OPENGL CHESS

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

Add controllers and menus with OpenGL

Never a man to miss deadlines, **Andrew Smith** stays in control adding mouse menus to our OpenGL-powered 3D chess game.

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.

This month, we are continuing with the 3D chess program that we started in **LXF301**. We're creating a basic mouse-driven menu system to be shown at the start of the game program and adding game controller support.

The original source code for the 3D chess program can be downloaded or cloned from **https://github.com/stevenalbert/3d-chess-opengl**. Some aspects of the original code have been modified for this tutorial.

We used a PC-compatible Xbox One controller to test and implement the functionality, but other PC-compatible game controllers should also be applicable.

Before we continue any further, let's set up our development environment:

```
$ sudo apt-get update
$ sudo apt-get install libglu1-mesa-dev freeglut3-dev mesa-common-dev
$ sudo apt-get install libglfw3
$ sudo apt-get install libglfw3-dev
$ git clone https://github.com/asmith1979/lxf302
```

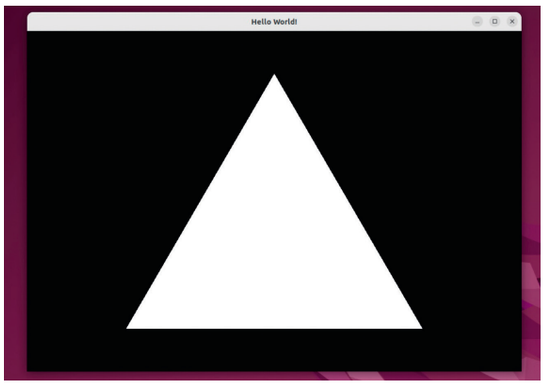
Run the OpenGL test program to make sure OpenGL has been set up properly. There should be a file called **testOpenGL.cpp** in the **lxf302** folder. The `g++ build` command is in the **build.txt** file.

Going on the basis that OpenGL and GLFW have been set up correctly, navigate into the folder called **3d-chess-lxf302.** You should find a file called **build.txt** that contains the build code for the project, which you can copy and paste into the command line to build an executable program called **chessgame**. After this has been built, run the program to familiarise yourself with the chess program. Currently, it is keyboard controlled using the a, s, w and d keys to navigate round the chess board, and Space to select a piece on the board. When you've finished, press Escape to exit the program.

To edit and view the source code, you can use any editor or IDE of your choice. The build code to put into the command line is stored in **build.txt**, which can be copied and pasted to save you having to type out all the commands. Also, as we will be both editing and executing the chess program simultaneously to see the result of code changes we have made, it may be advisable to open up two terminal windows – one used for editing and the other for building and executing the game program.

QUICK TIP

You can find the full source to this project at https://github.com/asmith1979/lxf302/.



The test OpenGL program showing that OpenGL is installed correctly. Adjust the size of the window if it opens with no image.

Game on!

For the first task, we are going to create a mouse-driven menu system. Currently, when the game is started, we are just given a prompt to press the N key to start the game. Even though sufficient, some may argue it's a little too basic. The menu we are going to create will consist of two options: Start Game and Exit, with either option being selected by using the mouse.

As the user will be selecting which option to choose via the mouse, we must set up each point of each menu option in memory, which means there will be eight points in total (four for the Start Game menu option and four for the Exit option). Each point could be created from a structure – known as a struct in the C/C++ programming world – but for simplicity we will just use variables to specify points. Near the top of **main.cpp** (after the `#include` entries), type the following:

```
// Start Game Button
int stGame_pt1_x = 300;
int stGame_pt1_y = 183;
int stGame_pt2_x = 454;
int stGame_pt2_y = 183;
int stGame_pt3_x = 300;
```

```
int stGame_pt3_y = 200;
int stGame_pt4_x = 454;
int stGame_pt4_y = 200;
```

As shown in the screenshot *(below-right)*, each variable maps to a point on the Start Game menu option. Next we will amend the text output to show the Start Game menu option. Scroll down near to the end of **displayFunction**, where you will find an **else** clause:

```
else
{
  showWord(-150, 0, "- - Press N to Start The Game - -");
}
```

Amend to the following:

```
else
{
  showWord(-100, 100, "START GAME");
}
```

As our menu system will be controlled via a mouse, we must identify to OpenGL a function that will be used to deal with events from the mouse. We will call the callback function **MyMouse**. Write out the following **MyMouse** callback function in **main.cpp**.

```
void MyMouse(int button, int state, int x, int y)
{
  switch(button)
  {
    case GLUT_LEFT_BUTTON:
      if (state == GLUT_UP)
                  {
                    // Start Game Button
                    if ((x >= stGame_pt1_x) && (x <= stGame_pt2_x) && (y >= stGame_pt1_y) && (y <= stGame_pt4_y))
                    {
            if(!inGame)
              newGame();
            else
              verify = true;
          }
                  }
                  break;
  } // End switch statement
} // End MyMouse
```

Scroll down the **main()** function and add:

```
glutMouseFunc(MyMouse);
```

The line of code we have just inserted tells OpenGL the callback function for the mouse operations we are using. The values we have put in for the variables that represent each point of the Start Game menu option may need to change depending on your screen resolution or setup so they are more in range with what is displayed. The callback mouse function we have implemented, **MyMouse**, takes in four arguments, all of which are integers that identify the mouse button pressed, the state of the button, and x and y position of the mouse pointer. If you do need to adjust the values of the points, you can insert the following lines of code to output the mouse x and y position, which will give you an indication of what to change the values to:

```
cout << x << endl;
cout << y << endl;
```

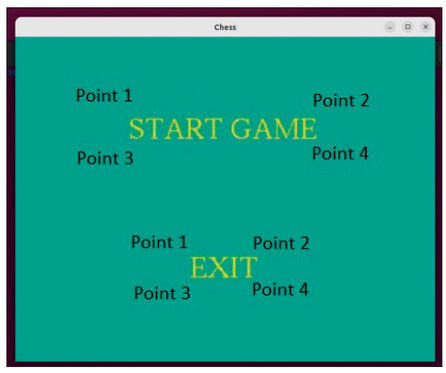
Save your work and rebuild the executable file to run the program again. On successful execution of the program, you should be able to play the game if you click the left mouse button on the Start Game menu option. When you are ready to continue, press the Escape key to exit the program. Now that we have added the Start Game menu option, we need to add the Exit option to exit the game program if the user chooses to do so. To make this possible, another set of variables is needed to specify the four points of the Exit menu option and then an **if** condition needs adding to the **MyMouse** callback function. Have a go at trying to add what is needed to make the Exit menu option work yourself. If you're struggling, the full implementation is in **main_lxf302.cpp**.

Controller support

To add game controller support, we are going to add a header file to the top of **main.cpp**:

```
#include"GameController.h"
```

In **GameController.h** is a class that's been created specifically for using a PC-compatible game controller. Now that we have added the class, we now need to plug the class into the chess game program so that control can be done by both the keyboard and a game controller. Once we have plugged the class into the chess program, we will



The proposed menu and the points needed to define the clicking/selection area of each of the menu options.

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.

» VARIABLE SCOPE

Variables in a C/C++ program can be declared locally, globally or across different files. In this project, variables are mainly declared both globally and locally, which determines how code within the program can access them. Usually, global variables are declared near the top of a C/C++ program, just after the library files have been included, so they can be accessed by functions in the code. For example, the function **processGameControllerOutput()** uses variables such as **actionRightTaken**, **actionLeftTaken**, **actionUpTaken** and **actionDownTaken**. These variables are accessible to any function in the program because they are declared globally. A variable would be a local variable if it was declared inside a function itself, which results in only that function being able to access the variables. In **main.cpp** there is a function called **drawMoveToSquare** that, at the start of the function, declares two local variables of type float that can only be accessed by the code within that function and nowhere else. Variables or values may also be passed into functions as what is known as arguments or parameters, which can be accessed by the code within the function. An example of this is a function in **main.cpp** called **showWord** that takes in two integer values and one string value. This is then used by the code in the function to manipulate text and position.

QUICK TIP

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

look in further detail as to how the **GameController** class works.

To use the **GameController** class, we need to create an instance of it. Near the top of **main.cpp**, where the variables are declared, type the following code:

```
GameController gController;
```

This line of code alone initialises the GLFW library used to interface with the game controller. Scroll down to the main function in **main.cpp** and insert the following after OpenGL has been initialised:

```
cout << "Library enabled: " << gController.
isLibEnabled() << endl;
cout << "Game Controller Detected: " << gController.
GC_Connected() << endl;
```

Now when the game program is run, it outputs on the console if both the library has been initialised and if a game controller has been connected or not. You may want to build the executable again and run the program to check this before continuing.

Next we next going to implement a function called **processGameControllerOutput()** that will provide a link between the **GameController** class and the chess program so the game controller can be used as well as the keyboard. Before we implement this, we need to create some global variables that the function will use. Scroll to the top of **main.cpp** and type the following:

```
int actionRightTaken = -1;
int actionLeftTaken = -1;
int actionUpTaken = -1;
int actionDownTaken = -1;
int aButtonPressed = -1;
```

The variables we have declared above help us to identify when a button on the game controller has been pressed so we can manipulate the control more. There is no callback function for the game controller, so it has to be called from an already defined callback function so that game controller input is detected – in this case, it is the render function **displayFunction**. As a result of this, a button press may be executed more than once, so we need to enforce some control over this, which the above variables help us do.

As the function **processGameControlleOutput()** is fully implemented in **main_lxf302.cpp**, the code below shows a partial implementation of this function, as each controller button is dealt with in the same way. This code shows how to implement code to deal with the left and right directional buttons on the controller:

```
void processGameControllerOutput()
{
        // Left Button Direction
if (actionLeftTaken == -1 && gController.
processGameControllerAction() == LEFT_BUTTON_
DIRECTION)
{           // Left
  if (gController.processGameControllerAction() ==
LEFT_BUTTON_DIRECTION)
  {
    if (!needPromote && !checkMate && !verify &&
inGame && !board_rotating)
          key_A_pressed(chess->getTurnColor());
                    }
   actionLeftTaken = 1;
          }
else if (actionLeftTaken == 1 && gController.
processGameControllerAction() == LEFT_BUTTON_
DIRECTION)
{
  actionLeftTaken = -1;
}
// Right Button Direction
if (actionRightTaken == -1 && gController.
processGameControllerAction() == RIGHT_BUTTON_
DIRECTION)
{
                    if (gController.
processGameControllerAction() == RIGHT_BUTTON_
DIRECTION)
  {
    if(!needPromote && !checkMate && !verify &&
inGame && !board_rotating)
          key_D_pressed(chess->getTurnColor());
  }
                                actionRightTaken = 1;
                    }
else if (actionRightTaken == 1 && gController.
processGameControllerAction() == RIGHT_BUTTON_
DIRECTION)
{
  actionRightTaken = -1;
}
} // End of function
```

An Xbox One controller was tested and used for this tutorial. The GameController class uses the buttons on the game controller.

As this function will be called from in the **displayFunction**, implementation just above this function would be an appropriate place to implement **processGameControllerOutput()**. The variables declared before the implementation of this function are used to tell us if a certain button on the controller has been pressed or not: **-1** to identify not being pressed and **1** to indicate the button has been pressed.

If we look at how the left directional button is dealt with, the condition is first testing to see if the button has not been pressed (**-1** value) along with the left directional button being pressed. If it has, **actionLeftTaken** is set to the value of **1** so the button is not processed more than once. There is also an **if-else** condition to reset **actionLeftTaken** back to **-1** when the button transaction has been completed. This concept is repeated for the right, up and down direction, as well as the A button on the Xbox One controller.

Although we've only implemented code to deal with left and right, you might want to rebuild the executable to try out the new code. Going from the above, try to work out the code for moving up, down and pressing the A button. Viewing **GameController.h** may help or see the full implementation in **main_lxf302.cpp**. Before

doing another build, ensure you've included the following at the top of the **displayFunction**.

```
void displayFunction()
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_
BUFFER_BIT);
   if(inGame)
   {
            // Only Process GameController actions if
GameController is connected
            if (gController.GC_Connected())
            {
                     processGameControllerOutput();
                              }
...
```

Without calling **processGameControllerOutput()**, we will not see our code changes. The call to this function has also been included in an **if** statement that checks to see if the controller is connected – the code is not executed if the controller is not connected. Without the **if** condition, the game program could crash if a controller is not plugged in.

GameController class

Now we've implemented game controller support to the 3D chess game, we'll take a closer look at how the **GameController** class has been created and how the class itself works. The **GameController** class has been implemented in both **GameController.h** and **GameController.cpp** using GLFW library. It contains the following basic functionality:

- Initialise the GLFW library
- Initialise the Game Controller
- Detect if library initialised or not
- Detect if game controller is connected or not
- Detect button presses on Xbox One controllers

The GLFW library is first attempted to be initialised in the **GameController** constructor. On success, a private member called **isLibraryEnabled** is set to true, or false if initialisation has failed. The controller is only initialised if the GLFW library is initialised. Let's look at the constructor and the code to initialise the controller:

```
GameController::GameController()
{
            // Init properties
            isLibraryEnabled = false;                    isGC_
Connected = false;
                     retGAction = -1;
// Attempt to Init library
            if (glfwInit())
            {
                     isLibraryEnabled = true;
            }
            else
            {
                     isLibraryEnabled = false;
            }
}
bool GameController::GC_Connected()
{
                     if (isLibEnabled() == true)
            {
                                       if
(glfwJoystickPresent(GLFW_JOYSTICK_1))
                     {
                                       isGC_Connected = true;
                     }
                     else
                     {
                                       isGC_Connected = false;
                     }                         }
            return isGC_Connected;
}
```

To initialise the controller, the **glfwJoyStickPresent** function is called – this is a function of the GLFW library. This is why an **if** condition is implemented to see if the GLFW library is initialised first.

In **GameController.h**, various macros are set up using the **#define** directive that specify the values of each button press on the game controller. These were manually tested to get the value before implementation into the class (so the implemented values are known values):

```
#define LEFT_BUTTON_DIRECTION 14
#define RIGHT_BUTTON_DIRECTION 12
#define UP_BUTTON_DIRECTION 11
#define DOWN_BUTTON_DIRECTION 13
#define A_BUTTON 0
#define X_BUTTON 2
```

There may be a possibility that these values will need to be altered depending on the game controller you are using. Hopefully, however the above values be sufficient for your PC-compatible controller. LXF

» DATA TYPES

In C/C++, data types can be applied to variables and functions (and methods in C++). The main data types in C/C++ are:

- Integers (int)
- Decimal values (float)
- String (string or char *)
- Boolean (bool)

A lot of the methods used in the **GameController** class have been declared of type bool, which means the method returns a true or false value. Some of the methods in the **GameController** class have been implemented in this way to identify if anything has failed to initialise that the game program needs to work, for example:

```
bool GameController::getGC_ConnectionState()
{
  return isGC_Connected;
}
```

The above method shown is purely there to return the value of **isGC_Connected**, which identifies whether the game controller is connected to the PC or not. Whatever the value is, it is a Boolean value that is returned, so the method is prewritten with the data type bool to identify it will be returning a Boolean value. If the method was to be declared as void, no data type would be returned. It is good practice to data-type functions and methods, so if any issues occur, it can help identify where a problem might be coming from, because functions that don't return a value may mean that you have to debug through the code in that function, even though that may not be where the source of the problem is located.