**Portfolio Project: Rotating Cube WebGL**

Alejandro Ricciardi

Colorado State University Global

CSC405: Graphics and Visualization

Professor: Dr Jennifer Marquez

October 06, 2024

**Portfolio Project: Rotating Cube WebGL**

This reflection is part of the Module 8 Portfolio Project from CSC405: Graphics and Visualization at Colorado State University Global course. It provides a paragraph discussion about the Hidden-Surface Removal problem, as well as an overview and reflection on the program's functionality. It also describes the steps I took to create an interactive 3D rotating cube in WebGL using triangles primitives, the Blinn-Phong lighting model, orthographic and perspective projections, and the Painter's Algorithm for Hidden Surface Removal (HSR)

The program is titled "Rotating Cube”, and it is coded using WebGL-GLSL 3 (OpenGL Shading Language), JavaScript, and HTML. A video showcasing the program functionality can be found here:.

**Hidden-Surface Removal**

In computer graphics, Hidden-Surface Removal (HSR) is the process of determining which objects in a 3D scene are visible to the camera and which are not visible as they are behind other objects (Angel & Shreiner). HSR is crucial for rendering realistic scenes, more specifically 3D animations. Without it, 3D shapes will incorrectly overlap, the entire 3D scene will have a cluttered and chaotic appearance, viewers will not be able to tell distances between objects, and both visible and hidden surfaces will be rendered affecting performance. These issues can be handled by various HSR algorithms, each with advantages and disadvantages. The algorithms can be categorized into two approaches, the object-space and image-space approaches. The object-space approach compares polygons pairwise, using a depth parameter to determine the visibility of each object relative to the camera view and each other. This approach has a complexity of *O(k2)*, where *k* is the number of polygons, making it inefficient for scenes with many polygons. On the other hand, the image-space approach casts a ray through each object’s pixels in the scene to determine which objects are closer to the camera. This approach has a complexity of *O(k)* offering better efficiency than the object-space approach for scenes with many polygons; however, this approach had significant overhead due to the need to compute each pixel visibility. Whatever approach and associated algorithm are used, implementing HSR is essential for the realistic and accurate rendering of objects in 3D scenes.

**The Painter’s Algorithm**

In my program, the Painter’s Algorithm is implemented to address the HSR problem. The Painter’s Algorithm has an object-space approach to HSR, it mimics how a painter would create a painting by applying the background first and then applying layer upon layer until the full image is painted (Brown, 2018). The major drawback of implementing this algorithm is that each primitive triangle forming the cube in my program must be sorted using the depth parameter to determine its visibility, and this sorting needs to be done before rendering each frame. This a suitable for my program because the scene has few polygons or triangles having a minor impact on performance. Another issue raised by the implementation of Painter’s Algorithm is triangle intersection. In other words, when triangles intercept the sorting does not necessarily determine the correct triangle order, and the triangles may be rendered in the wrong order. This can be addressed by dividing the intersecting triangles into smaller triangles; this is not necessary for my program as no triangles are intersecting.

**The Painter’s Algorithm Implementation and WebGL Z-Buffer Algorithm**

WebGL does handle automatically HSR. However, it can be implemented explicitly by activating WebGL built-in Z-buffer algorithm using the following code line:

JavaScript

gl.enable(gl.DEPTH\_TEST); // Enable depth testing to handle overlapping geometry

Before rendering, the color buffer and the Z-buffer need to be cleared, and this can be done by using the following code line:

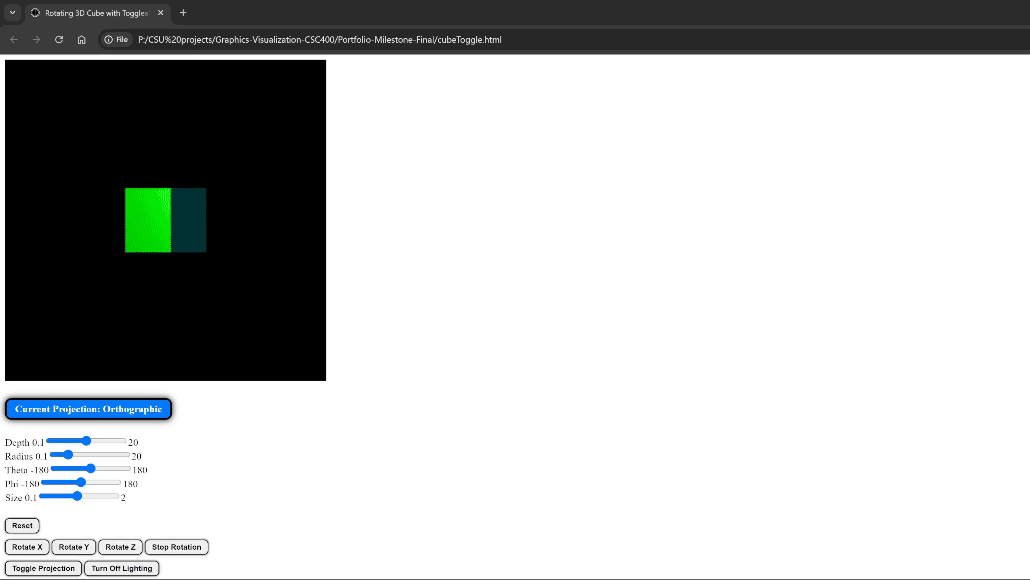
Javascript (in the render function)

gl.clear(gl.COLOR\_BUFFER\_BIT | gl.DEPTH\_BUFFER\_BIT); // Clear the screen

The Z-buffer Algorithm has an object-space approach to HSR, it maintains a Z-buffer that stores depth information, or z-values, for each pixel in the scene, determining which objects are the closest to the camera. The algorithm is widely used as it has significant advantages over the Painter’s Algorithm as it is more efficient at rendering 3D scenes with a large number of polygons.

**Figure 1**

*No HSR Implemented*



*Note:* My Program with no HSR algorithm implemented.

The program showcased in this paper is the second version of my portfolio milestone program, before implementing the Painter’s Algorithm it was necessary to remove the implementation of the WebGL Z-buffer Algorithm. This was done by removing the following code line from the program:

JavaScript

gl.enable(gl.DEPTH\_TEST); // Enable depth testing to handle overlapping geometry

This removes the WebGL Z-buffer Algorithm from the program, and the following code line was modified from:

Javascript (in the render function)

gl.clear(gl.COLOR\_BUFFER\_BIT | gl.DEPTH\_BUFFER\_BIT); // Clear the screen

to:

Javascript (in the render function)

gl.clear(gl.COLOR\_BUFFER\_BIT); // Clear the screen

as there is no need to clear the z-buffer anymore (gl.DEPTH\_BUFFER\_BIT). Figure 1 is the result of removing the Z-buffer Algorithm without implementing an algorithm substitute, where it can be observed the cube’s faces overlap incorrectly. The next step was the implementation of the Painters Algorithm, by first declaring a variable to store the faces:

Javascript

var faces = [];    // Array to store the cube's faces with their associated data

(indices, color, normal)

Note that the faces are composed of primitive triangles:

Javascript (in the render function)

// Form two triangles for each face from the four vertex indices (quad -> 2 triangles)

var indices = face.indices;

After declaring the ‘Faces’ array I implemented a function to build the faces see code lines below:

Javascript

 /\*\*

 \* Builds the faces array, storing face data including vertex indices, color,

\* normal, and positions.

 \* This function is called whenever the cube's size changes.

 \*/

function buildFaces() {

    // Empty the faces array to be filed by the new faces data to be rendered

    faces = [];

    for (var i = 0; i < faceIndices.length; i++) {

        var indices = faceIndices[i];

        // Compute the face normal

        var t1 = subtract(vertices[indices[1]], vertices[indices[0]]);

        var t2 = subtract(vertices[indices[2]], vertices[indices[1]]);

        var normal = normalize(cross(t1, t2));

        normal = vec3(normal);

        // Store face data

        var face = {

            indices: indices,

            color: faceColors[i],

            normal: normal,

            depth: 0,

            positions: [

                vertices[indices[0]],

                vertices[indices[1]],

                vertices[indices[2]],

                vertices[indices[3]]

            ]

        };

        faces.push(face);

    }

}

In the render function, I started implementing the Painter's Algorithm by creating a loop that computes the depth for each face see code below:

Javascript (in the render function)

//------ Compute depth for each face

for (var i = 0; i < faces.length; i++) {

    var face = faces[i];

    var depthSum = 0;

    // Iterate over each vertex of the face

    for (var j = 0; j < face.positions.length; j++) {

        // Apply the model-view matrix to transform the vertex to camera space

        var transformedVertex = mult(modelViewMatrix, face.positions[j]);

        // Sum up the Z-coordinates (depth) of the transformed vertices

        depthSum += transformedVertex[2]; // Z-coordinate in camera space

    }

    // Store the average Z-depth of the face (used for sorting)

    face.depth = depthSum / 4; // Average depth of the face (since each face has

4 vertices)

}

Next, I created a loop to sort the faces based on the average depth of each face, see the code below:

Javascript (in the render function)

//------- Sort faces based on depth (from farthest to nearest)

faces.sort(function(a, b) {

    // Sort faces by depth to implement back-to-front rendering (Painter's Algorithm)

    return a.depth - b.depth; // Faces with higher Z-values (farther) are rendered first

});

Finally, I implemented the rendering of the faces themselves. This was done in a for loop that iterates through the Faces array, see the code below:

Javascript (in the render function)

//------- Render faces in sorted order (back-to-front)

for (var i = 0; i < faces.length; i++) {

    var face = faces[i];

    // Prepare arrays for storing the vertex positions, colors, and normals of

the face

    var facePositions = [];

    var faceColors = [];

    var faceNormals = [];

    // Form two triangles for each face from the four vertex indices (quad -> 2

triangles)

    var indices = face.indices;

    // Create 6 indices (two triangles) from the 4 vertices of the face

    var idx = [indices[0], indices[1], indices[2], indices[0], indices[2],

indices[3]];

    // Loop through the indices and populate the positions, colors, and normals

arrays

    for (var j = 0; j < idx.length; j++) {

        facePositions.push(vertices[idx[j]]); // Add the vertex positions

        faceColors.push(face.color);          // Assign the color of the face

        faceNormals.push(face.normal);        // Assign the face's normal vector

for lighting

    }

    // Bind and fill the position buffer with vertex data for the current face

    gl.bindBuffer(gl.ARRAY\_BUFFER, vBufferId);

    gl.bufferData(gl.ARRAY\_BUFFER, flatten(facePositions), gl.STATIC\_DRAW);

// Set attribute pointer

    gl.vertexAttribPointer(aPositionLoc, 4, gl.FLOAT, false, 0, 0);

    // Bind and fill the color buffer with color data for the current face

    gl.bindBuffer(gl.ARRAY\_BUFFER, cBufferId);

    gl.bufferData(gl.ARRAY\_BUFFER, flatten(faceColors), gl.STATIC\_DRAW);

    gl.vertexAttribPointer(aColorLoc, 4, gl.FLOAT, false, 0, 0); // Set attribute

pointer

    // Bind and fill the normal buffer with normal vectors for the current face

    gl.bindBuffer(gl.ARRAY\_BUFFER, nBufferId);

    gl.bufferData(gl.ARRAY\_BUFFER, flatten(faceNormals), gl.STATIC\_DRAW);

// Set attribute pointer

    gl.vertexAttribPointer(aNormalLoc, 3, gl.FLOAT, false, 0, 0);

    // Send the model-view matrix and the normal matrix to the vertex shader

    gl.uniformMatrix4fv(modelViewMatrixLoc, false, flatten(modelViewMatrix));

    gl.uniformMatrix3fv(normalMatrixLoc, false, flatten(normalMatrix));

    // Render the face as 2 triangles (6 vertices total) using the drawArrays

function

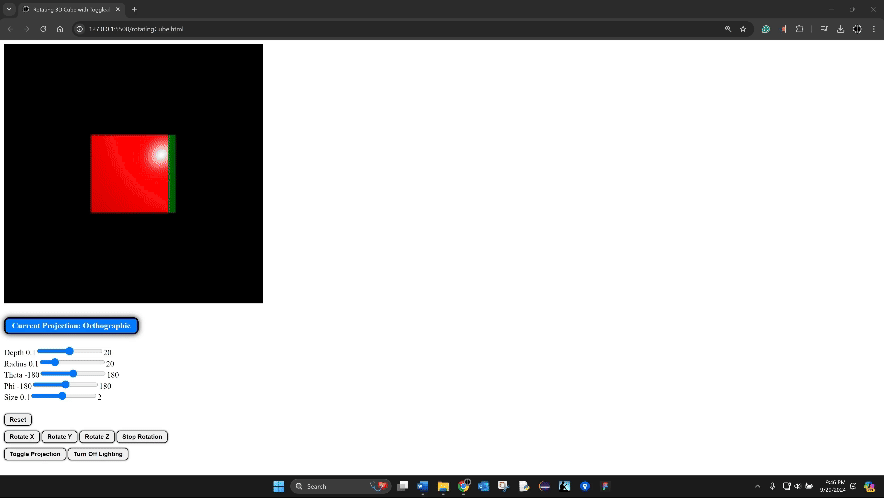
    gl.drawArrays(gl.TRIANGLES, 0, 6);

}

The final result of the Painter’s Algorithm implementation is that the cube's faces render correctly without overlapping; see Figure 2.

**Figure 2**

*Painter’s Algorithm Implemented*



*Note:* This illustrates the result of implementing the Painter’s Algorithm into my program, eliminating face overlapping. The size of the cube was also changed from the previous illustration.

**Added Functionalities**

This program is the third version of the portfolio milestone assignment from the course module 5. The first version of the program is an implementation of an interactive viewer with an orthographic projection of a 3D rotating cube. Where the user can rotate the cube along the x, y, and z axes, stop the rotation, and reset all parameters using buttons. Additionally, the user can resize the cube using a slider, as well as control the interactive viewer depth, radius, theta angle, and phi angle with sliders. The code for this assignment can be found here:…. A video showcasing the project can be found here:…. The figures below showcases the different functionality of the program.

**Figure 3**

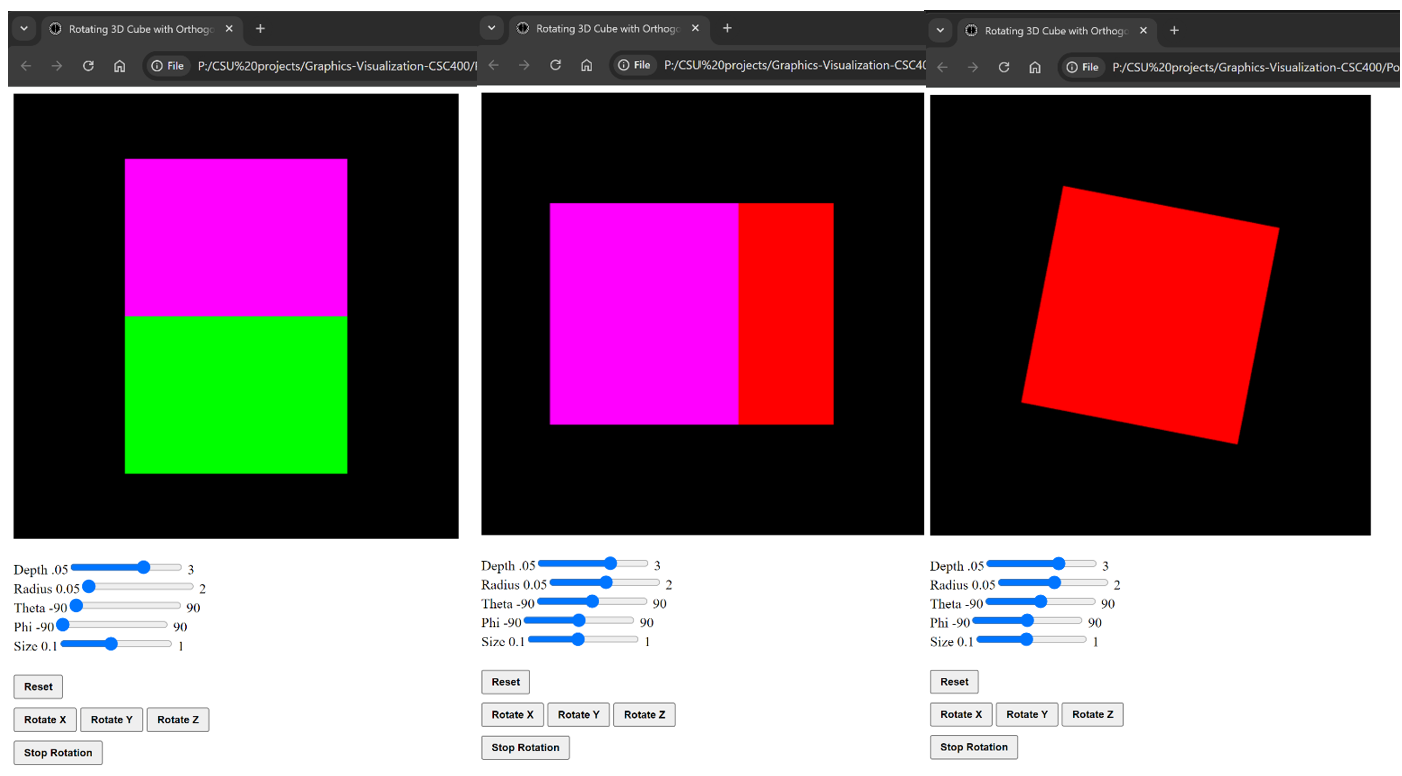
*A screenshot of a computer screen

Description automatically generatedPortfolio Milestone First Version*

*Note:* the program implements an orthographic projection of a 3D rotating cube without lighting.

**Figure 4**

*Portfolio Milestone First Version Rotating X, Y, and Z*



*Note:* the figure illustrates the cube rotating about the x, y, and z axes respectively.

The second version of the program was the first step in this final portfolio assignment. In this version, the toggled function between orthographic and perspective projections, as well as an interactive Blinn-Phong lighting toggled between on and off state were added to the interactive viewer. The code for this version can be found here: cubeToggle. The figures below illustrate the toggle functionality between projections and the off-and-on lighting functionality.

**Figure 5**

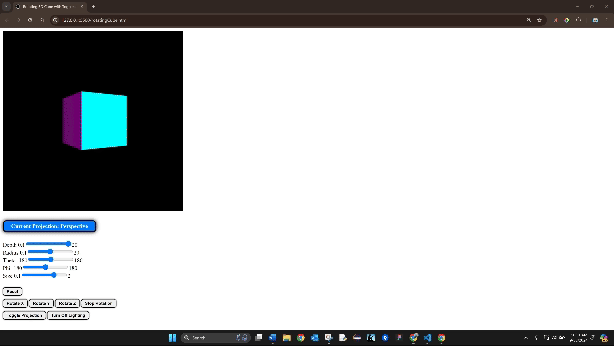
*Lighting On-and-Off*

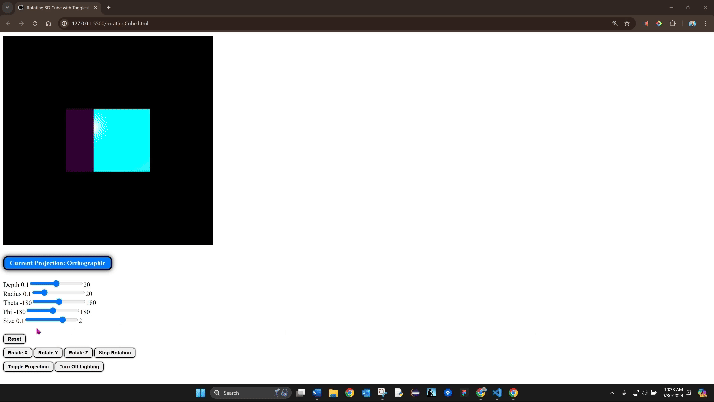
A screenshot of a computer

Description automatically generated

*Note:* the figure illustrates the lighting turned on and off by using the buttons ‘Turn Off Lighting’ and ‘Turn On Lighting’ functionality.

**Figure 6**

*Toggle Projections and Radius Slider*



*Note:* the figure illustrates the functionalities of the “Radius’ slider that controls how far or how near the camera is from the cube and “Toggle Projections” which toggles between orthographic projection, on the left, and perspective projection, on the right, showcasing the visual differences between the two projections. In my video,…, I explain the difference between the two projections, and in my video, … , I explain in detailed viewing and the orthographic projection.

**References**

Angel, E., & Shreiner, D. (2020). Chapter 12.6: Hidden-surface removal. *Interactive computer graphics. 8th edition*. Pearson Education, Inc. ISBN: 9780135258262

Brown, W. (2018 March 24). Chapter 12.2: Hidden surface removal. *Learn computer graphics using WebGL*. https://www.webgl.brown37.net/12\_advanced\_rendering/02\_hidden\_surface\_removal.html

Ricciardi A. (2024, August 22). *Interactive 3D recursive approximated sphere in WebGL* [Video]. YouTube. https://www.youtube.com/watch?v=Rp3mV8I62QE