Graphics

Graphics Report

Demonstrating OpenGL in Python

Using Chess as a Motivating Example

Name: Aaron Lim

Student Number: 46420763

Contents

Introduction

Computer graphics is becoming more and more accessible and easier to implement with modern programs such as blender, unreal-engine etc. However, for casual users, the key concepts and techniques that these programs are based on are not required for you to know.

One of the most popular graphics libraries used is OpenGL - a low level graphics framework. In this report, we will explore OpenGL in Python to create a chess program that allows you to input a PGN (Portable Game Notation) file and display the game as well as play a game of chess against a computer, all rendered in 3D space. Through the use of this program, key-concepts such as 3D rendering, lighting and texture mapping will be demonstrated and discussed.

Method

The project was coded in python using OpenGL APIs with the use of the PyOpenGL. The game itself was rendered in a pygame display using OpenGL with double buffering. pygame was also used to handle user input like button presses and mouse clicks. The pyrr and numpy libraries were used together to create matrices and vectors for various calculations.

For the chess logic implementation, the python-chess library was used as well as the stockfish library which allows us to use a chess engine within python.

Results

Mesh

The first thing that was implemented were the meshes for the various models that will be used in the scene. The meshes outline an objects vertices - containing their position, texture coordinates and normals.

Using obj files, we can load in models using the `load_model_from_file` function from the tutorials. An obj file contains a list of vertices, texture coordinates, normals and faces. A simple cube obj file would look like:

```
# Blender v2.76 (sub 0) OBJ File: ''
# www.blender.org
mtllib cube.mtl
o Cube
v 1.000000 -1.000000 -1.000000
v 1.000000 -1.000000 1.000000
v -1.000000 -1.000000 1.000000
cont...
vt 1.000000 0.333333
vt 1.000000 0.666667
vt 0.666667 0.666667
...
vn 0.000000 -1.000000 0.000000
vn 0.000000 1.000000 0.000000
vn 1.000000 0.000000 0.000000
cont...
usemtl Material
s off
f 2/1/1 3/2/1 4/3/1
f 8/1/2 7/4/2 6/5/2
cont...
```

Lines that begin with **v** denote a position of a vertex, giving its x, y and z coordinate. **vt** denotes a texture coordinate, a horizontal and vertical coordinate on the given texture image. **vn** denotes a normal vector which specifies the direction a surface is facing. **f** denotes the faces which are the polygons that make up the surfaces of the model. The faces in the given example are given by 3 sets of numbers of which each set outlines one point in a triangle. The first number in the set indicates the index of the vertex, the second its texture coordinate and the third its normal vector.

The function `load_model_from_file` takes all of this information and compiles it into a list called vertices which stores the v, vt and vn in that repeating order to create the faces.

These are stored and implemented in the Mesh class. First we have to tell OpenGL the vertex array object and vertex buffer object we're working with. We assign our created vertices list to our VBO and with the GL_STATIC_DRAW flag as we won't be changing the mesh of any of the objects.

```
glBindVertexArray(self.vao)
glBindBuffer(GL_ARRAY_BUFFER, self.vbo)
glBufferData(GL_ARRAY_BUFFER, vertices.nbytes, vertices, GL_STATIC_DRAW)
```

To use the list of vertices, we have to tell openGL how to read it:

```
#position
glEnableVertexAttribArray(0)
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 32, ctypes.c_void_p(0))
#texture
glEnableVertexAttribArray(1)
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 32, ctypes.c_void_p(12))
#normal
glEnableVertexAttribArray(2)
glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, 32, ctypes.c_void_p(20))
```

`glEnableVertexAttribArray()` tells the shader what attribute of the vertex you will be assigning (position, texture or normal denoted in the shader) and `glVertexAttribPointer()` tells the shader the number of elements in that attribute, its type (in this case it's a 32 bit floating-point value), its stride (The byte offset of where the same attribute is stored between consecutive vertices) and a pointer which specifies the offset between attributes.

Using set of free chess piece obj files I found online, the meshes of the pieces were implemented using `load_model_from_file`. A simple quad mesh was also implemented to create the board. The positions of these objects were then placed on the board according to a PGN file. But if we try to render this now, the scene will look wrong! We have to apply some more transformations so that the camera can see them.

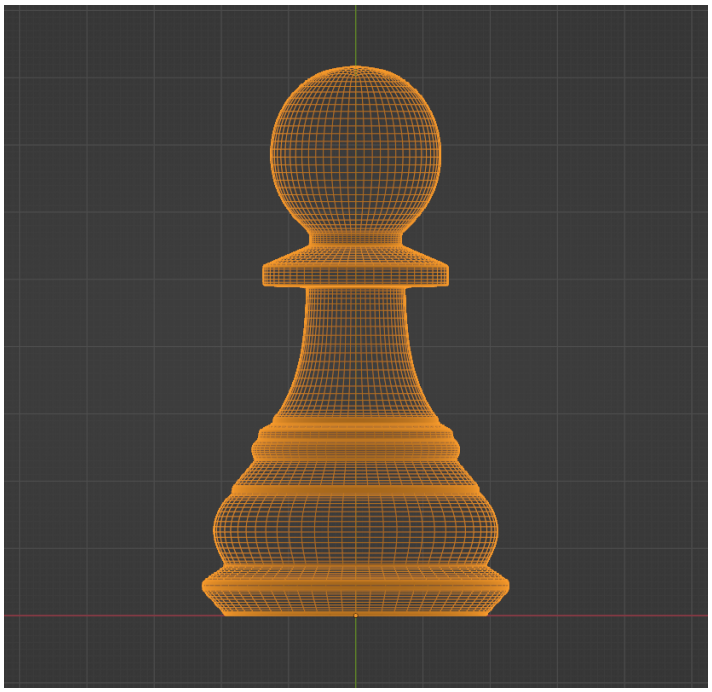


Figure 1: The mesh of the pawn model as seen in blender, outlining the edges and vertices as specified in the .obj file

To do this we have to create a **model** matrix that converts a position of a mesh's vertex in its local space into a model in the world space when multiplied. With a Model class we can define variables such as position, scale and angles which we can use to link to a mesh and specify how it appears in the world space.

3D transformation Pipeline

The **model** matrix is created by first separating it into different transformation matrices. My implementation creates the matrices in the following order: a scaling, an x rotation, a y rotation, a z rotation and a translation transformation matrix. The transformation matrix is created by multiplying these matrices together in the order above. It's useful to note that order matters as matrix multiplication is not commutative. This defines the model's **model** matrix.

In my implementation, these matrices are created using the **pyrr** library which can easily create these matrices.

Now we need to convert these into eye space - the positions relative to the camera's position and the camera is positioned at the origin. This is easily created using `pyrr`'s `create_look_at_()` method which creates the **view** matrix given a camera's position and its up and right vectors.

We now have to convert the eye space into normalized device space (A unit cube) which shows us how the scene looks relative to the perspective of the camera given its fov, aspect ratio and near and far parameters (the nearest and furthest distance an object has to be rendered in the scene). Such matrix is called the **projection** matrix. Again, this matrix is created using `pyrr`'s `create_perspective_projection()`. Without the perspective-projection matrix, the scene will lack depth and look flat:

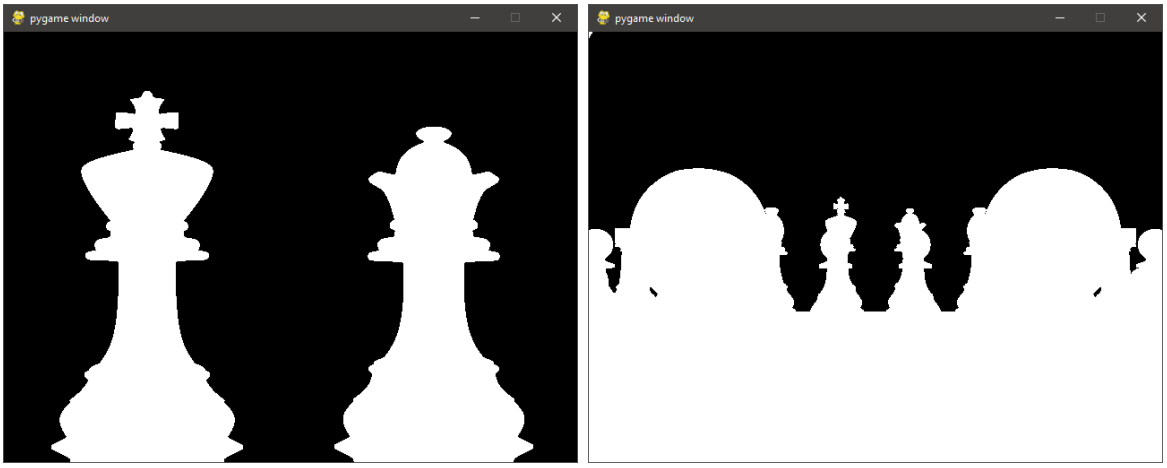


Figure 2: The scene with no projection and with projection

In the shader, the final calculation of a vertex's position is:

```
gl_Position = projection * view * model * vec4(vertexPos, 1.0);
```

Finally, this is converted into the viewport space - the x and y coordinates of the display. This is handled by **pygame**.

Textures

In order to implement textures, a Material class was implemented. Which allows openGL to read a given image file and use it when drawing objects.

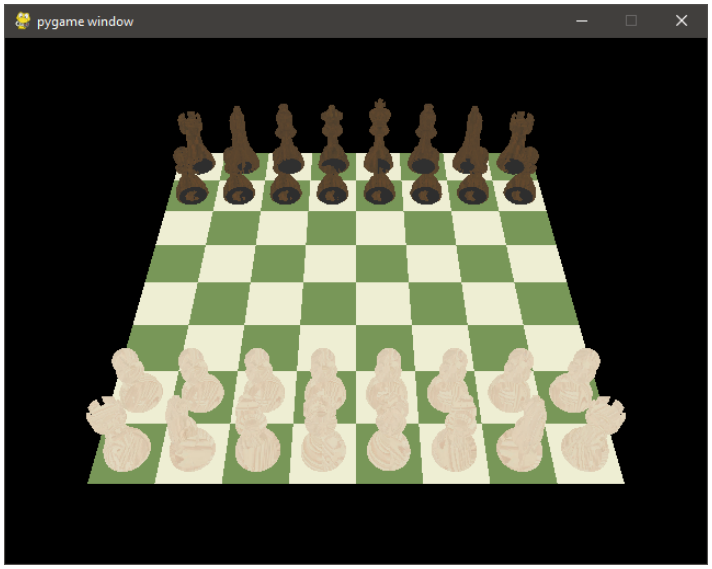
```
class Material:

def __init__(self, img_data, image_width, image_height):
    self.texture = glGenTextures(1)
    glBindTexture(GL_TEXTURE_2D, self.texture)
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT)
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST_MIPMAP_LINEAR)
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR)

    glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,image_width,image_height,0,...
                    GL_RGBA,GL_UNSIGNED_BYTE,img_data)
    glGenerateMipmap(GL_TEXTURE_2D)

def use(self):
    glActiveTexture(GL_TEXTURE0)
    glBindTexture(GL_TEXTURE_2D,self.texture)

def destroy(self):
    glDeleteTextures(1, (self.texture,))
```

`glGenTextures(1)` generates a new texture object and stores it in `texture`. We then tell openGL that we are working with this newly created texture object and that it is a 2D texture using `glBindTexture()`. `glTexParameteri()` is used to set parameters of the texture. In the code snippet we set it so that if the texture coordinates are outside [0, 1] for both x and y coordinates, it repeats. We also setup a mipmap so that textures that are further away have lower resolution and closer texture are higher res. For both cases, it is set so that it interpolates linearly between colors.

We then define the image using `glTexImage2D()` given the given image data. We also generate the mipmaps for the texture (the given image in increasingly smaller resolutions)

We can now create assets. Assets are created by linking a Mesh, Model and Material instance. When objects are called to be rendered, the Mesh's vertex array object stored in the Mesh instance is rebound using `glBindVertexArray()`, the linked Material uses the method: `material.use()` and the Model's transformation matrix is calculated and passed into the shader.

Now our scene looks like:



But it looks a bit weird - we can see the bottom of the black pieces. This is because we haven't setup depth testing. Depth testing works by looking at the depth buffer - a buffer that stores the depth for each drawn pixel when rendering. When the fragment wants to draw a pixel, it finds its depth and checks it against the depth buffer - only drawing the pixel if the depth is closer than on the depth buffer. Since we don't have depth testing yet, our scene looks weird because the bottom of those pieces are drawn last, putting them in front of the pieces. Depth testing was easily implemented:

```
glEnable(GL_DEPTH_TEST)
glDepthFunc(GL_LEQUAL)
```

`GL_LEQUAL` specifies that the drawn pixel's depth must be less than or equal to the depth in the depth buffer. This is used because I wanted to draw squares directly on the board which will be useful later. Now the scene looks like:

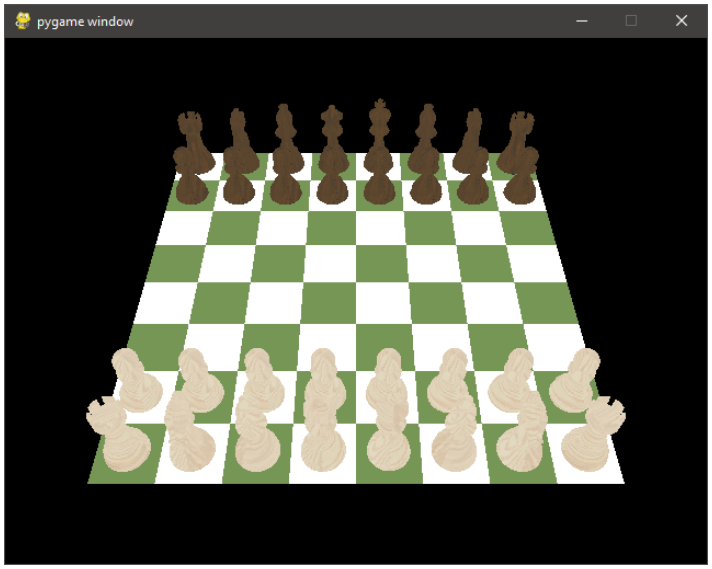


Figure 3: The scene with textures and depth-testing

But it's a bit hard to read the depth and contour of the pieces, especially when pieces are drawn on each other. To remedy this, we can add lighting.

Lighting

Lighting was implemented using Blinn-Phong shading without attenuation (the strength of the light doesn't drop off the further it travels). Blinn-Phong shading breaks up lighting into 3 components - ambient, diffuse and specular lighting and uses this to calculate the color of a fragment of the object. This takes into account the surface normal, the direction to the light source and the direction towards the camera.

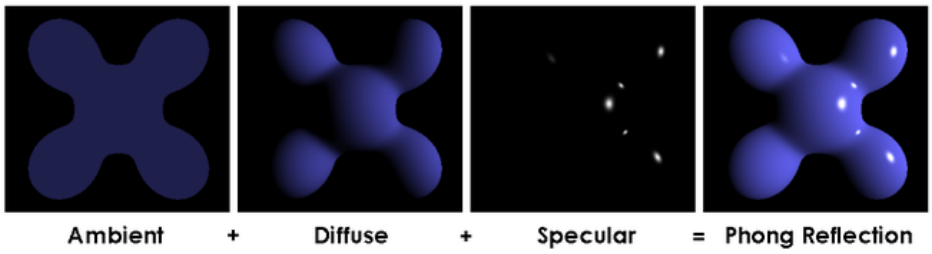


Figure 4: The 3 Components of Bling-Phong Shading [2]

The ambient light refers to the 'scattered' light of the scene, giving a constant level of illumination across the whole scene. This is replicated in the fragment shader by multiplying the texture color at a fragment's texture coordinate by 0.8.

Diffuse light is the light that is scattered by a surface and represents the parts of the object that catches the light. In the fragment shader this is calculated by multiplying the texture color at a fragment's texture coordinate by the dot product of the unit vectors of the fragments normal and the direction from the fragment to the light (vectors n and l in fig 5) which gives a value between 0 and 1 where 1 means that the surface is reflecting all the light and 0 meaning no light is reflected.

Specular light is the that is reflected off a surface and into the camera. This is used to make a surface look shiny. To calculate the specular light, the dot product of the half vector (the vector between the direction towards the camera and the light from the surface) and the surface normal is used (vectors h and n in fig 5). When the angle between these vectors is smaller, the more light reflects towards the camera. In the code this is calculated by multiplying the color of the fragment by the resultant dot product raised to some power which determines the surface's shininess. In my case I chose 32 as I didn't want my objects to be too shiny.

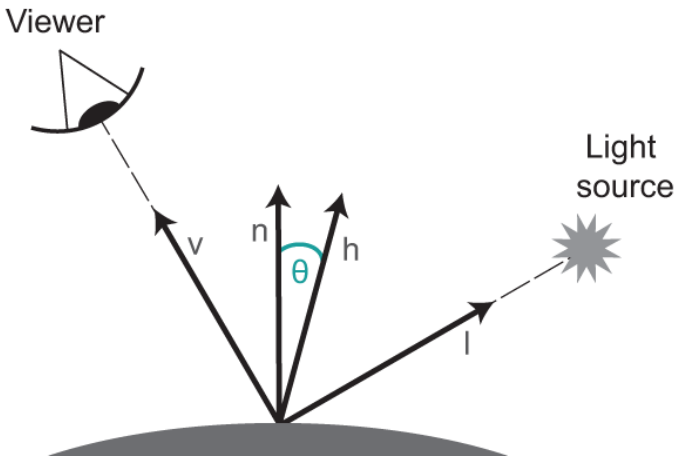


Figure 5: The Vectors from the surface. [1]

To get the final color of the fragment, you take the sum these components. All of this is calculated within the fragment shader.

In the scene, I didn't like how the lighting looked on the board as it reflected too much. To remove this, I removed the normal vectors from the quad mesh so that when calculated in the shader, it uses a normal of (0,0,0) so only the ambient lighting will show.

A point light was added above the center of the board and looks like:

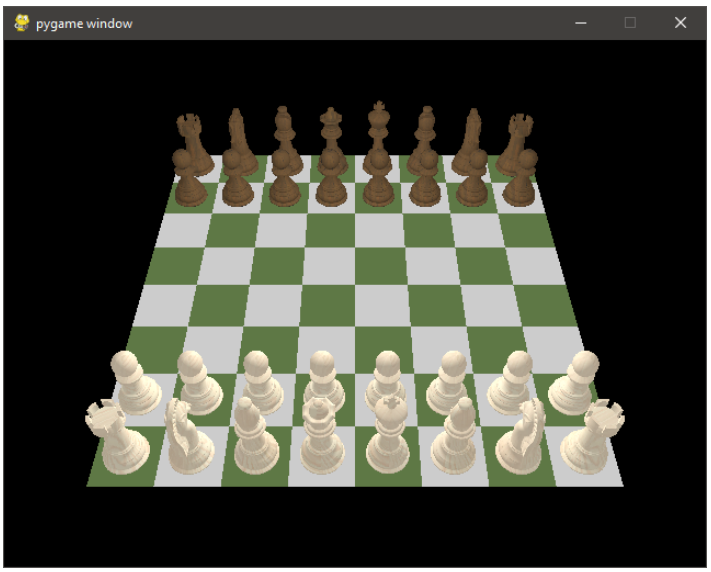


Figure 6: The scene with lighting

And as you can see, it is much easier to read and looks better too.

User Interaction

Now that we have the rendering setup, we can now implement the functionality of the program. A simple main menu was made using **pygame** so that you can select what mode you want - Go through a PGN file or play against a bot - and which side you want to view/play from.

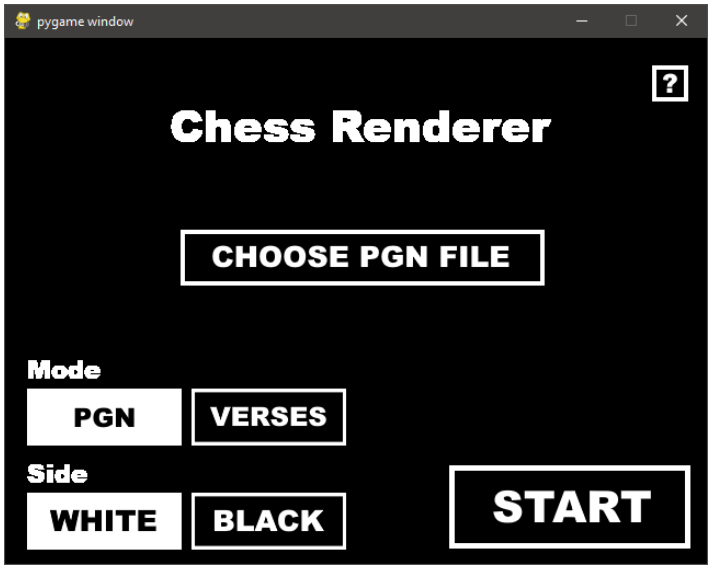


Figure 7: The scene with lighting

In order to read the PGN file, the Chess.pgn python library was used which has methods that allow you to access information from the given game. Using pygame's event handler, the arrow keys were bound so that when left/right is pressed, the program shows the previous/next move. A method was made in the scene class called 'setup_board' which removes all the pieces on the board and re-adds them according to the PGN file. This is called every time a move is made.

Animations were added next. When a move was made I wanted to the pieces to move towards its destination instead of just appearing there. To achieve this I created a method for the Piece Class (which is a sub-Class of Model) called animate:

```
def animate(self, from_square, to_square):
        self.frameCounter += 1
        from_pos = np.array(SQUARES[from_square] + (0,))
        to_pos = np.array(SQUARES[to_square] + (0,))
        self.position = ...
            tuple(from_pos + (to_pos - from_pos) / 7 * self.frameCounter)
        if self.frameCounter == 7:
            self.frameCounter = 0
            return True
        return False
```

`animate()` takes in the square in which the piece came from and the square it is going to. We then convert this to x, y coordinates in the world space and find the direction

from these squares by subtracting these coordinates. The position of the piece is then translated by this direction vector over 7 frames. This is run for every call in the game loop until the animation is finished.

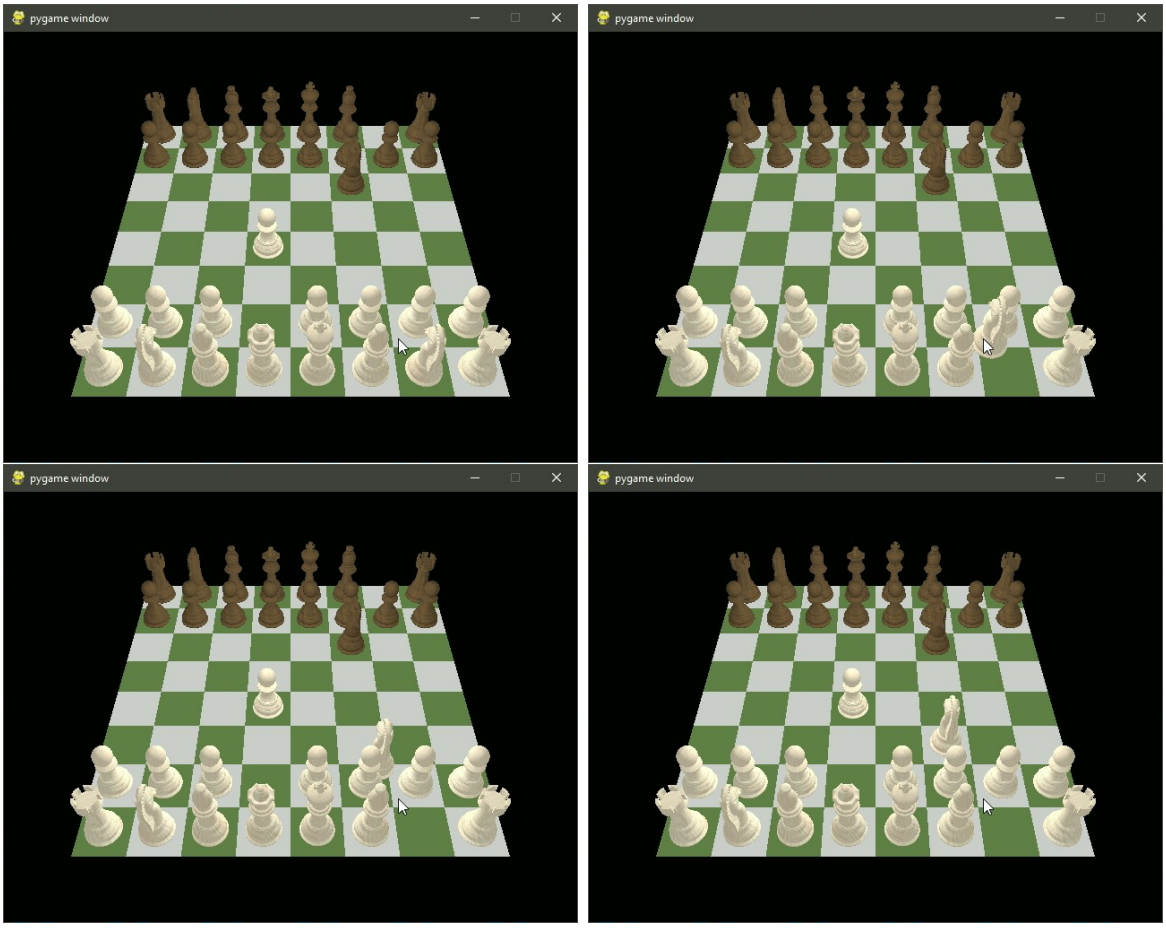


Figure 8: Animation of a move

Everything for the PGN mode has been implemented. Now it's time for the 'verses' mode. In order for the user to pick their moves, I implemented a function called `get_square_from_mouse()` which essentially reverses what was outlined in the 3D transformation pipeline section. `get_square_from_mouse()` takes in the mouses coordinates in the viewport and normalizes it. We then apply the inverse perspective projection matrix and inverse view matrix. This gives us a vector from the camera in the direction that was clicked on the screen. We then parameterize the path of this vector like:

$$X(t) = (p_1 + d_1 t, p_2 + d_2 t, p_3 + d_3 t)$$

where The direction vector is $D = (d_1, d_2, d_3)$ and the position of the camera is $P = (p_1, p_2, p_3)$

We then find the value of t that makes $p_3 + d_3 t = 0 \implies t = -\frac{p_3}{d_3}$. We then sub this into $X(t)$ to get the position on the ground in which the mouse clicked and find the square that this position is contained in. With a bit of logic, moves can be made by

clicking the piece you want to move and then clicking the desired square. The chess library was used to find the list of legal moves a piece can make. A new blue square model was added to show where these legal moves are on the board.

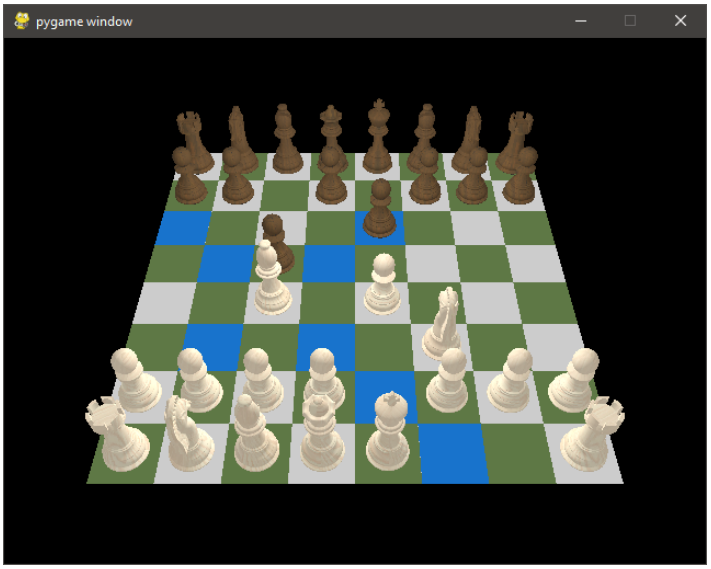


Figure 9: Making moves with screen clicking

The final part is implementing 'Stockfish' - a chess engine to play against. This was done using the Stockfish library setting the depth to 20 moves ahead and a skill level of 5. The computers move was chosen with the method `get_best_move()` and was called every time the player makes a move.

All the features were added, and we now have a fully functional chess program that allows you to go through a game given a PGN file and play against a bot.

Conclusion

Through the use of OpenGL in python, I have developed a program that renders a chess board in 3D with multiple ways of user interaction. In this project I have demonstrated important graphics techniques, including 3D rendering, lighting, and texture mapping as well as discussing them in the report.

Self-Assessment

In this project I have demonstrated a range of graphics techniques. I have demonstrated: creating the vertices list from an obj file, the techniques used to render a mesh in local space to a pixel on the viewport, texturing in OpenGL, lighting in OpenGL using shaders and converting a mouse click from the viewport to a vector from the camera. These methods were discussed in the report.

The report is concise, readable and accessible to people unfamiliar to OpenGL. Relevant tool and libraries were discussed - OpenGL, pyrr, pygame, chess.pgn and Stockfish.

Appropriate images were used to show the results of the different sections - e.g the scene with and without projection, textures and no textures, etc.

The goal of the report was clear - to make a chess program while explaining the key concepts in 3D rendering which I believe I have done effectively.

I believe that the project has the qualities of a 5 or better.

References

Bibliography

[1] Ariën, Geert. Ariën, 2017. URL: https://www.geertarien.com/blog/2017/08/30/blinn-phong-shading-using-webgl/.

[2] Brad Smith. Phong components version 4, 2006. URL: https://en.wikipedia.org/wiki/Phong_reflection_model#/media/File:Phong_components_version_4.png.