CS7GV5 Realtime Animation
Assignment 1
Plane Rotations

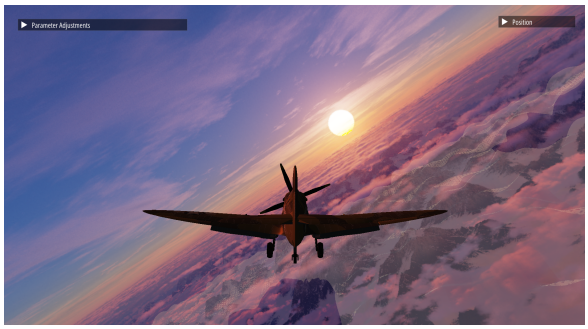
Name: Michael Millard
Student ID: 24364218

February 11, 2025

YouTube link: https://youtu.be/bGt7S5pHhe0

Introduction

In this assignment I used a 3D spitfire plane model, a mountainous sunset skybox and a 3D cloud model to implement an airplane game in which a plane is able to either freely fly around the scene or can be fixed into place for demonstration purposes (e.g. showing gimbal lock using Euler angles). Views of the game are provided in Sub-figures 1a and 1b below.



(a) Third person view of the scene

(b) First person view of the scene

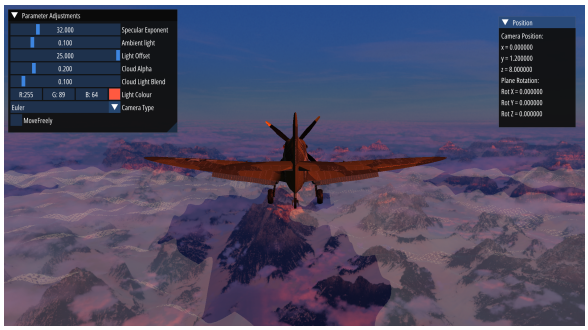
Figure 1: Two views of the game illustrating the plane, cloud and skybox

The required features (Euler angles and gimbal lock) are discussed first, followed by extra features implemented for this assignment.

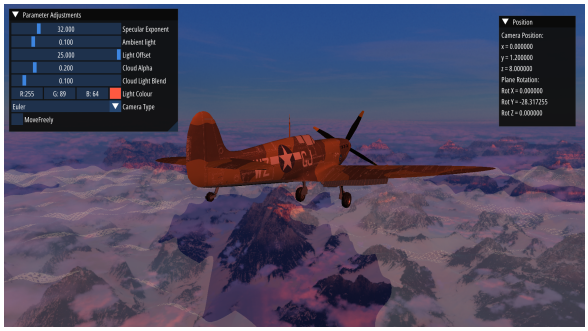
Required Features

Pitch, Roll & Yaw Using Euler Angles

Euler angles were used to initially control the rotation of the 3D plane model. To do so, I created 3 floating point values: *rotX, rotY, and rotZ*, which were used to control the pitch, yaw and roll of the plane, respectively. I used the Q and E keys to control the pitch of the plane (increase/decrease *rotX*), the A and D keys to control the yaw of the plane (increase/decrease *rotY*), and the Z and X keys to control the roll of the plane (increase/decrease *rotZ*). The Euler angle rotations are illustrated below in Sub-figures 2a, 2b, 2c, and 2d.



(a) No rotation



(b) Yaw illustration

(c) Pitch illustration

(d) Roll illustration

Figure 2: Euler angles implementation showing yaw, pitch and roll

To implement the Euler angle approach, I passed user keyboard input into a switch function which changed the value of the respective rotation angle accordingly. This function is shown below in Figure 3. Note that the variable *rotationSpeed* is hard-coded above this function and was found using trial and error.

```
switch (key)
{
// Yaw Rotation (Left/Right)
case GLFW_KEY_A:
    rotY += glm::radians(rotationSpeed);
    break;
case GLFW_KEY_D:
    rotY -= glm::radians(rotationSpeed);
    break;
// Pitch Rotation (Up/Down)
case GLFW_KEY_Q:
    rotX += glm::radians(rotationSpeed);
    break;
case GLFW_KEY_E:
    rotX -= glm::radians(rotationSpeed);
    break;
// Roll Rotation (CCW/CW)
case GLFW_KEY_Z:
    rotZ -= glm::radians(rotationSpeed);
    break;
case GLFW_KEY_X:
    rotZ += glm::radians(rotationSpeed);
    break;
default:
    break;
}
```

Figure 3: Code to adjust the Euler angles

The model matrix of the plane is then retrieved by the render loop by calling the *getPlaneModelMatrix*

defined in my camera class. The Euler angle implementation of this function is provided below in Figure 4. Note that *WORLD_FRONT, WORLD_UP and WORLD_RIGHT* are defined as constants at the top of my camera script as glm::vec3(0.0f, 0.0f, -1.0f), glm::vec3(0.0f, 1.0f, 0.0f) and glm::vec3(1.0f, 0.0f, 0.0f), respectively.

```
glm::mat4 getPlaneModelMatrix()
{
    glm::mat4 model = glm::mat4(1.0f);
    model = glm::translate(model, planePosition);

    glm::mat4 rotation = glm::mat4(1.0f);
    rotation = glm::rotate(rotation, rotZ, WORLD_FRONT);
    rotation = glm::rotate(rotation, rotY, WORLD_UP);
    rotation = glm::rotate(rotation, rotX, WORLD_RIGHT);
    model *= rotation;
    return model;
}
```

Figure 4: Code to get the plane model matrix using Euler angles

Gimbal Lock

As I show in the video demonstration, using Euler angles leads to gimbal lock when the model is rotated by 90 degrees about one of its axes of rotation. This results in the other two axes becoming aligned such that varying their corresponding rotation angle variable results in the same plane rotation. In other words, the plane model has lost a rotation degree of freedom (now only has two instead of three). In my video demonstration I adjust the yaw of the plane (*rotY*) by 90 degrees before showing that adjusting either its pitch or roll (*rotX or rotZ*) cause the plane to pitch, it can no longer roll as it is gimbal locked. The screenshots in Sub-figures 5a, 5b, 5c, and 5d below attempt to visualize the example described above.

(a) Yaw of 90°

(b) Adjusting the pitch (rotX) of the plane

(c) Adjusting the roll (rotZ) of the plane

(d) Adjusting both pitch (rotX) and roll (rotZ)

Figure 5: Euler angle gimbal lock demonstration

In Sub-figure 5a above the yaw of the plane is 90^o with the pitch and the roll both set to 0^o. I then pitch the plane upward using the Q key to demonstrate that this degree of freedom is still available; this is shown in Sub-figure 5b. I then attempt to roll the plane using the Z key but since it is gimbal locked, the plane simply continues to pitch, as shown in Sub-figure 5c. It is unable to roll. I then show one last

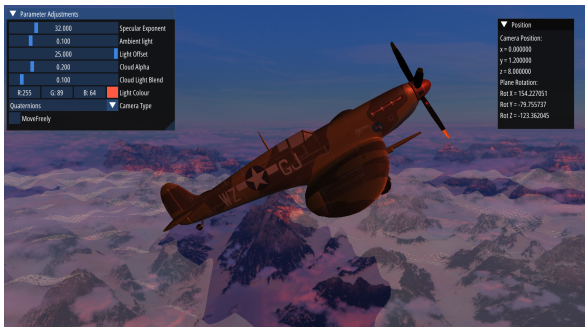
combination of both attempting to pitch and roll the plane in Sub-figure 5d, but as is clearly visible, the plane is only able to pitch as it is gimbal locked and has therefore lost one degree of freedom (rotating about the z-axis in this instance).

This issue is then resolved using quaternions, which is discussed in the next section.

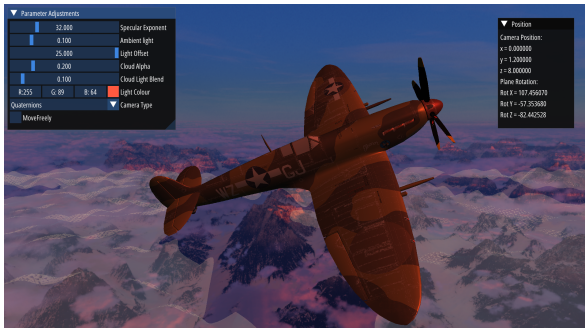
Extra Features

Quaternions

Quaternions were used to overcome gimbal lock which was encountered in the Euler angle implementation of this assignment. This entailed making changes to how I handled keyboard inputs, how I computed my camera position relative to the plane position and orientation, how I computed the plane model matrix and how I returned the view matrix of the camera. To demonstrate that quaternions were able to successfully overcome gimbal lock I have included Sub-figures 6a and 6b below.



(a) Plane pitching



(b) Plane pitching and rolling

Figure 6: Illustration of quaternions overcoming gimble lock

Sub-figure 6a shows the plane pitched at some angle in a similar manner to the Euler angle implementation where it was unable to roll due to gimble lock. Sub-figures 6b then shows that plane can roll to the side despite the yaw being 90°, indicating that gimbal lock has been overcome and the plane still has its three rotation degrees of freedom.

The code changes to my *getPlaneModelMatrix* function to use quaternions instead of Euler angles is shown below in Figure 7. It is a very simple implementation through the use of *glm::toMat4(glm::quat)* function, which I pass my *planeOrientation* quaternion to.

```
glm::mat4 getPlaneModelMatrix()
{
    glm::mat4 model = glm::mat4(1.0f);
    model = glm::translate(model, planePosition);
    glm::mat4 rotation = glm::toMat4(planeOrientation); // Convert quaternion to matrix
    model *= rotation;
    return model;
}
```

Figure 7: Code to get the plane model matrix using quaternions

The code that implements the change from Euler angles to quaternions regarding keyboard input handling is provided below in Figure 8.

```cpp
// Create rotation quaternion
glm::quat rotationQuat;

switch (key)
{
// Move Forward/Backward
case GLFW_KEY_W:
    planePosition += planeOrientation * WORLD_FRONT * velocity; // Move along forward direction
    break;
case GLFW_KEY_S:
    planePosition -= planeOrientation * WORLD_FRONT * velocity;
    break;
// Yaw Rotation (A/D) - Rotate Around Up Axis
case GLFW_KEY_A:
    angle = glm::radians(rotationSpeed);
    rotationQuat = glm::angleAxis(angle, WORLD_UP); // Rotate around Y (up)
    planeOrientation = glm::normalize(rotationQuat * planeOrientation);
    break;
case GLFW_KEY_D:
    angle = glm::radians(-rotationSpeed);
    rotationQuat = glm::angleAxis(angle, WORLD_UP); // Rotate around Y (up)
    planeOrientation = glm::normalize(rotationQuat * planeOrientation);
    break;
// Pitch Rotation (Q/E) - Rotate Around Right Axis
case GLFW_KEY_Q:
    angle = glm::radians(rotationSpeed);
    rotationQuat = glm::angleAxis(angle, planeOrientation * WORLD_RIGHT); // Rotate around Right
    planeOrientation = glm::normalize(rotationQuat * planeOrientation);
    break;
case GLFW_KEY_E:
    angle = glm::radians(-rotationSpeed);
    rotationQuat = glm::angleAxis(angle, planeOrientation * WORLD_RIGHT); // Rotate around Right
    planeOrientation = glm::normalize(rotationQuat * planeOrientation);
    break;
// Roll Rotation (Z/X) - Rotate Around Front Axis
case GLFW_KEY_Z:
    angle = glm::radians(-rotationSpeed);
    rotationQuat = glm::angleAxis(angle, planeOrientation * WORLD_FRONT); // Rotate around Front
    planeOrientation = glm::normalize(rotationQuat * planeOrientation);
    break;
case GLFW_KEY_X:
    angle = glm::radians(rotationSpeed);
    rotationQuat = glm::angleAxis(angle, planeOrientation * WORLD_FRONT); // Rotate around Front
    planeOrientation = glm::normalize(rotationQuat * planeOrientation);
    break;
default:
    break;
}
```

Figure 8: Code to move the plane using quaternions

Hierarchical Movements

The hierarchical movements I implemented in this assignment were my propeller mesh and my two wheels meshes. The 3D plane model I imported was one complete mesh. As such, I manually separated the nose and propeller into a new mesh called "propeller", and did the same for the two wheels; calling them "wheel1" and "wheel2", respectively. I did this in Blender. I then measured the offsets of each of these new meshes relative to the global origin in order to rotate them independently of the rest of the plane mesh by first translating the mesh to the global origin, rotating it as desired, and translating it back to its intended position. To implement this in my code, I checked the name of the mesh whenever I called

my draw function. If it matched any of the separated meshes mentioned above, it passed the drawing over to a new function called *drawHierarchy* which implemented the hierarchical transformations. This function is provided below in Figure 9.

```
void drawHierarchy(Shader& shader, glm::mat4& modelMat,
    float& rot, glm::vec3 meshOffset, int axis)
{
    glm::mat4 model = modelMat;
    glm::mat4 trans = glm::mat4(1);
    trans = glm::translate(trans, meshOffset);
    switch (axis)
    {
    case x_axis:
        trans = glm::rotate(trans, glm::radians(rot), glm::vec3(1.0f, 0.0f, 0.0f));
        break;
    case y_axis:
        trans = glm::rotate(trans, glm::radians(rot), glm::vec3(0.0f, 1.0f, 0.0f));
        break;
    case z_axis:
        trans = glm::rotate(trans, glm::radians(rot), glm::vec3(0.0f, 0.0f, 1.0f));
        break;
    default:
        break;
    }
    trans = glm::translate(trans, -meshOffset);
    shader.setMat4("model", model * trans);

    *... DRAW AS NORMAL AFTER THIS POINT ...*
```

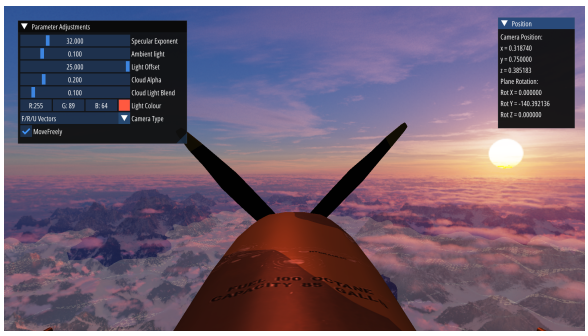
Figure 9: Code to implement hierarchical movements

As can be seen in the code above, I pass the shader, the plane model matrix, the rotation angle, the *meshOffset* and the axis of rotation to the *drawHierarchy* function. The plane model matrix is the parent model matrix which gets multiplied by the child model matrix transformation (called trans above) comprising a translation to the global origin (by the mesh offset), a rotation about the desired axis (by the given rotation amount in degrees), and a translation back to its original position after. The model matrix in my vertex shader is then set to the new model matrix before drawing as normal. A closer view of the plane is provided below in Figure 10 to better show the propeller and wheels on the plane.

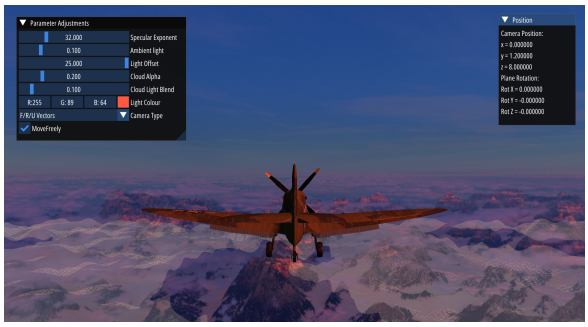
Figure 10: Close up of the plane for the wheels and propeller

1st-Person & 3rd-Person Views

My implementations of the first person and third person perspectives of the plane are illustrated below in Sub-figures 11a and 11b, respectively.



(a) First-person view



(b) Third-person view

Figure 11: Illustration of the first person and third person views

In order to implement both first person and third person views for this assignment, I opted to perform all transformations on the plane model and had a camera offset which could be toggled between two hard-coded offsets: a first person one and a third-person one. As such, I would compute the position and orientation of the plane based on user input before computing the new camera position relative to the plane. The code that implemented the toggling between first and third person views functionality and the code that placed the camera relative to the plane is given below in Figure 12.

```
void updateCameraPosition()
{
    switch (selectedCameraType)
    {
    case EulerAngles:
        if (moveFreely)
            cameraPosition = planePosition + getRotatedOffset(cameraOffset);
        else
            cameraPosition = planePosition + thirdPersonOffset;
        break;
    case Quaternions:
        if (moveFreely)
            cameraPosition = planePosition + planeOrientation * cameraOffset;
        else
            cameraPosition = planePosition + thirdPersonOffset;
        break;
    default:
        break;
    }
}

void changeCameraPerspective()
{
    // Going to 3rd person
    if (firstPerson)
    {
        cameraOffset = thirdPersonOffset;
        firstPerson = false;
    }
    // Going to 1st person
    else
    {
        cameraOffset = firstPersonOffset;
        firstPerson = true;
    }
    updateCameraPosition();
}
```

Figure 12: Code to toggle between first person and third person views

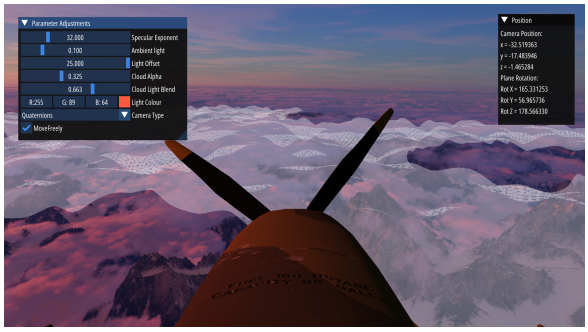
Note that *firstPersonOffset* and *thirdPersonOffset* were defined as constants at the start of the program. They were defined as glm::vec3(0.0f, 0.75f, -0.5f), right by the cockpit, and glm::vec3(0.0f, 1.2f, 8.0f), well behind and a bit above the plane, respectively. Also note that the boolean *moveFreely* was how I changed between fixing the plane in position for demonstration purposes (e.g. showing gimble lock) and allowing it to fly around the scene freely.

Good Visual Appearance

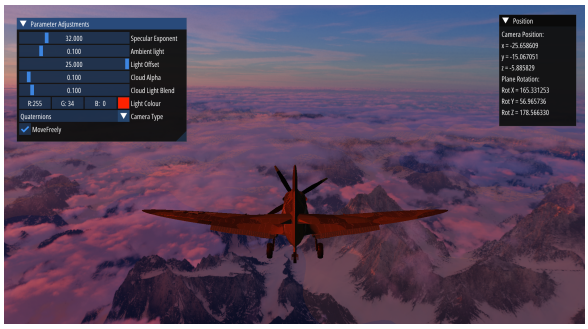
In order to have a good visual appearance for this assignment I spent a good amount of time finding a realistic plane model that wasn't too low poly and had a great texture mapping. I also spent a while finding what I believed would be an aesthetically pleasing skymap. I believe Figure 13 below demonstrates some of the aesthetic choices I made when designing this game.

Figure 13: Aesthetic 3D plane model and skymap above my 3D cloud model

Futhermore, I added an IMGUI that allowed me to tweak certain lighting parameters, cloud translucency parameters, and the colour of the sunset using a colour gamut. For instance, Sub-figures 14a and 14b below show how using my IMGUI sliders I was able to adjust the translucency and colours of my cloud model.



(a) Whiter and more translucent clouds



(b) Warmer and less translucent clouds

Figure 14: Illustration of changeable clouds for aesthetic purposes

Other tweakable parameters on my IMGUI window included the specular exponent, the ambient lighting, and the position of the light source. I tweak each of these parameters in my video demonstration to show how each of them affects the scene.

YouTube Link

The video includes a demonstration of gimbal lock using Euler angles, overcoming it using quaternions, free flying around the scene using a quaternion-based camera movement, toggling between 1st and 3rd person perspectives, and modifying lighting parameters to make the scene more appealing (using IMGUI).

YouTube link: https://youtu.be/bGt7S5pHhe0

References

I obtained the 3D spitfire airplane model from:
"Spitfire Mk IXe" (https://skfb.ly/6txDP)
by martinsifrar is licensed under
Creative Commons Attribution (http://creativecommons.org/licenses/by/4.0/)

I obtained my skymap from:
https://hdrmaps.com/above-the-clouds-2/

Declaration

I declare that all code and report writing for this submission is entirely my own work and I have not collaborated with anyone or used any code found online.

Signed: Michael Millard
Date: 11/02/2025