OPENGL CHESS

Credit: https://github.com/stevenalbert/3d-chess-opengl

Become a C++ chess master

Part One! Don't miss next issue, subscribe on page 16!

Tiring of retro gaming rewrites, **Andrew Smith** turns his grandmaster mind to more classical games in a 3D style.

OUR EXPERT

Andrew Smith is a software developer for NHS England. He enjoys video gaming and started coding in C/C++ in 1997 on a Borland C++ for DOS compiler.

We're going to look at a 3D chess program that has been created in C++ and OpenGL. Throughout the decades, there have been many chess programs (both 2D and 3D) created on various platforms that can be played in various ways: single-player against an AI opponent or multiplayer (two players on same machine or across LAN/internet).

The chess program that we are going to look at has been created as a two-player game that can be played on the same machine. It was created by Steven Albert and the original source code can be found at **https://github.com/stevenalbert/3d-chess-opengl**.

For this tutorial, the original source code has been modified. The tutorial involves writing a basic OpenGL program to test that everything has been set up and installed correctly for an OpenGL program to be run. It also includes a walk-through of some of the 3D chess program code and how to solve some basic errors that occur when the project has been compiled. Before we go further, let's set up our development environment.

Hello OpenGL

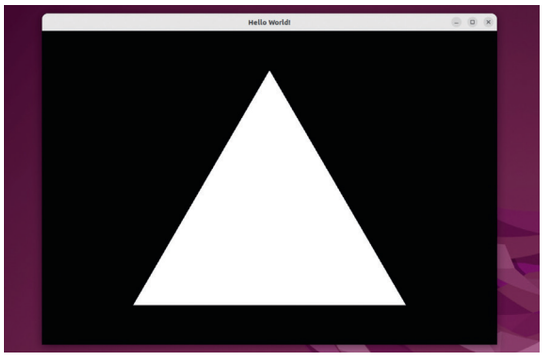
Before a program that uses OpenGL can be run, we must ensure OpenGL is installed and set up correctly. Type in the following code to do this and to clone the repository we need from GitHub for this tutorial:

```
$ sudo apt-get update
$ sudo apt-get install libglu1-mesa-dev freeglut3-dev mesa-common-dev
$ git clone https://github.com/asmith1979/lxf301_chesspt1
```

Now that we have set up and installed OpenGL, before we go any further we will create a basic OpenGL program to make sure everything has been installed correctly. On successful execution, the program should just display a triangle (or polygon) as shown in the screenshot *(above-right)*.

Navigate into the cloned project folder, **lxf301_chesspt1**, open your chosen editor and type in the following C++/OpenGL code:

```
#include <GL/glut.h>
void displayMe(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glBegin(GL_POLYGON);
      glVertex3f(-0.6, -0.75, 0.5);
      glVertex3f(0.6, -0.75, 0.0);
      glVertex3f(0.0, 0.75, 0.0);
   glEnd();
   glFlush();
}
int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE);
   glutInitWindowSize(400, 300);
   glutInitWindowPosition(100, 100);
   glutCreateWindow("Hello world!");
   glutDisplayFunc(displayMe);
   glutMainLoop();
   return 0;
}
```



The successful execution of the test OpenGL program to demonstrate that OpenGL has been properly installed on the system.

Save the file as **tstOpenGL.cpp**. The code is already written in a file called **testOpenGL.cpp**, but if you are new to OpenGL, this is a good example program to start practising with.

Now that the code has been written for our test OpenGL program, we need to compile and build an

executable file so that the program will run. Type the following to build an executable file:

```
$ g++ tstOpenGL.cpp -o firstOpenGlApp -lglut -lGLU -lGL
```

The above line of code, if successful, should build an executable program called **firstOpenGlApp**, which can be executed by typing `./firstOpenGLApp`. If a window appears with no triangle shape shown, try to resize the window with your mouse so that the triangle appears. For those who are new to C/C++ and OpenGL, we will now look at what this basic program does to display our polygon on screen.

The entry point to our program begins with the **main** function shown in the example code. The code inside the **main** function is used to initialise then set up various aspects of OpenGL to use later on (in a bigger program). The code of interest in the **main** function is:

```
glutDisplayFunc(displayMe);
```

This line of code in the **main** function identifies the rendering function used in the OpenGL program. In this case, it is a function called **displayMe**, which is used to display the polygon on screen. As OpenGL is a 3D graphics API, every point is defined with a three-dimensional vertex, specifying x, y and z as parameters. A point of a vertex is defined using the **glVertex3f** function call that takes in three floating-point type values (decimal values). The way in which OpenGL is used to define a point in 3D space is as follows:

```
glVertex3f(0.5, 0.0, 0.5);
```

After running the program and seeing it displays our polygon, close the program by either pressing Ctrl+C or closing the window so we can continue to look at the main OpenGL game project now we know everything is set up for it to work. If you are new to OpenGL, you may want to play around with the values put into the **glVertex3f** functions, rebuild and see the result. To start viewing the main game project, navigate into a folder called **3d-chess-opengl**.

Chess mate!

The project includes the following folders and content:

- **build.txt:** a file that contains the compile/build code.
- **main.cpp:** a C++ code file that is the entry point.
- A folder called **Chess** containing .h and .cpp files.
- A folder called **model** containing chess piece data.
- **Model.cpp** and **Model.h**, which deal with model data.

Before we look at the code, have a go at running the game by typing the following:

```
$ g++ main.cpp Model.cpp ./Chess/Board.cpp ./Chess/Game.cpp ./Chess/Gameplay.cpp ./Chess/GameStatus.cpp ./Chess/Move.cpp ./Chess/Piece.cpp ./Chess/Square.cpp -o chessgame -lglut -lGLU -lGL
```

The chess program should compile with warning errors, which we will deal with later.

Run the program by typing `./chessgame` then press the N key to start the game. The program is constructed in a way that allows two players to take turns playing on the same machine, all keyboard-driven. To select a chess piece, use the a, d, w and s keys, and press Space to move that chess piece. Once a move has been made, the board rotates so the other player can make their move. Once you have got used to how the program works, close the program window and go back to the editor to go through some of the code that makes the game work.

The entry point to the game project is **main.cpp**, so if you have not done so already, open it in your editor. As you begin to look down through **main.cpp** you will notice various variables and functions declared, one of them being a function called **main**, which is used as the main entry or starting point to the program. As explained, the **main** function is used to set up and initialise various aspects of OpenGL that we'll want to use later on. As this has been covered earlier, we won't go into further detail of what is in the **main** function.

At the very top of **main.cpp** is a mixture of libraries and variables that are needed for the game project to function properly. Near the top, variables and values are defined for the window position and size that you may wish to alter to suit your desired screen size:

```
//Window size and position
#define WINDOW_WIDTH 800
#define WINDOW_HEIGHT 600
#define WINDOW_POS_X 50
#define WINDOW_POS_Y 50
```

If you are new to C++/OpenGL coding, remember that any code changes you make require a rebuild of the code to see the changes take effect in the game. Unlike Python, C++ is not an interpreted language.

Also near the top of **main.cpp**, each of the chess pieces are loaded into memory:

```
/**
  Model Loading
*/
Model  Pawn("model/Pawn.obj");
Model  Rook("model/Rook.obj");
Model  Knight("model/Knight.obj");
Model  Bishop("model/Bishop.obj");
Model  King("model/King.obj");
Model  Queen("model/Queen.obj");
```

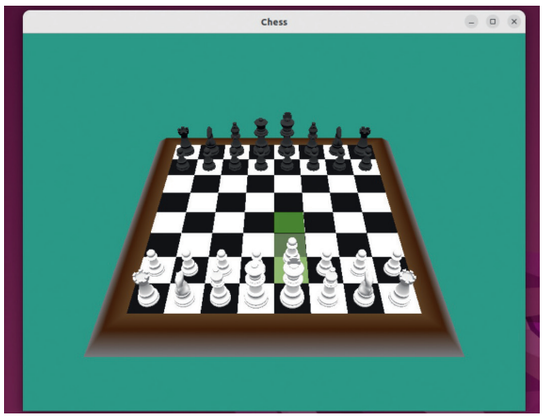
The main rendering function is defined as **displayFunction**, which is responsible for the overall graphics processing of the games program as well as

QUICK TIP

There is a file in the folder of the chess program and of the OpenGL test program called build.txt, which contains the build commands that can be copied and pasted into command line.

» OBJECT-ORIENTED PROGRAMMING (OOP)

A good example in this project where OOP has been used is the **Model** class, defined in **Model.h** and implemented in **Model.cpp**. The sole objective of this class is to load a 3D model object (chessboard piece) into memory so it can be used later on in the program. A class includes variables (or properties) and functions (or methods) in one unit that will contribute to that main purpose. From viewing the **Model** class (**Model.h** and **Model.cpp**), it contains a constructor of the same name that takes in a path to the chess piece needed to be loaded into memory. A class constructor is always executed first on the creation and execution of a class. As seen, there are various properties and functions created to help load in the object model. Also, in C++ there are access specifiers, most commonly public and private, that determine the level of access to the methods and properties used. Private members can only be accessed by elements of the **Model** class itself, while public elements can be accessed by objects outside the **Model** class.



When a chess piece on the board is selected, the valid moves the piece can perform are highlighted so they can be selected by the player.

other essential operations such as updating the board after a player has taken their move, for example. It is also worth mentioning that **displayFunction** is regarded as a callback function, in that it is consistently called throughout the game program's operation until the game comes to an end state. A boolean variable called **inGame** is used to determine this.

There are a lot of graphical processing elements that happen in the rendering function, which are outside the scope of this tutorial – however, the rendering function engages in what is often known as the rendering graphics pipeline, where various operations have to happen in a certain order for the graphics to get processed and display on screen correctly. Without going through the graphics rendering pipeline process, the graphics for the game program would not display correctly or even at all. Usually, the very first stage of the process is to clear what is on the screen (start afresh), sometimes known as clearing the buffer. The code below shows the start of the rendering function, called **displayFunction**:

```
void displayFunction()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_
BUFFER_BIT);

  if(inGame)
  {
    /**
      Changing view perspective
    */
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(fovy, screen_ratio, zNear, zoomOut *
zFar);
    /**
      Drawing model mode
    */
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(zoomOut * eyeX, zoomOut * eyeY,
zoomOut * eyeZ, centerX, centerY, centerZ, upX, upY,
upZ);
    /**
      Draw code here
    */
    if(board_rotating) doRotationBoard(chess-
>getTurnColor());
    GLfloat ambient_model[] = {0.5, 0.5, 0.5, 1.0};
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT,
ambient_model);
    glMaterialfv(GL_FRONT_AND_BACK, GL_
AMBIENT_AND_DIFFUSE, mat_diffusion);
    glMaterialfv(GL_FRONT_AND_BACK, GL_
SPECULAR, mat_specular);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, diffusion);
...
```

After the screen has been cleared with the **glClear** function, the way in which various aspects of the game scene are viewed in the game program is set up. In OpenGL, there is a concept of a camera and a lens, where the Model view represents the camera itself, where you are able to control aspects such as position and pointing. The Projection view is where you are able to control and manipulate what is the equivalent of the camera's lens.

OpenGL also includes the ability to introduce and control different lighting sources in the game scene, which will react directly with the materials used on the models (chess pieces here) imported into the game. Ambient lighting is used in this project. This is a type of lighting that is used to light up (make viewable) the whole game scene, where light is not coming from a particular source but from all around. Below is a brief overview of what some of the functions displayed in the above example code are used for:

- **glClear** – clears buffers to present values.
- **glMatrixMode** – specifies which matrix is the current matrix being used.
- **glLoadIdentity** – replaces the current matrix with the identity matrix.
- **gluPerspective** – used to set up a perspective projection matrix.
- **gluLookAt** – defines a viewing transformation.
- **glLightModelfv** – sets lighting model aspects.
- **glMaterialfv** – specifies material parameters for the lighting model.

If you are new to OpenGL and 3D programming, the above list of functions should give you a good starting point to find out further information on their use using Google, YouTube and so on – as there are many ways these functions can be configured and used, it is outside the scope of this tutorial to cover these functions in detail. For the remaining part of the rendering algorithm, the functionality focuses more on the gameplay functionality after configuring the view of the game, such as:

- Drawing the chessboard on screen.
- Drawing the board squares on the chessboard.
- Drawing the chessboard pieces.
- Identifying valid moves.

Before moving on to changing some of the source code to remove the warning errors we received earlier when compiling and building an executable file, we will look at the code and functionality of how valid moves are identified.

As you will have seen earlier from playing the game program, when a chess piece is selected on the board, the program highlights all valid moves in which the piece selected can be moved, because each piece on the chessboard can only be moved in certain ways.

```
void drawValidMoves()
{
  if(selected)
```

.H AND .CPP FILES

C++ is an object-oriented programming language and classes are implemented usually across multiple files, mainly .h and .cpp files – .h files contain properties and method signatures to be used in the program. A good example in this project is the **GameStatus** class in the **Chess** folder. This has been implemented in both **GameStatus.h** and **GameStatus.cpp**. When first viewing **GameStatus.h**, various library files have been included at the top using `#include`, which is the C++ way of including library files. Further down, the **GameStatus** class is created with public and private access specifiers along with the properties and method signatures to be used. **GameStatus.cpp** first includes **GameStatus.h** and then implements the methods as defined in **GameStatus.h**. Notice the way methods are referred to in **GameStatus.cpp**:

```
void GameStatus::setKingMove(Piece
Color color)
{
        if(color==PieceColor::WHITE)
whiteKingMove = true;
        else if(color==PieceColor::BLACK)
blackKingMove = true;
}
```

In the above code, the class name **GameStatus** is written out followed by two colons and then the method name itself. Without doing this, it would result in a compile error if and when we tried to build the executable file (usually a linker error). The parameters and parameter data types that are passed in also have to match as this may also result in a compile error if it does not match with the .h file. In general practice, the code is first written in the .h file and then implemented in the .cpp file.

```
  {
    std::vector<Move> validMoves = chess-
>getValidMoves(selectedRow, selectedCol);
    int vec_size = validMoves.size(), row, col;
    for(int id = 0; id < vec_size; id++)
    {
      row = validMoves[id].getDestinationPosition().first;
      col = validMoves[id].getDestinationPosition().
second;
      switch(validMoves[id].getType())
      {
        case MoveType::NORMAL:
          glColor3f(0.8f, 1.0f, 0.6f);
          break;
        case MoveType::CAPTURE:
          glColor3f(1.0f, 0.0f, 0.0f);
          break;
        case MoveType::EN_PASSANT:
          glColor3f(0.8f, 1.0f, 0.6f);
          break;
        case MoveType::CASTLING:
          glColor3f(0.196f, 0.804f, 0.196f);
          break;
      }
...
```

To start with, before any checking is done, the algorithm checks to see if a chess piece has been selected, identified by the boolean variable selected. A collection of valid moves is created based on selected row and column of the chess piece, and is stored in a collection called **validMoves**. A switch/case statement is used to identify what sort of move the player is attempting to do, which could be moving normally, attacking, castling and so on, each of which has a different colour. You'll have seen for moving, the colour used to identify a valid move of a chess piece is green. This is all encapsulated in a **for** loop that processes each valid chess move in turn for the selected chess piece, which is graphically highlighted at the end.

Fixing warning errors

You may recall that when compiling and building the source code, there were warning errors shown when the executable file was produced. We will now attempt to fix the warning errors so they do not appear when the source code is compiled and built in the future. In your chosen editor, navigate into the **Chess** folder and open the file called **GameStatus.cpp.** There are four warnings in total, all of the same problem, so we will cover one here and using what we do in this case, you will then need to apply it to the other areas of the code to fix the other warning errors. Scroll down or search for a method called **isKingMove**.

```
bool GameStatus::isKingMove(PieceColor color)
{
        if(color==PieceColor::WHITE)
                return whiteKingMove;
        else if(color==PieceColor::BLACK)
                return blackKingMove;
}
```

The above code needs to be changed to:

```
bool GameStatus::isKingMove(PieceColor color)
{
        if(color==PieceColor::WHITE)
        {
                return whiteKingMove;
        }
        else
        {
                return blackKingMove;
        }                }
```

The way the condition was written originally had three outcomes, but the code only accounted for two outcomes, hence why the warning error was appearing. As the code only needed to account for two outcomes, the condition has been amended so that only two outcomes are accounted for, in that it will return whether it is white's or black's turn to move. Now that this has been corrected for **isKingMove**, the same needs to be done for the remaining three methods: **pieceEnPassantable**, **isFirstColRookMove** and **isLastColRookMove**. Now when you compile and rebuild the code, the warning errors should no longer appear. **GameStatus_lxf.cpp** contains the modified code for this, but from a learning point of view, it's better if you attempt to type the code out yourself. LXF

QUICK TIP

Instead of scrolling through code, it may be better to use the editor's search facility to go to directly where you want.