**Program Two:** **The Cube**

Jordon Paynter

Department of Computer Science, Colorado State University Global

CSC 405: Graphics and Visualization

Professor Jennifer Marquez

December 10, 2023

Contents

[**Reflection** 3](#_Toc152850304)

[**Conclusion** 4](#_Toc152850305)

[**Steps taken to create the 3D cube** 5](#_Toc152850306)

[**Images of the Sierpinski Gasket** 8](#_Toc152850307)

[Cube Sides Green, Blue, Magenta 8](#_Toc152850308)

[Cube sides Green, Yellow, Cyan 9](#_Toc152850309)

[Cube sides Yellow, Cyan, and Red 10](#_Toc152850310)

[Cube **Starting camera position changed.** 11](#_Toc152850311)

[Cube mat4.perspective(projectionMatrix, (160 \* Math.PI) / 180, aspect, 0.1, 100); 12](#_Toc152850312)

[Cube increased near clipping to 5 and far clipping to 200 13](#_Toc152850313)

[Cube Near Clipping increased enough that when a corner points towards the camera, it doesn’t draw the corner. 14](#_Toc152850314)

[Cube Lowered the Far Clipping to 4 and doesn’t finish drawing the rest of it as it goes out of scope. 15](#_Toc152850315)

## **Reflection**

Reflecting on this exercise in creating a 3D rotating cube using WebGL/OpenGL, several key learning points and insights emerge, highlighting the intricacies and power of 3D graphics programming. The exercise underscored the importance of understanding the foundational elements of WebGL/OpenGL - vertices, shaders, buffers, and transformations. Vertices form the backbone of any 3D object, defining its shape and structure. Defining and manipulating these vertices was crucial for shaping the cube. Shaders, comprising vertex and fragment shaders, played a pivotal role in bringing the cube to life. The vertex shader’s handling of vertex positions and transformations was key to placing and rotating the cube in 3D space. In contrast, the fragment shader’s role in coloring illuminated the importance of pixel processing for rendering. Buffers, the bridge between JavaScript and the GPU, emphasized the importance of efficient data management. Through this exercise, the practical application of these concepts provided a much deeper understanding than theoretical study alone could offer.

The exercise also reinforced the significance of transformations in 3D rendering. Implementing rotation transformations to animate the cube made the scene dynamic and provided a deeper grasp of how objects behave in 3D space. This aspect of the project brought to light the complexities of matrix operations and their critical role in 3D graphics. Experimenting with different transformation matrices offered practical insights into how objects can be scaled, translated, and rotated, revealing the nuances of 3D object manipulation.

Finally, this exercise was an invaluable lesson in problem-solving and debugging within graphics programming. 3D graphics programming, particularly with a low-level API like WebGL, involves a steep learning curve, where even small mistakes can lead to significant issues. Debugging shader code, rectifying incorrect transformations, and ensuring the proper setup of buffers required meticulous attention to detail and persistence. The process was challenging but immensely rewarding, as each solved problem deepened my understanding of WebGL and bolstered my problem-solving skills.

### **Conclusion**

Creating a 3D rotating cube in WebGL was a comprehensive and enlightening journey through the core aspects of 3D graphics programming. It provided practical experience in graphics programming concepts, emphasized the importance of mathematical transformations in 3D space, and honed problem-solving and debugging skills. The knowledge and skills gained from this exercise are invaluable and form a solid foundation for further exploration in computer graphics.

## **Steps taken to create the 3D cube**

1. **Initialize WebGL Context:**
   * Retrieve the canvas element from the HTML document.
   * Initialize the WebGL context from the canvas.
2. **Create Shader Programs:**
   * Write the vertex shader code to handle vertex positions and transformations.
   * Write the fragment shader code to handle the coloring of the pixels.
   * Compile the shaders and create a shader program by attaching and linking the shaders.
3. **Define Cube Geometry:**
   * Define vertices for the cube. Duplicate vertices for each face to assign unique colors later.
   * Define the indices to instruct WebGL on constructing the cube’s faces using the vertices.
4. **Create Color Data:**
   * Define a color array with RGBA values, ensuring each face of the cube has a unique color. Associate each group of color values with the corresponding vertices of each face.
5. **Buffer Setup:**
   * Create and bind vertex buffer objects for vertices and colors.
   * Transfer the vertices and color data into their respective buffers.
   * Create and bind an element buffer object for the indices and transfer the index data.
6. **Configure Shaders:**
   * Specify how to retrieve vertex and color data from the buffers using attribute pointers.
   * Enable the vertex attribute arrays to use the provided data.
7. **Set Up Matrices for Transformations:**
   * Define model, view, and projection matrices.
   * Implement rotation transformations on the model matrix to achieve the 3D rotating effect.
8. **Rendering Loop:**
   * Clear the canvas and depth buffer.
   * Use the shader program.
   * Bind the buffers and set attribute pointers as needed.
   * Update transformation matrices and pass them to the shader program.
   * Draw the cube using **gl.drawElements**.
   * Continuously call the render function using **requestAnimationFrame** to keep the cube rotating.
9. **Depth Testing:**
   * Enable depth testing to ensure proper rendering of 3D objects based on their Z-coordinates.

# **Images of the 3D Cube**

**Figure 1a.**

Cube Sides Green, Blue, Magenta

A screenshot of a computer

Description automatically generated

**Figure 1b.**

Cube sides Green, Yellow, Cyan

A screenshot of a computer

Description automatically generated

**Figure 1c.**

Cube sides Yellow, Cyan, and Red

A screenshot of a computer

Description automatically generated

**Figure 1d.**

Cube **Starting camera position changed.**

**A screenshot of a computer

Description automatically generated**

**Figure 1e.**

## Cube mat4.perspective(projectionMatrix, (160 \* Math.PI) / 180, aspect, 0.1, 100);

A screenshot of a computer

Description automatically generated

**Figure 1f.**

## Cube increased near clipping to 5, allowing to see the far side through the inside of the cube.

A colorful cube with red blue and green squares

Description automatically generated

**Figure 1g.**

## Cube Near Clipping increased enough that when a corner points towards the camera, it doesn’t draw the corner.

**A colorful cube with red and blue center

Description automatically generated**

**Figure 1h.**

## Cube lowered the Far Clipping to 5 and didn’t finish drawing the rest as it went out of scope.

**A screenshot of a computer

Description automatically generated**