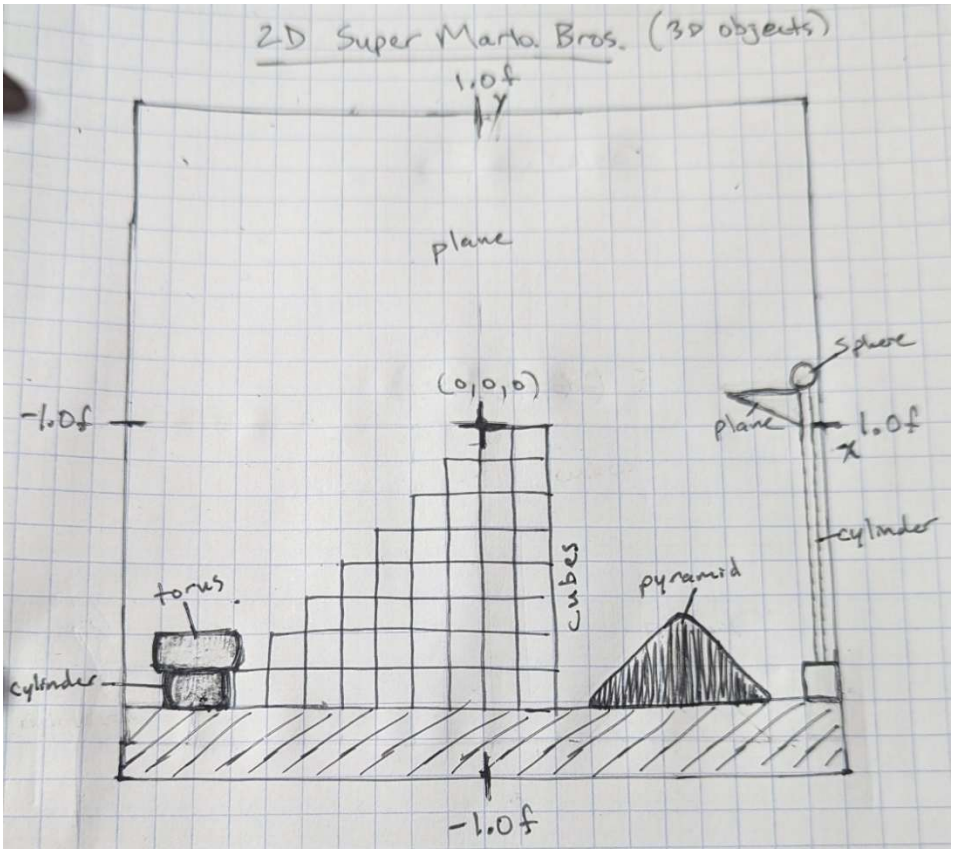
Super Mario Bros. (OpenGL/C++)

I recreated a 2D image of Super Mario Bros. (Nintendo) because it consists of a dynamic range of 3D primitive shapes, and I found it to be a very interesting piece to potentially recreate from a game I have loved since my early childhood. I recreated most objects seen in the scene, but decided to leave off the spherical flagpole top due to time constraints, and ended up using a total of four different types of 3D primitive shapes in the recreation of this image:

- cubes (staircase blocks and flagpole base block)
- planes (background, ground, and triangular flag)
- cylinders (pipe and flagpole)
- irregular pyramid (hill)

Recreating a 2D scene with this many shapes can easily become an overwhelming task, so it's important to actively manage the overall scope of the work. I sketched the image using graph paper on a 20 x 20 region consisting of 10 x 10 quadrants with the origin centered at (0,0), or (0,0,0) if considering a 3D space:



This made it a simple process of conceptualizing more quantitatively the components of the system and what 3D primitive shapes would be needed in the recreation process. Even more helpful was having a simple way of spatially organizing the elements of the image to be used

when calculating the coordinates of different vertices, when programming them using C++. With these dimensions, a single graph square has the 2D dimensions $0.1f \; x \; 0.1f$, and objects consisted of a up to a $0.2f$ depth. This was extremely convenient to know during development, although final sizes were somewhat different.

Mouse movement involved setting the yaw (y-axis rotation) and pitch (x-axis rotation); however, roll (z-axis rotation) was not used in this first-person view. The velocity (speed) can be adjusted by the mouse's scroll wheel if desired, and cutoff limits were used to keep the movement from going too fast (or from going below 0). The camera utilized the GLM (OpenGL Mathematics) *lookAt* function, which takes the position, position + front, and up parameters to determine the view at any given moment. The view matrix and projection matrix are passed to the GPU shader program once every frame, as well as model matrices for the objects before they are rendered. The user is also able to use the keyboard to strafe left and right or up and down and move the "eye" forward and backward. The calculation for the new position uses the velocity, which is based on the movement speed and the amount of time passed. Other keyboard controls are also available, such as changing from a 3D (perspective) to 2D (orthographic) projection and turning the ground boundary off and on.

Once I had most of my program complete, I ended up going through my code and cutting the amount of code in my main CPP file down from about 2000 lines of code to about 1600 lines of code. One significant contributor to this was modularizing the code that created and bound the Vertex Array Object (VAO), Vertex Buffer Object (VBO), and Element Buffer Object (EBO) by creating a function named *createVertexArray* that was called after creating the vertices and indices for each 3D object. I created a function for each 3D shape I needed in which a mesh from a custom Mesh struct was passed. This mesh holds all the VAO, VBO, EBO, vertex, and index data for each object. These functions make the code modular by allowing the class file to be included in other programs in which the functions can be used. The functions also made the code easier to understand and skim through and reduced the number of lines in my code by almost 20%! Many thanks to Learn OpenGL (https://learnopengl.com), as well as Professor Scott Gray and Samantha Chapman of Southern New Hampshire University (SNHU), as each played a part in my educational success during this term! Thank you for your time.

Matthew Pool

December 8, 2023