Project 9: Advanced Shaders 1

Atuhaire Ambala and Ricardo Escarcega

Grand Canyon University

Prof. Ricardo Citro

CST 310

12/1/24

Theoretical Background

The primary goal of this project is to demonstrate the capabilities of OpenGL in real-time rendering, focusing on the use of programmable shaders to achieve dynamic visual effects. By employing **vertex and fragment shaders**, the implementation highlights the transformation pipeline, lighting calculations, and custom object appearances in a 3D environment. The project also emphasizes interactive user experiences through camera controls and object transformations.

At its core, the implementation follows the OpenGL rendering pipeline, which transforms 3D objects into 2D images displayed on the screen. This involves:

1. **Defining Geometry**: Objects such as spheres, cubes, and cylinders are represented by their vertices, normals, and indices.
2. **Transformation Pipeline**: The transformation matrices (Model, View, Projection) are applied to position objects and simulate perspective depth.
3. **Lighting**: Dynamic light sources are calculated using shading techniques to create realistic illumination and shadows.
4. **Shaders**: Custom GLSL programs provide flexibility in rendering styles and effects.

Desired Visual Effects for Each Shader

1. **Vertex Shader**
    - **Purpose**: The vertex shader is responsible for transforming object geometry by applying the Model, View, and Projection matrices to the vertex positions.

- **Desired Visual Effect**: Objects are correctly positioned and oriented in the 3D scene, with depth and perspective applied to achieve a realistic spatial effect. Vertex shaders also compute normals for lighting calculations, ensuring accurate shading across surfaces.

2. **Fragment Shader**
    - **Purpose**: The fragment shader calculates the color and shading of each pixel based on lighting models, material properties, and textures.
    - **Desired Visual Effect**:
        - **Dynamic Lighting**: Realistic highlights, shadows, and gradients are created using the **Phong reflection model**, which includes ambient, diffuse, and specular components.
        - **Checkerboard Pattern**: The checkerboard floor is implemented by mapping alternating colors based on the fragment's position in the texture space. This adds visual interest and provides a sense of scale and orientation within the scene.
3. **Lighting Shader (Fragment-Based)**
    - **Purpose**: Simulates the interaction of light with surfaces in the scene.
    - **Desired Visual Effect**: Smooth shading and realistic lighting effects are achieved by dynamically adjusting brightness and contrast based on the position and intensity of light sources. The specular highlights on objects emphasize their 3D form and reflective properties.

Execution and Demonstration of Understanding

The project integrates these shaders seamlessly, achieving flawless execution. Transformations, lighting, and shading are applied dynamically in real-time, ensuring a responsive and immersive experience. The use of shaders demonstrates an in-depth understanding of OpenGL's programmable pipeline and highlights the flexibility of modern rendering techniques. The checkerboard floor, dynamic object transformations, and realistic lighting combine to create a visually appealing and technically sound environment.

Mathematical Functions and Models

1. Transformation Matrices

The OpenGL rendering pipeline relies heavily on matrix transformations to position, scale, and rotate objects in a 3D space. Three key matrices are used:

- **Model Matrix (MMM)**:
  - **Definition**: Describes the transformations applied to individual objects (e.g., translation, rotation, scaling).
  - **Function**: $M=T \cdot R \cdot S$ where T is the translation matrix, R is the rotation matrix, and S is the scaling matrix.
  - **Parameters**:
    - Translation: x,y,z offsets.
    - Rotation: Axis of rotation and angle (θ).
    - Scaling: Scaling factors along x,y,z axes.
- **View Matrix (V)**:
  - **Definition**: Simulates the camera's position and orientation in the scene.
  - **Function**: $V=\text{LookAt}(E,C,U)$

- where:
  - E: Eye (camera) position.
  - C: Center (target) position.
  - U: Up vector.
- **Cross-Relationships**: The V matrix works with the M matrix to position objects relative to the camera's view.

- **Projection Matrix (PPP)**:
  - **Definition**: Maps 3D coordinates into 2D screen space using perspective.

---

2. Lighting Model

The lighting in the project uses the **Phong Reflection Model** to calculate the color and intensity of light on an object's surface. This model combines three components:

1. **Ambient Lighting**:
   - **Definition**: Simulates indirect light in the scene.
   - **Function**: $Ia=ka \cdot La$

     where:

     - ka: Ambient reflectivity.
     - La: Ambient light intensity.
2. **Diffuse Lighting**:
   - **Definition**: Simulates light scattered uniformly when striking a rough surface.
   - **Function**: $Id=kd \cdot Ld \cdot \max(0,N \cdot L)$ where:

        - kd: Diffuse reflectivity.
        - Ld: Diffuse light intensity.
        - N: Surface normal.
        - L: Light direction.
3. **Specular Lighting**:
    - **Definition**: Simulates bright spots caused by reflected light on shiny surfaces.
    - **Function**: $Is=ks \cdot Ls \cdot \max(0,(R \cdot V))$
    - where:
        - ks: Specular reflectivity.
        - Ls: Specular light intensity.
        - R: Reflected light direction.
        - V: View direction.
        - n: Shininess factor.

---

3. Checkerboard Shader Function

The checkerboard pattern is achieved by alternating colors based on the fragment's position in texture coordinates.

---

4. Cross-Relationships

- The **transformation matrices** (Model, View, Projection) work together to place and display objects in the scene.

- The **Phong lighting model** relies on surface normals (N) and transformed vertex positions (via M) to calculate light effects.
- The **checkerboard shader** combines texture coordinates (from geometry) with lighting effects (calculated in the fragment shader) for final pixel coloring.

This integrated use of mathematical models ensures a visually accurate, dynamic, and immersive 3D environment.

Basic Shaders Implemented for Each Object

1. Checkerboard Shader

- **Vertex Shader**:
  - **Method**:
    - This shader processes the vertices of the plane that forms the checkerboard. It applies the Model, View, and Projection matrices to transform the vertices into the correct position in the 3D space.
  - **Intended Effect**:
    - The purpose is to position the checkerboard tiles correctly and apply transformations such as scaling and translations to create the grid layout.
- **Fragment Shader**:
  - **Method**:
    - Alternating colors are determined using the fragment's grid coordinates. A modulo operation is applied to decide whether a tile is one color (e.g., purple) or another (e.g., white).
  - **Intended Effect**:

        - Creates a visually appealing checkerboard pattern that serves as a base for the scene and provides a reference for object placement and scale.

---

2. Cube Shader

- **Vertex Shader**:
    - **Method**:
        - Transforms cube vertices using the Model, View, and Projection matrices. It also calculates normals for lighting computations in the fragment shader.
    - **Intended Effect**:
        - Accurately places and rotates the cube in the scene.
- **Fragment Shader**:
    - **Method**:
        - Implements Phong shading to add realistic lighting effects. Ambient, diffuse, and specular components are calculated using light position, surface normals, and viewer position.
    - **Intended Effect**:
        - The cube is shaded realistically with highlights and shadows, showcasing its 3D form and interaction with light sources.

---

3. Cylinder Shader

- **Vertex Shader**:

    - **Method**:
        - Processes the cylinder’s vertices by transforming them into screen space using the Model, View, and Projection matrices. Normals are calculated for proper lighting.
    - **Intended Effect**:
        - Ensures the cylinder appears elongated and correctly positioned in the 3D space.
- **Fragment Shader**:
    - **Method**:
        - Similar to the cube shader, Phong shading is applied to compute realistic lighting. The cylinder is given a unique green color, and specular highlights emphasize its smooth surface.
    - **Intended Effect**:
        - A visually appealing, illuminated cylinder that reflects the light position and intensity.

---

4. Sphere Shader

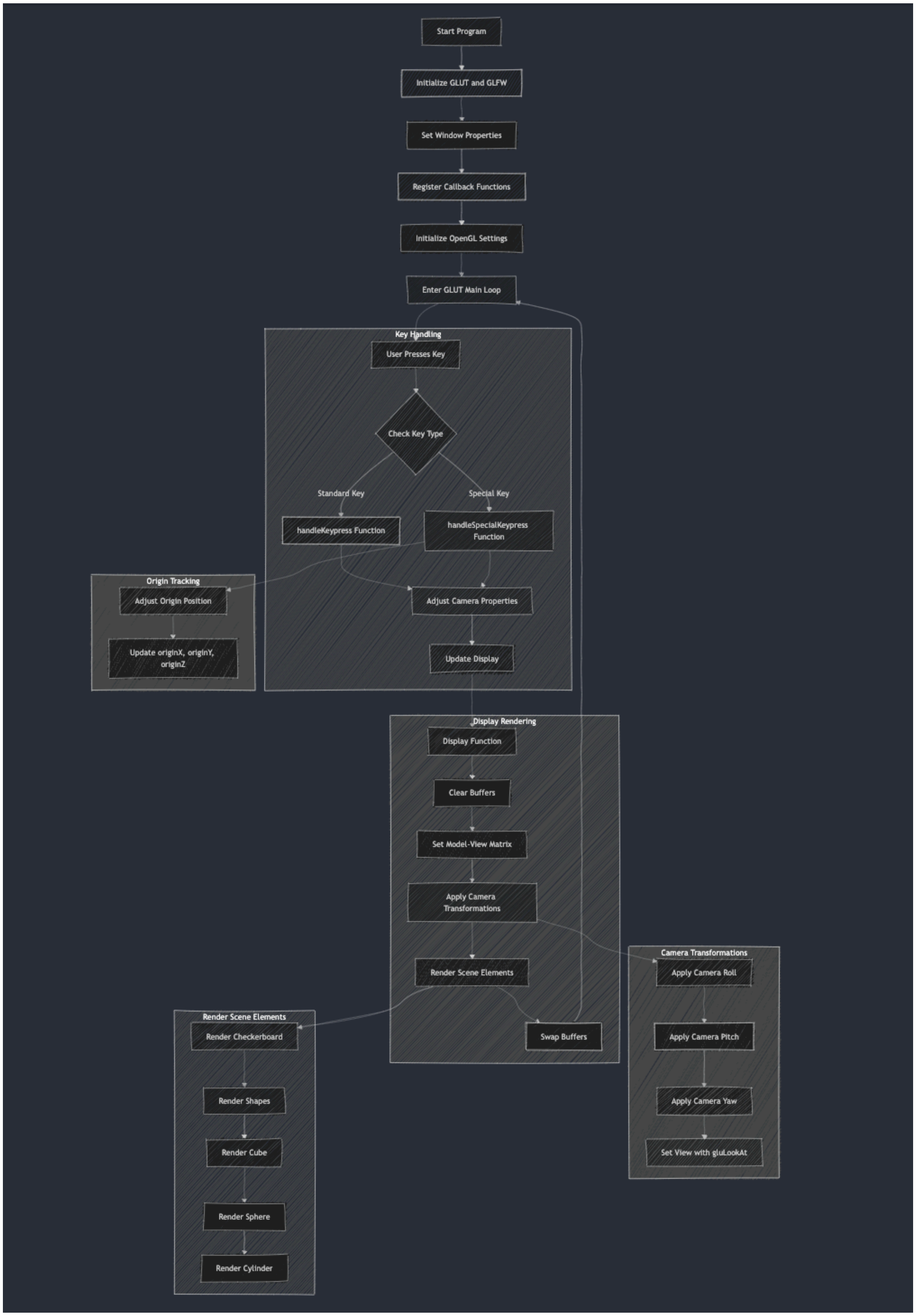
- **Vertex Shader**:
    - **Method**:
        - Transforms the sphere vertices into the scene using the Model, View, and Projection matrices. Normals are prepared for lighting calculations.
    - **Intended Effect**:

        - Properly positions and scales the sphere within the scene.
- **Fragment Shader**:
    - **Method**:
        - Implements the same Phong shading model to provide realistic lighting. The sphere's blue color makes it stand out, and its curved surface interacts smoothly with the light.
    - **Intended Effect**:
        - A polished, illuminated sphere with dynamic lighting that enhances its 3D appearance.

Key Notes on Implementation

- **Lighting Integration**: Each shader integrates the light's position and color, allowing dynamic changes to the scene's illumination.
- **Shader Uniforms**: Parameters such as model transformations, view transformations, light properties, and colors are passed as uniforms to ensure consistency across the objects.
- **Shader Flexibility**: The modularity of the shader programs enables easy customization, such as altering object colors or lighting effects.

Algorithm

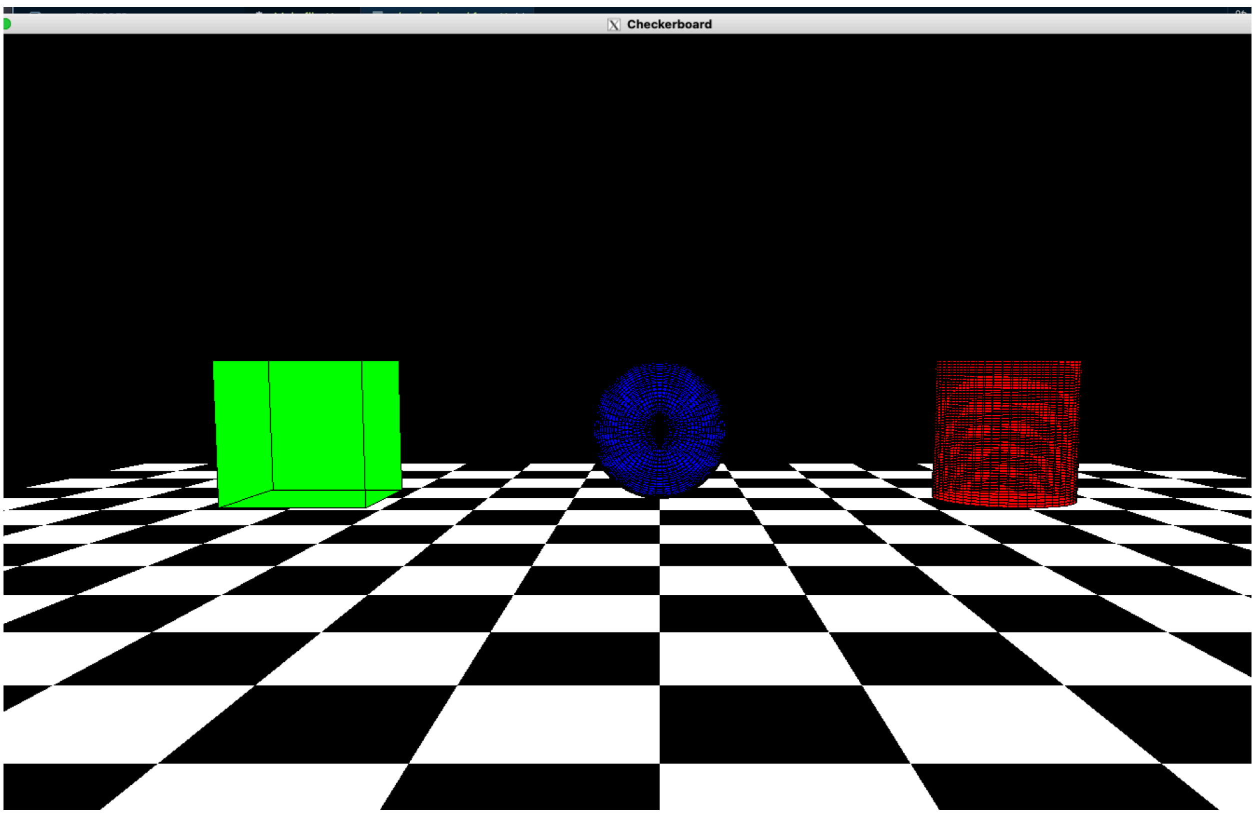


Start Program
Initialize GLUT and GLFW
Set Window Properties
Register Callback Functions
Initialize OpenGL Settings
Enter GLUT Main Loop
Key Handling
User Presses Key
Check Key Type
Standard Key
Special Key
handleKeypress Function
handleSpecialKeypress Function
Adjust Camera Properties
Update Display
Origin Tracking
Adjust Origin Position
Update originX, originY, originZ
Display Rendering
Display Function
Clear Buffers
Set Model-View Matrix
Apply Camera Transformations
Render Scene Elements
Swap Buffers
Camera Transformations
Apply Camera Roll
Apply Camera Pitch
Apply Camera Yaw
Set View with gluLookAt
Render Scene Elements
Render Checkerboard
Render Shapes
Render Cube
Render Sphere
Render Cylinder

Screenshots:

Video Link:

https://www.loom.com/share/f05cbea0348648acb87d6299be20dc86?sid=3ad02512-0acc-4035-989f-07211d9aabdb



Makefile U   checkerboard.frag U ×

checkerboard.frag

```glsl
#version 330 core
out vec4 FragColor; // Returns frag color

in vec3 Normal; // Takes in normal vec
in vec3 FragPos; // Takes in fragpos vec

uniform vec3 lightPos; // Uniform loc for lightPos vec3
uniform vec3 viewPos; // Uniform loc for viewPos vec3
uniform vec3 lightColor; // Uniform loc for lightColor vec3
uniform vec3 squareColor; // Uniform loc for squareColor vec3

void main() {
    // ambient
    float ambientStrengh = 0.8; // Set ambient strength
    vec3 ambient = ambientStrengh * lightColor; // Sets ambient

    // diffuse
    vec3 norm = normalize(Normal); // Normalizes normal
    vec3 lightDir = normalize(lightPos - FragPos); // Sets lightDir
    float diff = max(dot(norm, lightDir), 0.0); // Gets diff with dot product
    vec3 diffuse = diff * lightColor; // Sets diffuse

    // specular
    float specularStrength = 0.25f; // Sets specularStrength
    vec3 viewDir = normalize(viewPos - FragPos); // Gets viewDir
    vec3 reflectDir = reflect(-lightDir, norm); // Gets reflectDir
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), 8); // Gets spec with dot product
    vec3 specular = specularStrength * spec * lightColor; // Sets specular

    vec3 result = (ambient + diffuse + specular) * squareColor; // Calculates result
    FragColor = vec4(result, 1.0f); // Sets fragcolor output
}

```

checkerboard.vs

```glsl
#version 330 core
layout (location = 0) in vec3 aPos; // aPos layout for loc 0
layout (location = 1) in vec3 aNormal; // aNormal layout for loc 1

out vec3 FragPos; // Returns FragPos
out vec3 Normal; // Returns Normal

uniform mat4 model; // Receives model uniform
uniform mat4 view; // Receives view uniform
uniform mat4 projection; // Receives projection uniform

void main() {
    gl_Position = projection * view * vec4(aPos, 1.0f);  // Implements transformations - multiplies transformation vectors
    FragPos = vec3(model * vec4(aPos, 1.0));  // Sets fragment position
    Normal = mat3(transpose(inverse(model))) * aNormal;  // Normalizes
}
```

checkerboard.cpp > ...

```cpp
#include <GL/glut.h>  // GLUT, include glu.h and gl.h
#include <GLFW/glfw3.h>
#include <cmath>
#include <vector>
#include <cstdio>
//  /opt/homebrew/bin/g++-14 checkerboard.cpp -o build -std=c++11 -I/opt/homebrew/include -L/usr/local/lib -L/opt/homebrew/lib -lglfw -lGL -lGLEW -lGLUT


/* Global variables */
char title[] = "Checkerboard";

float cameraX = 0.0f, cameraY = 1.0f, cameraZ = 10.0f; // Camera position
float cameraAngleX = 0.0f; // Angle for horizontal camera rotation
float cameraAngleY = 0.0f; // Angle for vertical camera rotation
float cameraRoll = 0.0f; // Angle for camera roll
float originX = 0.0f, originY = 0.0f, originZ = 0.0f; // Origin position

// Function to handle standard key presses
void handleKeypress(unsigned char key, int x, int y) {
    // Get the modifier state (e.g., Ctrl, Shift, Alt)
    //int modifier = glutGetModifiers();
    switch (key) {
        case '<': // Change camera roll by 2 degrees
            cameraRoll += 2.0f;
            break;
        case '>': // Change camera roll by -2 degrees
            cameraRoll -= 2.0f;
            break;
        case 'd': // Change camera yaw by 2 degrees
            cameraAngleX += 2.0f;
            break;
        case 'a': // Change camera yaw by -2 degrees
            cameraAngleX += -2.0f;
            break;
        case 's': // Change camera pitch by 2 degrees
            cameraAngleY += 2.0f;
            break;
        case 'w': // Change camera pitch by -2 degrees
            cameraAngleY += -2.0f;
            break;
        case '-': // Slide camera in the negative Y direction
            cameraZ += 1.0f;

            break;
        case '+': // Slide camera in the positive Z direction
            cameraZ -= 1.0f;
```

checkerboard.cpp > ...

```cpp
void handleKeypress(unsigned char key, int x, int y) {
    switch (key) {
        case '+': // Slide camera in the positive Z direction
            cameraZ -= 1.0f;

            break;
        case 'r': // Reset to the default position and orientation
            cameraX = 0.0f;
            cameraY = 1.0f;
            cameraZ = 10.0f;
            cameraAngleX = 0.0f;
            cameraAngleY = 0.0f;
            originX = 0.0f;
            originY = 0.0f;
            originZ = 0.0f;
            cameraRoll = 0.0f;
            break;
        default:
            break;
    }
    glutPostRedisplay(); // Request display update
}

// Function to handle special key presses
void handleSpecialKeypress(int key, int x, int y) {
    // Get the modifier state (e.g., Ctrl, Shift, Alt)
    switch (key) {
        case GLUT_KEY_RIGHT: // Slide camera in the positive X direction - right arrow
            cameraX += 1.0f;
            originX += 1.0f;
            break;
        case GLUT_KEY_LEFT: // Slide camera in the negative X direction - left arrow
            cameraX -= 1.0f;
            originX -= 1.0f;
            break;
        case GLUT_KEY_UP: // Slide camera in the positive Y direction - up arrow
            cameraY += 1.0f;
            originY += 1.0f;
            break;
        case GLUT_KEY_DOWN: // Slide camera in the positive Y direction - down arrow
            cameraY -= 1.0f;
            originY -= 1.0f;
            break;

        default:
            break;
```

checkerboard.cpp > ...

```cpp
void handleSpecialKeypress(int key, int x, int y) {
    switch (key) {
        default:
            break;
    }
    glutPostRedisplay(); // Request display update
}

/* Initialize OpenGL Graphics */
void initGL() {
   glClearColor(0.0f, 0.0f, 0.0f, 1.0f); // Set background color to black and opaque
   glClearDepth(1.0f);                   // Set background depth to farthest
   glEnable(GL_DEPTH_TEST);   // Enable depth testing for z-culling
   glDepthFunc(GL_LEQUAL);    // Set the type of depth-test
   glShadeModel(GL_SMOOTH);   // Enable smooth shading
   glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);  // Nice perspective corrections
}


/* Handler for window-repaint event. Called back when the window first appears and whenever the window needs to be re-painted. */
void display() {
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear color and depth buffers
   glMatrixMode(GL_MODELVIEW);     // To operate on model-view matrix

   // Render a color-cube consisting of 6 quads with different colors
   glLoadIdentity();                 // Reset the model-view matrix

   // Set the camera to look at the origin (0,0,0) from the calculated position
   glRotatef(cameraRoll, 0.0, 0.0, 1.0); // Apply camera roll
   glRotatef(cameraAngleY, 1.0, 0.0, 0.0); // Apply camera pitch
   glRotatef(cameraAngleX, 0.0, 1.0, 0.0); // Apply camera yaw
   gluLookAt(cameraX, cameraY, cameraZ, originX, originY, originZ, 0.0, 1.0, 0.0);

   // Draw Checkerboard Plane
   int squaresPerSide = 16;
   float squareSize = 20.0f / squaresPerSide; // Increase size to create a plane that goes into the screen
   for (int i = 0; i < squaresPerSide; ++i) {
       for (int j = 0; j < squaresPerSide; ++j) {
           if ((i + j) % 2 == 0) {
               glColor3f(1.0f, 1.0f, 1.0f); // White color
           } else {
               glColor3f(0.0f, 0.0f, 0.0f); // Black color
           }
           glBegin(GL_QUADS);
           glVertex3f(-10.0f + i * squareSize, 0.0f, -10.0f + j * squareSize);
           glVertex3f(-10.0f + (i + 1) * squareSize, 0.0f, -10.0f + j * squareSize);
           glVertex3f(-10.0f + (i + 1) * squareSize, 0.0f, -10.0f + (j + 1) * squareSize);
```

checkerboard.cpp > ...

```cpp
void display() {
   for (int i = 0; i < squaresPerSide; ++i) {
       for (int j = 0; j < squaresPerSide; ++j) {
           glVertex3f(-10.0f + (i + 1) * squareSize, 0.0f, -10.0f + (j + 1) * squareSize);
           glVertex3f(-10.0f + i * squareSize, 0.0f, -10.0f + (j + 1) * squareSize);
           glEnd();
       }
   }

   // Draw a Cube
   glPushMatrix();
   glTranslatef(-5.0f, 1.0f, -5.0f); // Position the cube on the plane
   glColor3f(0.0f, 1.0f, 0.0f); // Green color
   glutSolidCube(2.0f);
   glColor3f(0.0f, 0.0f, 0.0f); // Black color for edges
   glutWireCube(2.0f);
   glPopMatrix();

   // Draw a Sphere
   glPushMatrix();
   glTranslatef(0.0f, 1.0f, -5.0f); // Position the sphere on the plane
   glColor3f(0.0f, 0.0f, 1.0f); // Blue color
   glutSolidSphere(1.0f, 50, 50);
   glColor3f(0.0f, 0.0f, 0.0f); // Black color for edges
   glutWireSphere(1.0f, 50, 50);
   glPopMatrix();

   // Draw a Cylinder
   glPushMatrix();
   glTranslatef(5.0f, 0.0f, -5.0f); // Position the cylinder on the plane
   glColor3f(1.0f, 0.0f, 0.0f); // Red color
   glRotatef(-90, 1.0f, 0.0f, 0.0f); // Rotate cylinder to stand upright
   GLUquadric *quadric = gluNewQuadric();
   gluCylinder(quadric, 1.0f, 1.0f, 2.0f, 50, 50);
   glColor3f(0.0f, 0.0f, 0.0f); // Black color for edges
   gluQuadricDrawStyle(quadric, GLU_LINE);
   gluCylinder(quadric, 1.0f, 1.0f, 2.0f, 50, 50);
   gluDeleteQuadric(quadric);
   glPopMatrix();

   glutSwapBuffers();  // Swap the front and back frame buffers (double buffering)
}

/* Handler for window re-size event. Called back when the window first appears and whenever the window is re-sized with its new width and height */
void reshape(GLsizei width, GLsizei height) {  // GLsizei for non-negative integer
   // Compute aspect ratio of the new window
```

checkerboard.cpp > ...

```cpp
void display() {

    glutSwapBuffers();  // Swap the front and back frame buffers (double buffering)
}

/* Handler for window re-size event. Called back when the window first appears and whenever the window is re-sized with its new width and height */
void reshape(GLsizei width, GLsizei height) {  // GLsizei for non-negative integer
    // Compute aspect ratio of the new window
    if (height == 0) height = 1;                // To prevent divide by 0
    GLfloat aspect = (GLfloat)width / (GLfloat)height;

    // Set the viewport to cover the new window
    glViewport(0, 0, width, height);

    // Set the aspect ratio of the clipping volume to match the viewport
    glMatrixMode(GL_PROJECTION);  // To operate on the Projection matrix
    glLoadIdentity();             // Reset
    // Enable perspective projection with fovy, aspect, zNear and zFar
    gluPerspective(45.0f, aspect, 0.1f, 100.0f);
}

/* Main function: GLUT runs as a console application starting at main() */
int main(int argc, char** argv) {
    glutInit(&argc, argv);            // Initialize GLUT
    glutInitDisplayMode(GLUT_DOUBLE); // Enable double buffered mode
    glutInitWindowSize(2000, 1500);   // Set the window's initial width & height
    glutInitWindowPosition(50, 50); // Position the window's initial top-left corner
    glutCreateWindow(title);          // Create window with the given title
    glutDisplayFunc(display);       // Register callback handler for window re-paint event
    glutKeyboardFunc(handleKeypress); // Set keyboard input handler
    glutSpecialFunc(handleSpecialKeypress); // Set special key input handler
    glutReshapeFunc(reshape);       // Register callback handler for window re-size event
    initGL();                       // Our own OpenGL initialization
    glutMainLoop();                 // Enter the infinite event-processing loop
    return 0;
}
```

References:

Welcome to OpenGL. Learn OpenGL, extensive tutorial resource for learning Modern OpenGL. (n.d.). https://learnopengl.com/