# 

# CS 330 Project Paper

Ahmet Mujanovic

ahmet.mujanovic@snhu.edu

Southern New Hampshire University

## 3D Scene Choices

The requirements for the project were to recreate a photograph as a 3-D scene using the OpenGL graphics API. Choosing the objects was admittedly done with little consideration for much other than what might be challenging to model yet simple enough for one who is new to the realm of 3D graphics programming. I did not want to include overly complex objects that had many different curves or small intricate details as keeping the scene to objects that could be made with simple primitives would put a greater emphasis on learning the core concepts of OpenGL and graphics programming in general, rather than the details of how to render something extremely complex. Including a few different objects that required multiple primitives ensured that there was at least some thought involved in how to approach the modeling process.

Having the scraper and plastic cement bottle made it interesting to try and fit multiple different shapes together, while the Rubik’s cube presented a challenge in regards to texturing a cube while it was being created through essentially 6 planes. The approach I had taken was binding a new texture each time a different face of the cube was created; however, I think the best approach may be to create one texture with all the faces and wrap it around the model with texture mapping.

## Navigating the Scene

Control of the scene is primarily done through the camera class with a camera object. This class defines the properties of the camera and also contains multiple methods that adjust the camera based on particular inputs from the user, either through the mouse cursor, mouse buttons, mouse scroll wheel, or keyboard. From there, callback functions in the global functions file will call these camera functions to make changes based on user input, with the help of GLFW functions to assign these callbacks to the active window. The cursor controls the orientation of the camera, the scroll wheel controls the speed the camera moves, and the WASDQE keys controls the movement of the camera.

An important point is that the view matrix is actually set to the camera position, rather than being fixed. In other words, as the inputs are made by the user and the position of the camera changes, the view matrix is updated according. Finally, although not relating directly to navigation but rather projection, if the user presses the “P” key, the projection matrix with switch between orthographic and perspective projections, with perspective being the default.

## Modularity and Organization

As the milestones and assignments progressed, some more time was put into creating code that was more organized and allowed for many of the components needed to render a scene to be abstracted away. This allowed for a more straightforward approach when creating new buffer arrays or generating new textures as the class that was setup to do so could simply be instantiated with the required arguments. Not only that, but the code was no longer packed altogether in the main function, which previously made making changes and navigating cumbersome.

The shader code was separated into a text document while a shader class was setup with multiple functions relating to using the shader code within the program. A function was added to parse the shader text files into strings usable by C++, while other functions served to compile, create, and link the shader with proper error checking present. Additionally, the shader objects created also contained member functions to activate, deactivate, and delete the shaders from memory when necessary.

Vertex buffer object, vertex array object, and element buffer object classes were all created with functionality to bind the respective components in order to efficiently set up VAOs for a particular set of vertices in a clear and easy to read way. These components can be bound, unbound, and deleted with a simple call of the succinct member function names, rather than calling the more verbose OpenGL functions each time.

Finally, as mentioned previously, the texture class compartmentalizes the creation of textures so that textures can be created by calling the constructor for the texture class with the proper parameters based on the image type and properties. Additionally, the texture class includes functions to bind, unbind, and delete texture objects, just as the shader and vertex related classes do.

One aspect that could be improved upon would be a mesh class to hold all of the components of an object, including its vertices, indices, and textures, rather than defining all of those components separately. This would greatly tidy up code in the main function and make creating additional meshes simpler with much of the functionality abstracted away.

## References

*Welcome to OpenGL*. LearnOpenGL. (n.d.). Retrieved from https://learnopengl.com/