Reflection:

Module-6 Critical Thinking – Interactive Recursively Approximated Sphere

Alejandro Ricciardi

Colorado State University Global

CSC405: Graphics and Visualization

Professor: Dr Jennifer Marquez

September 22, 2024

Reflection:

Module-6 Critical Thinking – Interactive Recursively Approximated Sphere

This reflection is part of Module 6 Critical Thinking – Approximating a Sphere Recursively from CSC405: Graphics and Visualization at Colorado State University Global. It provides an overview and reflection on the program's functionality and output screenshots. As well as the steps I took to create an interactive approximated recursively sphere, an explanation of the role of shaders, buffers, and transformations in achieving the final result, and I reflect on what I have learned through this exercise. The program is titled " Interactive Approximated Recursively Sphere" and it is based on WebGL and is coded using GLES 3, JavaScript, and HTML. It is a very simple WebGL application that generates and displays an interactive 3D approximated sphere in WebGL. The sphere is created by recursively subdividing a tetrahedron. Users can control the sphere's radius, rotation (theta and phi angles), and the number of subdivisions using sliders. The program also supports pausing and resuming the rotation. This program visits the concepts of transformation, lighting, and shading in computer graphics, more specifically rotation, orthogonal projection, and rotation. Additionally, the model view and the light components are the ones experiencing the rotation, not the sphere object. A video showcasing the program functionality can be found here: Interactive Rotating Recursively Approximated 3D Sphere - WebGL.

Viewing

The program uses a parallel projection, more specifically an orthographic projection to view the sphere. In an orthographic projection, the size and shape of objects remain consistent on the projection plane without perspective distortion without perspective distortion. However, the program implements a perspective effect simulation where the size of the sphere dynamically changes as the camera view moves closer to the object or further away from it mimicking a perspective view without distorting the sphere size proportion or shape, as a true perspective projection would. Below is a camera view code snippet found in the JavaScript file of the program.

Code Snippet from ‘render’ Function

```
// Compute the camera's position using spherical coordinates.
// The camera's position (eye) is determined by `theta` (horizontal angle), `phi` (vertical
angle),
// and the distance from the origin (`radius`).
eye = vec3(
    radius * Math.sin(theta) * Math.cos(phi),  // X-coordinate of the camera position
    radius * Math.sin(theta) * Math.sin(phi),  // Y-coordinate of the camera position
    radius * Math.cos(theta)                   // Z-coordinate of the camera position
);

// Adjust the size of the orthographic projection view volume based on the radius of the
sphere
var viewSize = 1.5 * radius;
left = -viewSize;
right = viewSize;
bottom = -viewSize;
top_bound = viewSize;
near = -7 * radius;  // Set near clipping plane based on the radius
far = 10 * radius;   // Set far clipping plane based on the radius

// Compute the orthographic projection matrix (view volume) based on the updated view size
projectionMatrix = ortho(left, right, bottom, top_bound, near, far);
```

Note that orthographic projection is dependent on the ‘left’, ‘right’, ‘bottom’, top_bound’, ‘near’, and ‘far’ variables, and the ‘left’, ‘right’, ‘bottom’, and ‘top_bound’ variables are dependent on the ‘viewSize” which is dependent on the ‘radius’ variable as the ‘near’ and ‘far’ variables are. In other words, the variable ‘viewSize’ is used to dynamically adjust the view volume. This allows the simulation of a perspective projection by updating the camera position and view volume based on the distance ‘radius’ without a size proportion or shape distortion that is associated with a perspective projection.

Transformations

The camera view or model experiences a rotation along with the lighting component, not the sphere object itself. The rotation follows a circular path around the y-axis, creating the effect of the camera moving around the sphere. Below is a rotation code snippet found in the JavaScript file of the program.

Code Snippet from ‘render’ Function

```
// Update theta and phi angles for horizontal and vertical rotation (if rotation is not
paused)
// This means that the camera is rotating, not the sphere
if (!rotationPaused) {
    theta += 0.01;  // Rotate around the x-axis relative to the angle (theta)
    // phi += 0.005;  // Uncomment to enable vertical rotation (phi) - Rotate around the
                                                                                    xz-plane
}

// Compute the camera's position using spherical coordinates.
// The camera's position (eye) is determined by `theta` (horizontal angle), `phi` (vertical
angle),
// and the distance from the origin (`radius`).
eye = vec3(
    radius * Math.sin(theta) * Math.cos(phi),  // X-coordinate of the camera position
    radius * Math.sin(theta) * Math.sin(phi),  // Y-coordinate of the camera position
    radius * Math.cos(theta)                   // Z-coordinate of the camera position
);

// Compute the model-view matrix using the camera's position (eye), the point to look at (at),
// and the up direction (up). This matrix transforms object coordinates to camera (eye)
coordinates.
modelViewMatrix = lookAt(eye, at, up);

// Compute the normal matrix for transforming normals, which is the transpose of the inverse
of the model-view matrix
nMatrix = normalMatrix(modelViewMatrix, true);

// Pass the computed model-view and normal matrices to the vertex shader
gl.uniformMatrix4fv(modelViewMatrixLoc, false, flatten(modelViewMatrix));
gl.uniformMatrix3fv(nMatrixLoc, false, flatten(nMatrix));
```

Note that the angle theta controls the rotation around the y-axis and the phy angle the rotation around the xz-plane if uncommented. If the ‘rotationPaused’ is true, it will pause the sphere rotation; this part of the program functionality where the user can stop or resume the rotation of the sphere. Additionally, the program implements a translation transformation where the camera and the lighting component move along the y-axis and relative to the x- and z-axes, based on the theta and phi angles, respectively. Below is a code snippet found in the JavaScript file of the program related to the translation transformation.

Code Snippet from ‘render’ Function

```
// Compute the camera's position using spherical coordinates.
// The camera's position (eye) is determined by `theta` (horizontal angle), `phi` (vertical
angle),
// and the distance from the origin (`radius`).
eye = vec3(
    radius * Math.sin(theta) * Math.cos(phi),  // X-coordinate of the camera position
    radius * Math.sin(theta) * Math.sin(phi),  // Y-coordinate of the camera position
    radius * Math.cos(theta)                   // Z-coordinate of the camera position
);

// Compute the model-view matrix using the camera's position (eye), the point to look at (at),
// and the up direction (up). This matrix transforms object coordinates to camera (eye)
coordinates.
modelViewMatrix = lookAt(eye, at, up);
```

Note that the translations of the camera’s position along the x-, y-, and z-axes are controlled by spherical coordinates using the theta and phi angles, and the motions are circular not in a straight line. As these angles change, the camera and lighting components are repositioned. Below is a code snippet found in the HTML file of the program related to the user control over the transformations.

```
<!--
            Sliders for controlling the sphere's properties
        -->
        <!-- Camera and light -->
        <div class="control-container">
            <label for="radiusSlider">Radius:</label> <!-- Camera and light distance from the
                                                          sphere-->
            <input type="range" id="radiusSlider" min="0.1" max="5.0" step="0.01" value="1.5">
        </div>
        <div class="control-container"> <!-- camera along the y-axis -->
            <label for="thetaSlider">Theta (°):</label>
            <input type="range" id="thetaSlider" min="-180" max="180" step="1" value="0">
        </div>
        <div class="control-container"><!-- camera along the xz-plane -->
            <label for="phiSlider">Phi (°):</label>
            <input type="range" id="phiSlider" min="-180" max="180" step="1" value="0">
        </div>
        <!-- geometry of the sphere -->
        <div class="control-container"> <!-- number of subdivisions -->
            <label for="subdivisionSlider">Subdivisions:</label>
            <input type="range" id="subdivisionSlider" min="0" max="7" step="1" value="3">
        </div>
        <!-- Rotation -->
        <div class="control-container"><!-- Buttons to pause and resume sphere rotation -->
            <button id="Pause">Pause Rotation</button>
            <button id="Resume">Resume Rotation</button>
        </div>
```

Note that the user can control the translation transformation using sliders and pause or resume the rotation using buttons.

Lighting

To simulate the lighting component the program uses the Blinn-Phong model, where the lighting is calculated using three components: ambient, diffuse, and specular. Note that:

- Ambient reflection represents indirect lighting, bouncing light, or general brightness.

- Specular reflection is related to the specular surfaces where the light is reflected in a specific direction, resulting in the surface looking shiny.
- Diffuse reflection is related to the diffuse surfaces where the light is scattering in all directions.

The code snippets below from the JavaScript file are where the different lighting components are initialized and configured.

Code Snippet from ‘interactiveSphere’ Function Variable Initialization Section

```
// ----------------------------- Light ----------------------------

    // Arrays to hold lighting properties for each triangle
    var ambientProducts = []; // Store the ambient lighting component for each triangle
    var diffuseProducts = []; // Store the diffuse lighting component for each triangle
    var specularProducts = []; // Store the specular lighting component for each triangle

    // Light source properties
    var lightPosition = vec4(1.0, 1.0, 1.0, 0.0);  // Directional light (w == 0.0)
    var lightAmbient = vec4(0.2, 0.2, 0.2, 1.0);   // Ambient light color
    var lightDiffuse = vec4(1.0, 1.0, 1.0, 1.0);   // Diffuse light color
    var lightSpecular = vec4(1.0, 1.0, 1.0, 1.0);  // Specular light color

    // Shininess coefficient for specular reflection
    var materialShininess = 50.0;

    // Arrays of material properties for ambient, diffuse, and specular lighting
    var materialAmbientArray = [
        vec4(0.0, 0.0, 0.4, 1.0),  // Blue
        vec4(0.4, 0.0, 0.0, 1.0),  // Red
        vec4(0.0, 0.4, 0.0, 1.0),  // Green
        vec4(0.4, 0.4, 0.0, 1.0)   // Yellow
    ];
    var materialDiffuseArray = [
        vec4(0.0, 1.0, 1.0, 1.0),  // Cyan
        vec4(1.0, 0.0, 0.0, 1.0),  // Red
        vec4(0.0, 1.0, 0.0, 1.0),  // Green
        vec4(1.0, 1.0, 0.0, 1.0)   // Yellow
    ];
    var materialSpecularArray = [
        vec4(1.0, 1.0, 1.0, 1.0),  // White (for shininess)
        vec4(1.0, 1.0, 1.0, 1.0),
        vec4(1.0, 1.0, 1.0, 1.0),
        vec4(1.0, 1.0, 1.0, 1.0)
    ];
```

Note that the light position is set as a directional light (w == 0.0), and the arrays for ambient, diffuse, and specular properties are used to assign material-specific lighting to each triangle in the sphere. The shininess coefficient is used to compute how sharp or smooth the specular reflection should appear. Each triangle is then rendered with its calculated lighting properties. Below is a code snippet of the ‘triangle’ from the JavaScript file where the lighting components are configured.

Code Snippet from ‘triangle’ Function

```
function triangle(a, b, c) {
        // Compute the normal vector for the triangle
        var t1 = subtract(b, a);  // Vector from a to b
        var t2 = subtract(c, a);  // Vector from a to c
        var normal = normalize(cross(t2, t1));  // Normal vector (perpendicular to the
                                                   surface)
        // Store normals for each vertex
        // a normal (or normal vector) is a vector that is perpendicular to the shape, here
           triangles.
        // Normals detremine direction of the shape facing a light, it determines how the
           light interacts with surfaces
        normalsArray.push(vec4(normal[0], normal[1], normal[2], 0.0)); // Normal for the first
                                                                         vertex of the
                                                                         triangle
        normalsArray.push(vec4(normal[0], normal[1], normal[2], 0.0)); // Normal for the
                                                                         second vertex of the
                                                                         triangle
        normalsArray.push(vec4(normal[0], normal[1], normal[2], 0.0)); // Normal for the third
                                                                         vertex of the
                                                                         triangle
        // Store positions for each vertex
        positionsArray.push(a);
        positionsArray.push(b);
        positionsArray.push(c);

        // Compute the lighting products (ambient, diffuse, specular) for this triangle
        // The lighting products are computed by combining the light's properties (ambient,
           diffuse, specular)
        // with the material properties of the object (stored in arrays for each color)
        // These products will determine how the surface reflects different types of light
        var ambientProduct = mult(lightAmbient, materialAmbientArray[colorIndex]);
        var diffuseProduct = mult(lightDiffuse, materialDiffuseArray[colorIndex]);
        var specularProduct = mult(lightSpecular, materialSpecularArray[colorIndex]);

        // Store the lighting properties for each triangle
        ambientProducts.push(ambientProduct);
        diffuseProducts.push(diffuseProduct);
        specularProducts.push(specularProduct);

        // Cycle colors
        colorIndex = (colorIndex + 1) % 4;

        index += 3;  // Increment the vertex index counter
}
```

Note that the normal vector for each vertex is computed and stored in the ‘normalsArray’. The normals represent the direction the triangle is facing and they help to determine how light interacts with the surface of it. The light products are the compute of each lighting component (ambient, diffuse, and specular) by multiplying the light’s properties vector with the material properties matrix. The ‘colorIndex’ cycles through the material colors to apply different material properties to the triangles. The code snippet below from the HTML shows the lighting uniform variable initialization and its transformations computed by the vertex shader.

Code Snippet from the ‘vertex-shader’

```
// ---- Uniforms for material lighting properties ----
uniform vec4 uAmbientProduct, uDiffuseProduct, uSpecularProduct; // store the lighting
products (ambient, diffuse, and specular) used in the Phong lighting model.
uniform vec4 uLightPosition; // Position of the light source (if w == 0, it is directional)
uniform float uShininess;    // Shininess coefficient for specular highlights
uniform mat3 uNormalMatrix;  // Transforms normals to eye space for accurate lighting
. . .
//--- Main function of the vertex shader ----
void main() {
     // Compute the position of the vertex in eye coordinates (after applying model-view
        transformation)
     vec3 pos = (uModelViewMatrix * aPosition).xyz;
```

```
    // Light direction:
    // If the light is directional (w == 0), use its direction as is;
    // Otherwise, compute the direction vector from the vertex to the light source.
    vec3 L = (uLightPosition.w == 0.0) ? normalize(uLightPosition.xyz)
                         : normalize(uLightPosition.xyz - pos);

    // Vector pointing towards the camera (eye) from the vertex position
    vec3 E = -normalize(pos);

    // Halfway vector between light and eye directions (used for specular reflection)
    vec3 H = normalize(L + E);

    // Transform the vertex normal to eye space using the normal matrix
    vec3 N = normalize(uNormalMatrix * aNormal.xyz);

    // Compute the ambient lighting component (constant color)
    vec4 ambient = uAmbientProduct;

    // Compute the diffuse lighting component (Lambertian reflection)
    float Kd = max(dot(L, N), 0.0); // Dot product gives intensity based on angle between
                                    light and normal
    vec4 diffuse = Kd * uDiffuseProduct;

    // Compute the specular lighting component (Phong reflection)
    float Ks = pow(max(dot(N, H), 0.0), uShininess); // Highlight strength depends on the
                                                     shininess factor
    vec4 specular = Ks * uSpecularProduct;

    // If the light is behind the object, eliminate specular reflection
    if (dot(L, N) < 0.0) {
        specular = vec4(0.0, 0.0, 0.0, 1.0);
    }
    // Compute the final position of the vertex in clip coordinates
    gl_Position = uProjectionMatrix * uModelViewMatrix * aPosition;

    // Combine ambient, diffuse, and specular components to determine final color
    vColor = ambient + diffuse + specular;
    vColor.a = 1.0; // Set alpha to fully opaque
}
```

Note that the ambient, diffuse, and specular components are computed in the vertex shader using the Phong lighting model. The normal vector ‘N’ is transformed to eye space using the normal matrix ‘uNormalMatrix’. The light direction is determined based on the light position ‘uLightPosition’. If it is a directional light ‘w == 0.0’, the direction is constant. Otherwise, the direction is calculated relative to the vertex. The diffuse lighting is based on the angle between the light and the surface normal, while the specular lighting depends on the halfway vector between the light and the camera. The shininess factor ‘uShininess’ controls the intensity and sharpness of the specular reflection simulation. The final color is computed by the sum of the ambient, diffuse, and specular components.

Geometry

The program draws the sphere using a recursive subdivision of tetrahedrons. This results in an approximation of a sphere shape. It starts with one tetrahedron and subdivides each facet into smaller triangles recursively (resulting in a smaller tetrahedron) creating an approximation of a sphere (Angel & Shreiner, 2020). This gives the sphere a smooth surface which helps with lighting and shading computations, particularly when dealing with reflections and vector computations because the finer the subdivision is the more accurate normal are, making the surface light reflections appear more realistic. Below is a code example in the JavaScript file that implements the recursive subdivision of a tetrahedron.

Code Snippet from the ‘tetrahedron’ and the ‘triangle’ functions

```
// Function to create a tetrahedron by subdividing its four triangular faces
function tetrahedron(a, b, c, d, n) {
    // Subdivide and draw the first triangular face
    divideTriangle(a, b, c, n);

    // Subdivide and draw the second triangular face
    divideTriangle(d, c, b, n);

    // Subdivide and draw the third triangular face
    divideTriangle(a, d, b, n);

    // Subdivide and draw the fourth triangular face
    divideTriangle(a, c, d, n);
}

// Recursive function to subdivide a triangle into smaller triangles
// 'a', 'b', 'c' are vertices of the triangle, and 'count' is the recursion depth
function divideTriangle(a, b, c, count) {
  //---- Base case: if count reaches zero, stop recursion and draw the triangle ----
  if (count > 0) {
    // Find the midpoints of each edge of the triangle
    var ab = normalize(mix(a, b, 0.5), true);  // Midpoint of edge AB
    var ac = normalize(mix(a, c, 0.5), true);  // Midpoint of edge AC
    var bc = normalize(mix(b, c, 0.5), true);  // Midpoint of edge BC

    //--- Recursive call
    // Recursively subdivide the triangle into four smaller triangles
    divideTriangle(a, ab, ac, count - 1);  // Subdivide triangle A-AB-AC
    divideTriangle(ab, b, bc, count - 1);  // Subdivide triangle AB-B-BC
    divideTriangle(bc, c, ac, count - 1);  // Subdivide triangle BC-C-AC
    divideTriangle(ab, bc, ac, count - 1);  // Subdivide triangle AB-BC-AC
  } else {
    // If count is zero, draw the triangle as-is (base case of recursion)
    triangle(a, b, c);
  }
}

// Function to add the triangle's vertices to the position array for rendering
function triangle(a, b, c) {
  // Push the vertices of the triangle into the positions array
  positions.push(a);
  positions.push(b);
  positions.push(c);
}
```

Note that the number of subdivisions is controlled by ‘count’ determining how smooth the sphere will appear. The ‘triangle’ function adds each triangle’s vertices to the positions array, which is then used to render the sphere in WebGL. The midpoints of each triangle’s edges, for example ‘ var ab = normalize(mix(a, b, 0.5), true);’, are computed and normalized. This is done to ensure that the new vertices end up on the sphere not inside of it or outside of it. The program also allows the user to control the number of subdivisions the tetrahedrons undergo using a slider interface.

Final thoughts

In this assignment, I wanted to explore applying transformations, rotation and translation, to the model view (camera view) rather than to the sphere object. This worked well at first; however, it became a bit confusing when applying the lighting components to the scene, as they needed to move in sync with the camera. I initially thought that the lighting component needed to be fixed in place, just like the sphere, so I implemented it that way. This resulted in an interesting effect, but it was somewhat disorienting since the only object in the scene was the sphere. If there were two objects combined with the lighting in the scene, I believe the effect would look more like the camera was rotating around the entire scene rather than the objects. Anyhow, this was not the intent of this project, so implemented the transformations to the lighting component giving the appearance that the sphere was rotating not the camera. This was actually much easier than the implementation with the fixed lighting. In the non-fixed lighting implementation, the lighting transformations are combined with the camera rotation and translations. In contrast, with the fixed lighting implementation, I had to manage two separate sets of positions, accounting for the fixed light's position relative to the camera's rotation and translation. In other words, in the non-fixed lighting, the positions of the lighting components do not change relative to the camera as it rotates, other hand in the fixed lighting implementation the positions of the lighting change dynamically relative to the camera due to its rotation. Additionally, I dynamically applied four different colors to the triangles to better visualize the geometry of the tetrahedron pattern. Overall, I found this assignment very challenging but also very rewarding.

References

Angel, E., & Shreiner, D. (2020). Chapter 6: Light and shading. *Interactive computer graphics. 8th edition*. Pearson Education, Inc. ISBN: 9780135258262

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