;
case 9:
        glBindTexture(GL_TEXTURE_2D,  texture_id[14]);
        break;

default:
        glBindTexture(GL_TEXTURE_2D,  NULL);
        break;
}
//to draw a rectangle
glBegin(GL_QUADS);

glTexCoord2f(0.0f,0.0f);
glVertex2f(rectangle.left,rectangle.bottom );

glTexCoord2f(1.0f,0.0f);
glVertex2f(rectangle.right ,rectangle.bottom);
glTexCoord2f(1.0f,1.0f);
glVertex2f(rectangle.right,rectangle.top );
glTexCoord2f(0.0f,1.0f);
```

```
        glVertex2f(rectangle.left,rectangle.top);
        glEnd();
}
```

The function switch the item number, if it is menu item, it make sure that it is the currently selected item or not by checking the value of menu_number variable, if it is the currently selected item, it shows the corresponding image with border around it. The corresponding texture is bind to memory and then the rectangle is covered by it.

The DrawBackgroundImage() draws either the background image of the menu screen or the background image of the game screen.

The PrintMessageEsc() and PrintMessageGameOver() functions use the DrawText function to draw a text on the screen in a specified position.

The function DrawGameMenu() is straight forward and it uses the function DrawTexturedRectangle to draw the four items of the menu.

The function DrawGameSubwindows() is little complicated, but if you understood well the grid division shown before in Fig.11.4 it will be an easy task for you, let us see the implementation of this function

```
void DrawGameSubwindows()
{
        int unit=game_settings.getGameRectLength();
        // the rectangle which includes the next block shape
        RECTANGLE left_border_up_b={unit,unit,unit+4*unit,unit+4*unit};
        // the rectangle which include the score
        RECTANGLE left_border_down_b={unit,18*unit,unit+4*unit,18*unit+4*unit};
        //the recangle which include the game blocks
                    RECTANGLE
        game_screen={unit+4*unit+unit,unit,6*unit+16*unit,unit+4*unit+13*unit+4*u
        nit};

        //draw
        DrawRectangle(game_screen);
        DrawRectangle(left_border_up_b);
        DrawRectangle(left_border_down_b);

        //put text score
        glColor3f(1.0f,1.0f,1.0f);
        sprintf(string,"SCORE %d ",pcResult);
        drawText(string,unit+0.5*unit,18*unit+unit,1.5);

        //put the speed text
        sprintf(string,"SPEED %d ",game_settings.getSpeed());
        drawText(string,unit+0.5*unit,18*unit+(1.75)*unit,1.5);

        //put level text
        sprintf(string,"LEVEL %d ",game_settings.getLevel());
        drawText(string,unit+0.5*unit,18*unit+(2.5)*unit,1.5);
```

```
//put elpsed time
sprintf(string,"TIME %d ",game_settings.getElapsedTime());
drawText(string,unit+0.5*unit,18*unit+(3.25)*unit,1.5);


//put the shape text
drawText("NEXT",unit+unit,unit+unit,2);

//draw the next shape rectangles(block)
RECTANGLE first_rect;
RECTANGLE second_rect;
RECTANGLE third_rect;
RECTANGLE fourth_rect;

int next_shape=current_moving_block.getNextshape();
int hlf=unit/2;
switch(next_shape)
{
case 1:
	//T shape

	first_rect.left=unit+hlf;   first_rect.top=unit+2*unit;
	first_rect.right=2*unit+hlf;  first_rect.bottom=3*unit+unit;
	second_rect.left=2*unit+hlf;          second_rect.top=unit+2*unit;
	second_rect.right=3*unit+hlf; second_rect.bottom=3*unit+unit;
	third_rect.left=3*unit+hlf;            third_rect.top=unit+2*unit;
	third_rect.right=4*unit+hlf;  third_rect.bottom=3*unit+unit;
	fourth_rect.left=2*unit+hlf;fourth_rect.top=unit+3*unit;
	fourth_rect.right=3*unit+hlf; fourth_rect.bottom=4*unit+unit;
	break;
case 2:
	// L shape
                      first_rect.left=unit+hlf;
	first_rect.top=unit+2*unit;           first_rect.right=2*unit+hlf;
	first_rect.bottom=3*unit+unit;
	second_rect.left=2*unit+hlf;          second_rect.top=unit+2*unit;
	second_rect.right=3*unit+hlf; second_rect.bottom=3*unit+unit;
	third_rect.left=3*unit+hlf;            third_rect.top=unit+2*unit;
	third_rect.right=4*unit+hlf;  third_rect.bottom=3*unit+unit;
	fourth_rect.left=unit+hlf;            fourth_rect.top=unit+3*unit;
	fourth_rect.right=2*unit+hlf;  fourth_rect.bottom=4*unit+unit;

	break;
case 3:
	//Z shape
	first_rect.left=unit+hlf;   first_rect.top=unit+3*unit;
	first_rect.right=2*unit+hlf;  first_rect.bottom=4*unit+unit;
	second_rect.left=2*unit+hlf;          second_rect.top=unit+2*unit;
	second_rect.right=3*unit+hlf; second_rect.bottom=3*unit+unit;
```

```
            third_rect.left=3*unit+hlf;            third_rect.top=unit+2*unit;
            third_rect.right=4*unit+hlf;  third_rect.bottom=3*unit+unit;
            fourth_rect.left=2*unit+hlf;fourth_rect.top=unit+3*unit;
            fourth_rect.right=3*unit+hlf; fourth_rect.bottom=4*unit+unit;
            break;
      case 4:
            //I shape
                              first_rect.left=unit;
            first_rect.top=3*unit;                  first_rect.right=unit+unit;
            first_rect.bottom=3*unit+unit;
            second_rect.left=unit+unit;                 second_rect.top=3*unit;
            second_rect.right=unit+2*unit; second_rect.bottom=3*unit+unit;
            third_rect.left=unit+2*unit;                 third_rect.top=3*unit;
            third_rect.right=unit+3*unit;  third_rect.bottom=3*unit+unit;
            fourth_rect.left=unit+3*unit;fourth_rect.top=3*unit;
            fourth_rect.right=unit+4*unit; fourth_rect.bottom=3*unit+unit;

            break;
      case 5:
            //O shape
                              first_rect.left=unit+unit;
            first_rect.top=unit+3*unit;           first_rect.right=unit+2*unit;
            first_rect.bottom=4*unit+unit;
            second_rect.left=unit+unit;                 second_rect.top=3*unit;
            second_rect.right=unit+2*unit; second_rect.bottom=3*unit+unit;
            third_rect.left=unit+2*unit;                 third_rect.top=3*unit;
            third_rect.right=unit+3*unit;  third_rect.bottom=3*unit+unit;
            fourth_rect.left=unit+2*unit;fourth_rect.top=unit+3*unit;
            fourth_rect.right=unit+3*unit; fourth_rect.bottom=4*unit+unit;

            break;
      default:
            break;
      }

      DrawTexturedRectangle(first_rect,next_shape+4);
      DrawTexturedRectangle(second_rect,next_shape+4);
      DrawTexturedRectangle(third_rect,next_shape+4);
      DrawTexturedRectangle(fourth_rect,next_shape+4);
}
```

The left_border_up_b represents the rectangle in which the next shape will be drawn. It will be drawn one unit from top and one unit from left, with width = 4 units and height = 4 units. The unit = 30 pixels.

The Left_border_down_b represents the rectangle which holds the game data, the score, the level, the elapsed time, and the speed. The function Draw text will be used to draw the text needed on the screen.

The game_screen rectangle is the rectangle in which the game will be played and holds the collection of the blocks and the current moving block.

Four rectangles will be defined; first_rect, second_rect, third_rect, and fourth rectangle. These rectangles represent the four rectangles which are the elements of the next block to be displayed. Although this is tedious way to draw them on the screen by defining their coordinates rectangle by rectangle, but this way will not be used when it is time to draw the blocks in the game screen.

The DrawTexturedRectangle function is adopted to draw each block. We have passed the value next_shape+4 to skip the 4 elements of the main menu screen.

11.6 Conclusion

In this chapter we introduced an implementation of the Tetris game which is 2D game. The game interface design and main menu design is also introduced. Marices are used to store the status of the game. With an alteration of the matrix the corresponding interface element will be changed. In the next chapter we will continue explaining the Tetris game. We will implement the game logic.

Chapter 12

Game Logic
Tetris Game

12.1 Introduction

We have presented the interface development of the Tetris game. In this chapter we are going to introduce the core implementation of the game. First we will look at the game matrix and how to update the game with the new values updated in the game matrix.

12.2 The Overall Idea

We have the game block entity with the following properties

```
int row_pos;
int col_pos;
int rotation_pos;
int shape;
int next_shape;
bool is_halt_mode;
```

row position is the horizontal position of the block, col position is the vertical position of the block. is_halt_mode refers to the situation in which the user presses the Esc key in the keyword, accordingly the game must be halt until the user decides to continue or exit the game.

rotation_pos is the value of the rotation; every block has a rotation value, according to this rotation value the shape will not be changed but the orientation will be changed; it will be changed only if the user pressed the UP_ARROW from the keyboard, Fig.12.1 shows the concept of the block rotating. Shape refers to one of the list of shapes we stated before in the previous chapter, i.e number 1 for the T shape, 2 for the L shape, etc. The next_shape refers to the shape that will be the next and will be displayed in a small window at the top left corner of the game screen.

The above block properties will be taken to the Game_Matrix's single block matrix. The values in single block matrix will be changed according to block's properties (fields).

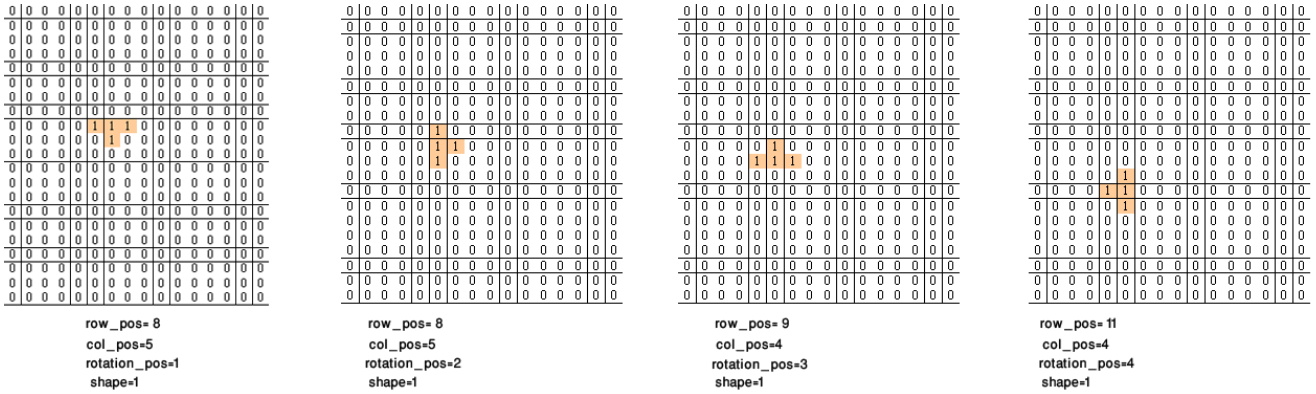


Fig.12.1 The single block matrix manipulation according to the block's fields.

The next step is to make a comparison between both single_block matrix and the block_store matrix. In this comparison the collision of the moving block with one of the blocks in the block store will be detected. also the collision between the moving block and one of the window sides will be detected. According to this detection the moving block will be restricted in its moving. If the moving block hits the bottom side or one block in the block store from the bottom it will be added directly to the block store matrix. The block store matrix will be detected to see if a complete row is exist, this will be done by examining the row to see if all the row values greater than zero, if so the row will be removed and all the rows above it will be shifted down one row.

12.3 The Moving Block Class

The following is the implementation of the block class

```
enum Sides{LEFT,RIGHT,TOP,BOTTOM};

class Block
{
private:
    int row_pos;
    int col_pos;
    int rotation_pos;
    int shape;
    int next_shape;
    bool is_halt_mode;
public:
    Block()
    {
           setNextShape();
           resetBlock();
    }
    int SetRandShape()
    {
           // 1:T, 2:L , 3:Z, 4:I 5:O
           int rand_shape=rand()%5; //output from 0 to 4
           rand_shape++;
           return rand_shape;
    }
```

The SetRandShape gets a random value from 1 to 5, This will be the next shape to be displayed. Also this random value will be the shape value of the next shape.

```
    void resetBlock()
    {
           row_pos=0;
           col_pos=7;
           is_halt_mode=false;
           rotation_pos=1;
```

```
        shape=next_shape;
        setNextShape();
}
```

In the above function we set the shape to be the next_shape, this means the next shape value (randome) will be the shape of the next shape to be displayed.

```
void setNextShape()
{
        next_shape=SetRandShape();
}
int getNextshape()
{
        return next_shape;
}
void MoveBottom(bool isDownArrowPressed)
{
        if(!isCollide(BOTTOM)&& !is_halt_mode)
        {
                row_pos++;
                //this way we will increase time
                //only if the keydown is not pressed
                if(!isDownArrowPressed)
                        game_settings.increaseElapsedTime();
        }
}
```

The MoveBottom function will move the block to bottom until it collides with the Bottom of the screen or one of the blocks in the block store matrix.

The time of the game will be increased only if the player didn't press the DOWN_ARROW from the keyboard otherwise the game time will not be increased, simply because pressing the DOWN_ARROW will make the block move directly from the current position down to the first block it hits in the block store.

```
void MoveRight()
{
        if(!isCollide(RIGHT)&& !is_halt_mode)
        {
                col_pos++;
                game_settings.increaseElapsedTime();
        }

}
void MoveLeft()
{
        if(!isCollide(LEFT)&& !is_halt_mode)
        {
                col_pos--;
                game_settings.increaseElapsedTime();
```

```
	}

}
```

MoveRight and MoveLeft move the block to right or left by increasing and decreasing the horizontal position of the block respectively.

```
void RotateShape()
{
	if(!is_halt_mode)
	{
		if(rotation_pos == 4)
			rotation_pos=1;
		else
			rotation_pos++;
	}
}
```

Rotate shape changes the orientation of the block, if the orientation is 4 it set back to 1.

```
void haltGame()
{
	is_halt_mode=true;
}
void continueGame()
{
	is_halt_mode=false;
}
bool isHaltMode()
{
	return is_halt_mode;
}
int getRow()
{
	return row_pos;
}
int getCol()
{
	return col_pos;
}
int getRotation()
{
	return rotation_pos;
}
int getShape()
{
	return shape;
}
```

```
    bool isCollide(Sides sd)
    {
           switch(sd)
           {
           case LEFT:
                  if(col_pos <= 0 )
                  {
                         return true;
                  }
                  break;
           case RIGHT:
                  if(col_pos >= MATRIX_COLS-1)
                  {
                         return true;
                  }
                  break;
           case BOTTOM:
                  if(row_pos >= MATRIX_ROWS-1)
                  {
                         return true;
                  }
                  break;
           }

           return false;
    }
```

The isCollide method detects if the block collided with one of the sides of the screen, if so it returns true, otherwise it returns false.

```
}current_moving_block;
```

12.4 The GameMatrix Class

The GameMatrix has internally two matrices, the single_block matrix and the block_store matrix. The single_block matrix in addition to the block store matrix together form the digital (number) representation of the game screen. The main objective is to update the single_block matrix according to the moving_block values then the single_block matrix is added to the block store matrix to form the digital representation of the game screen. This representation is taken to the Paint class to draw it to the screen. The implementation of the game matrix class is as follow:

```
class GameMatrix
{
private:
    int single_block[MATRIX_ROWS][MATRIX_COLS];
    int block_store[MATRIX_ROWS][MATRIX_COLS];
public:
    GameMatrix()
```

```
{
	ClearSingleBlockMatrix();
	ClearBlockStoreMatrix();
}
void ClearSingleBlockMatrix()
{
	for(int i=0;i<MATRIX_ROWS;i++)
	{
		for(int j=0;j<MATRIX_COLS;j++)
			single_block[i][j]=0;
	}
}
```

The ClearSingleBlockMatrix sets all the values of the single_block matrix to be zero

```
void ClearBlockStoreMatrix()
{
	for(int i=0;i<MATRIX_ROWS;i++)
	{
		for(int j=0;j<MATRIX_COLS;j++)
			block_store[i][j]=0;
	}
}
```

The ClearBlockStoreMatrix sets all the values of the block_store matrix to be zero

```
void setSingleBlockElement(int row,int col,int value)
{
	if(row >= 0 && row < MATRIX_ROWS && col >= 0 && col < MATRIX_COLS)
	single_block[row][col]=value;
}
```

The setSingleBlockElement sets the value at the row position row and column position col to the value "value". The function checks to see if the row and column are within the matrix boundaries.

```
int getBlockStoreMatrixElement(int i,int j)
{
	if(i >= 0 && i < MATRIX_ROWS && j >= 0 && j < MATRIX_COLS)
		//touch block
		return block_store[i][j];
	else
		//touch the walls
		return -1;
}
```

The function getBlockStoreMatrixElement gets the value of the block store element at the row position i and the column position j. Notice if the value is -1 means the block is outside the boundary and the block hit the wall.

```
bool isBlockTouchBlock(Sides sd)
{
	if(sd == BOTTOM)
	{

		for(int i=0;i<MATRIX_ROWS;i++)
		{
			for(int j=0;j<MATRIX_COLS;j++)
			{
				if(single_block[i][j] > 0)
				{
					if(getBlockStoreMatrixElement(i+1,j)>	0	||
					getBlockStoreMatrixElement(i+1,j)	==	-1)
					return true;
				}
			}
		}
	}
	else if(sd == RIGHT)
	{
		for(int i=0;i<MATRIX_ROWS;i++)
			{
				for(int j=0;j<MATRIX_COLS;j++)
				{
					if(single_block[i][j] > 0)
					{
						if(getBlockStoreMatrixElement(i,j+1)>  0
						||  getBlockStoreMatrixElement(i,j+1)  ==
						-1) return true;
					}
				}
			}
	}
	else if(sd == LEFT)
	{
		for(int i=0;i<MATRIX_ROWS;i++)
		{
			for(int j=0;j<MATRIX_COLS;j++)
			{
				if(single_block[i][j] > 0)
				{
					if(getBlockStoreMatrixElement(i,j-1)>	0	||
					getBlockStoreMatrixElement(i,j-1)	==	-1)
					return true;
				}
```

```
                }
            }
        }
        return false;
}
```

The isBlockTouchBlock function takes one side of the game screen and check if the moving block hits this side or not. If the side is the BOTTOM we check the current position of the moving block with the next row position of the block store matrix, if value in that position in the block store is greater than zero this means it has a block and the moving block hit it from the BOTTOM. Also the hitting will occur if the next position in the block store matrix is out of the boundary (hits the bottom boundary)

The same concept can be applied to the remaining sides.

```
bool isBlockStoreFull()
{
        for(int j=0;j<MATRIX_COLS;j++)
        {
                //if the first row has blocks
                if(block_store[0][j] > 0) return true;
        }
        return false;
}
```

The isBlockStoreFull checks the first row in the block store matrix, if one of its values is greater than zero this means it has a block and the block store is full or Game Over.

```
bool isCompleteRow(int row)
{
        for(int j=0;j<MATRIX_COLS;j++)
        {
                //if you find an empty spot return false
                if(block_store[row][j] == 0) return false;
        }
        pcResult+=4;
        return true;
}
```

The isCompleteRow checks the entire row values if one of them is zero it will return false otherwise it will return true. The player score will be increased by 4 since the entire row is 16 and the block has four elements, so by completing one row means completing 4 blocks.

```
void shiftBlocksDown()
{
        for(int i=0;i<MATRIX_ROWS;i++)
        {
                if(isCompleteRow(i))
```

```
		{
			//do the shift starting from the row
			//which is full moving backwords to
			//reach the top of the matrix
			for(int k=i;k>0;k--)
			{
				//first clear the full row
				for(int j=0;j<MATRIX_COLS;j++)
				{
					block_store[k][j]=0;
				}
				//second: move the data of the above row to
				//the full row
				for(j=0;j<MATRIX_COLS;j++)
				{
					block_store[k][j]=block_store[k-1][j];
				}
			}
		}
	}
}
```

The shifBlocksDown function uses the isCompletRow function and iterates all rows, if the row is completed it starts from this position k=i until k=0. In other words it will start from the full row and move up row by row until it reaches row 0. In every iteration, it clears the current row and copies the upper row values to the current row.

```
void DrawBlockStoreMatrix(void)
{
	int unit=game_settings.getGameRectLength();
	//this function to scan the blocks_store matrix and
	// if find an element not zero, so it will draw a single
	// rectangle "block element" in the corrisponding pos.

	int combined_element;
	for(int i=0;i<MATRIX_ROWS;i++)
		for(int j=0;j<MATRIX_COLS;j++)
		{
			combined_element=block_store[i][j]+single_block[i][j];

			if(combined_element!=0 )
			{

				RECTANGLE
temp_rect={6*unit+j*unit,unit+i*unit,6*unit+j*unit+unit,unit+i*unit+unit};

paint_background.DrawTexturedRectangle(temp_rect,combined_element+4);
			}
		}
```

```
}
```

The DrawBlockStoreMatrix is mainly for drawing the block store matrix to the screen, it checks to see if the element is greater than zero is draw the corresponding element with the required texture.

```
void addBlockToBlockStore()
{
	for(int i=0;i<MATRIX_ROWS;i++)
	{
		for(int j=0;j<MATRIX_COLS;j++)
			block_store[i][j]=block_store[i][j]+single_block[i][j];
	}
}
```

The addBlockToBlockStore adds the current moving block to the block store matrix.

```
void UpdateSingleBlockMatrix(Block *current_block)
{
	//get the block info
	int row =current_block->getRow();
	int col =current_block->getCol();
	int rotation=current_block->getRotation();
	int shape_value=current_block->getShape();
	//set all elements to be all zeroes
	ClearSingleBlockMatrix();
	switch(shape_value)
	{
	case 1: //T shape
		if(rotation == 1)
		{
			setSingleBlockElement(row,col,shape_value);
			setSingleBlockElement(row-1,col,shape_value);
			setSingleBlockElement(row-1,col-1,shape_value);
			setSingleBlockElement(row-1,col+1,shape_value);
		}
		else if(rotation == 2)
		{
			// this shape   |-
			setSingleBlockElement(row,col,shape_value);
			setSingleBlockElement(row-1,col,shape_value);
			setSingleBlockElement(row-1,col+1,shape_value);
			setSingleBlockElement(row-2,col,shape_value);
		}
		else if(rotation == 3)
		{
			// this shape   _|_
			setSingleBlockElement(row,col,shape_value);
```

```
                setSingleBlockElement(row,col+1,shape_value);
                setSingleBlockElement(row-1,col+1,shape_value);
                setSingleBlockElement(row,col+2,shape_value);
        }
        else
        {
                // this shape   -|
                setSingleBlockElement(row,col,shape_value);
                setSingleBlockElement(row-1,col,shape_value);
                setSingleBlockElement(row-1,col-1,shape_value);
                setSingleBlockElement(row-2,col,shape_value);
        }
        break;
case 2: //L shape
        if(rotation == 1)
        {
                //this shape |_
                setSingleBlockElement(row,col,shape_value);
                setSingleBlockElement(row,col+1,shape_value);
                setSingleBlockElement(row-1,col,shape_value);
                setSingleBlockElement(row-2,col,shape_value);
        }
        else if(rotation == 2)
        {
                // this shape   ___|
                setSingleBlockElement(row,col,shape_value);
                setSingleBlockElement(row,col+1,shape_value);
                setSingleBlockElement(row,col+2,shape_value);
                setSingleBlockElement(row-1,col+2,shape_value);
        }
        else if(rotation == 3)
        {
                // this shape  ''|
                setSingleBlockElement(row,col,shape_value);
                setSingleBlockElement(row-1,col,shape_value);
                setSingleBlockElement(row-2,col,shape_value);
                setSingleBlockElement(row-2,col-1,shape_value);
        }
        else
        {
                // this shape   |'''''
                setSingleBlockElement(row,col,shape_value);
                setSingleBlockElement(row-1,col,shape_value);
                setSingleBlockElement(row-1,col+1,shape_value);
                setSingleBlockElement(row-1,col+2,shape_value);
        }
        break;
case 3: //Z shape
        if(rotation == 1)
        {
                setSingleBlockElement(row,col,shape_value);
```

```
                setSingleBlockElement(row,col+1,shape_value);
                setSingleBlockElement(row-1,col+1,shape_value);
                setSingleBlockElement(row-1,col+2,shape_value);
        }
        else if(rotation == 2)
        {
                setSingleBlockElement(row,col,shape_value);
                setSingleBlockElement(row-1,col,shape_value);
                setSingleBlockElement(row-1,col-1,shape_value);
                setSingleBlockElement(row-2,col-1,shape_value);
        }
        else if(rotation == 3)
        {
                setSingleBlockElement(row,col,shape_value);
                setSingleBlockElement(row,col+1,shape_value);
                setSingleBlockElement(row-1,col,shape_value);
                setSingleBlockElement(row-1,col-1,shape_value);
        }
        else
        {
                setSingleBlockElement(row,col,shape_value);
                setSingleBlockElement(row-1,col,shape_value);
                setSingleBlockElement(row-1,col+1,shape_value);
                setSingleBlockElement(row-2,col+1,shape_value);
        }
        break;
case 4: //I shape
        if((rotation % 2) == 0)
        {
                setSingleBlockElement(row,col,shape_value);
                setSingleBlockElement(row,col+1,shape_value);
                setSingleBlockElement(row,col+2,shape_value);
                setSingleBlockElement(row,col+3,shape_value);
        }
        else
        {
                setSingleBlockElement(row,col,shape_value);
                setSingleBlockElement(row-1,col,shape_value);
                setSingleBlockElement(row-2,col,shape_value);
                setSingleBlockElement(row-3,col,shape_value);
        }
        break;
case 5: //O shape
        setSingleBlockElement(row,col,shape_value);
        setSingleBlockElement(row-1,col,shape_value);
        setSingleBlockElement(row,col+1,shape_value);
        setSingleBlockElement(row-1,col+1,shape_value);
        break;
default:
        break;
}
```

```
}
```

The UpdateSingleBlockMatrix updates the value in the single block matrix according to the fields of the moving block. The function takes a pointer to the moving block and gets the values of all of its fields then it uses the block fields to update the single block matrix. The values in the single block matrix will be different according to the shape value and rotation value.

For example if the shape value is 1 and the rotation value is also 1, we need to draw the "T" shape, we start by updating the pivot element which is the most lower element (the block is rotating around this element) the element is at (row,col) positions. Next we move up one step and draw the next element which is at (row-1,col), next we move left to draw the element to the left and then move right to draw the element at right.

12.5 The Sequential Execution of the Game

We have completely explained the building blocks of the game, now it is time to use the entities together and see how to join them to make a running game. The game will be running in a sequence depicted in the following pseudo-code

- Draw the background according to which state you are in.
- In state 2
    - Draw the background images
    - Shift blocks down (if complete row) and increase the score of the player
    - If block touches block from the bottom so we have two cases
      Case 1: Either this is a game over (the top row is full)
      Case 2: Or it is just the block settled down to the bottom of the screen or on the top of the block store, in this case we just need to add the block to the block store.
    - Else move the block down one step.
    - Update the GameMatrix by updating the position of the moving block.
    - Draw the matrix to the screen.
- In state 1
    - Draw the background images
    - Draw the menu itmes.

The above pseudo-code is depicted by the following code

```
void display()
{

   if(current_state.getState() == 2)
   {
```

```
        glClear(GL_COLOR_BUFFER_BIT);
        glLoadIdentity();

        glColor3f(1,1,1);
        paint_background.DrawBackgroundImage();

        glColor3f(0,0,0);
        paint_background.DrawGameSubwindows();

        //if there is a complete row shift down
        game_matrix.shiftBlocksDown();

        if(game_matrix.isBlockTouchBlock(BOTTOM))
        {
               if(game_matrix.isBlockStoreFull())
               {
                      //game over
                      current_moving_block.haltGame();
                      paint_background.printMessageGameOver();
                      pcResult=0;
                      game_settings.resetElapsedTime();
                      game_matrix.ClearBlockStoreMatrix();
               }
               else
               {
                      //if block touches block save the current block
                      //to the block store and start new shape
                      current_moving_block.resetBlock();
                      game_matrix.addBlockToBlockStore();
               }
        }
        else
        {
               current_moving_block.MoveBottom(false);
        }
        //apply changes of the block pos to the current
        //block
        game_matrix.UpdateSingleBlockMatrix(&current_moving_block);
        glColor3f(1.0f,1.0f,1.0f);
        game_matrix.DrawBlockStoreMatrix();

        glColor3f(1.0f,1.0f,1.0f);
        if(current_moving_block.isHaltMode())
        {
               paint_background.printMessageEsc();
        }
}
else if(current_state.getState() == 1)
{
        //Menu screen
        glClear(GL_COLOR_BUFFER_BIT);
```

```
            glLoadIdentity();

            glColor3f(1,1,1);
            paint_background.DrawBackgroundImage();

            glColor3f(0.6,0.3,0.6);
            paint_background.DrawGameMenu();

    }
    else if(current_state.getState() == 3)
    {
            //top score screen
            glClear(GL_COLOR_BUFFER_BIT);
            glLoadIdentity();
    }

    glutSwapBuffers();
}
```

Notice that the game will be running this way until the first row has block, i.e game over. But, how to make the game interactive? this is done by interacting with the game by the keyboard.

The game can be controlled by pressing one of the following keys:

- KEY_UP
- KEY_DOWN
- KEY_LEFT
- KEY_RIGHT

KEY_UP will rotate the shape, that if the game in state2 or move the cursor on the menu to the upper item if it is in state1.

KEY_DOWN will force the moving block to move until it touches the bottom screen or touches one block from bottom. If in state 1 it will move the menu cursor to the lower item.

KEY_LEFT will just move the block to right if it is not touches another block from left.

KEY_RIGHT will move the block to the left if it is not touches another block from right.

The code for the above procedure is shown in the following code:

```
void keyboard_s (int key, int x, int y)
{

    switch (key)
    {
        case GLUT_KEY_UP:
                if(current_state.getState() == 2)
```

```
                {
                        current_moving_block.RotateShape();
                }
                else if(current_state.getState() == 1)
                {
                        paint_background.setPrevMenuItem();
                }
        break;
        case GLUT_KEY_DOWN:
                if(current_state.getState() == 2)
                {
                        //move the block until it touches other block
                        while( ! game_matrix.isBlockTouchBlock(BOTTOM)  )
                        {
                                current_moving_block.MoveBottom(true);

                        game_matrix.UpdateSingleBlockMatrix(&current_moving_block);
                        }
                        game_settings.increaseElapsedTime();
                }
                else if(current_state.getState() == 1)
                {
                        paint_background.setNextMenuItem();
                }
        break;
        case GLUT_KEY_LEFT:

                        if(! game_matrix.isBlockTouchBlock(LEFT))
                        {
                                current_moving_block.MoveLeft();

                        game_matrix.UpdateSingleBlockMatrix(&current_moving_block);
                        }

        break;
        case GLUT_KEY_RIGHT:
                if(! game_matrix.isBlockTouchBlock(RIGHT))
                {
                        current_moving_block.MoveRight();
                        game_matrix.UpdateSingleBlockMatrix(&current_moving_block);

                }
        break;
    }
}
```

12.6 Conclusion

The tetris game logic has been implemented. The moving objects can be controlled by using the keyboard arrow, move up (flip object), move down(release), move right, and move left.

There is a collection of objects in the store. When there is a complete row, the player score is increased and the row is removed from the store. The game is terminated when the store is increased to the window limit (no space for moving the next object) or the players presses Esc button from the keyboard.

References

[1] Donald Hearn and M. PaulineBaker *A Computer Graphics with OpenGL, Third Edition*, Prentice Hall, 2004 ISBN: 0-13-015390-7.

[2] Salmon, *Computer Graphics and Concepts*, Addison-Wesley, 1988.

[3] Foley, *Computer Graphics Principles and Practice*, Addison-Wesley, 1990.

[4] Watt, Alan, *3D Games: Real-time Rendering and Software Technology Volume 1*, Addison-Wesley, 2001.

[5] Josuttis, N., *The C++ Standard Library: A Tutorial and Reference*, Addison-Wesley, 1999

[6] A. Rollings and D. Morris, *Games Architecture and Design: A New edition*, New Riders Publishing, 2004.

[7] T. Barron, *Strategy Games Programming with DirectX™ 9.0*, Wordware Publishing Inc, 2003.

[8] A. Champandard, *AI Games Development: Synthetic Creatures with Learning and Reactive Behaviours*, New Riders Publishing, 2004.

Websites

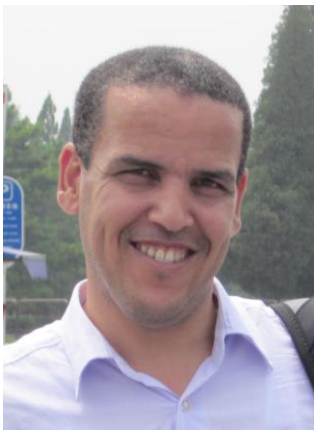
[1] CMP Group, *Gamasutra: The Art and Science of Making Games,* http://www.gamasutra.com/

[2] International Games Developers Association, http://www.igda.org/

[3] http://www.lua.org/

[4] http://www.aigamedev.net/

About the author



Osama Hosam Eldeen Is an Assistant Professor in Research City for Scientific Research and Technology Applications, Alexandria, Egypt. In 2007 he received his MSc. In computer systems and engineering, He pursued his study in Hunan University, China and worked in parallel in Nanjing University of Technology; in 2011 he received his PhD in Computer Engineering.. His research interests include, Computer Graphics, 3D Watermarking, Stereo Vision, and Pattern Recognition