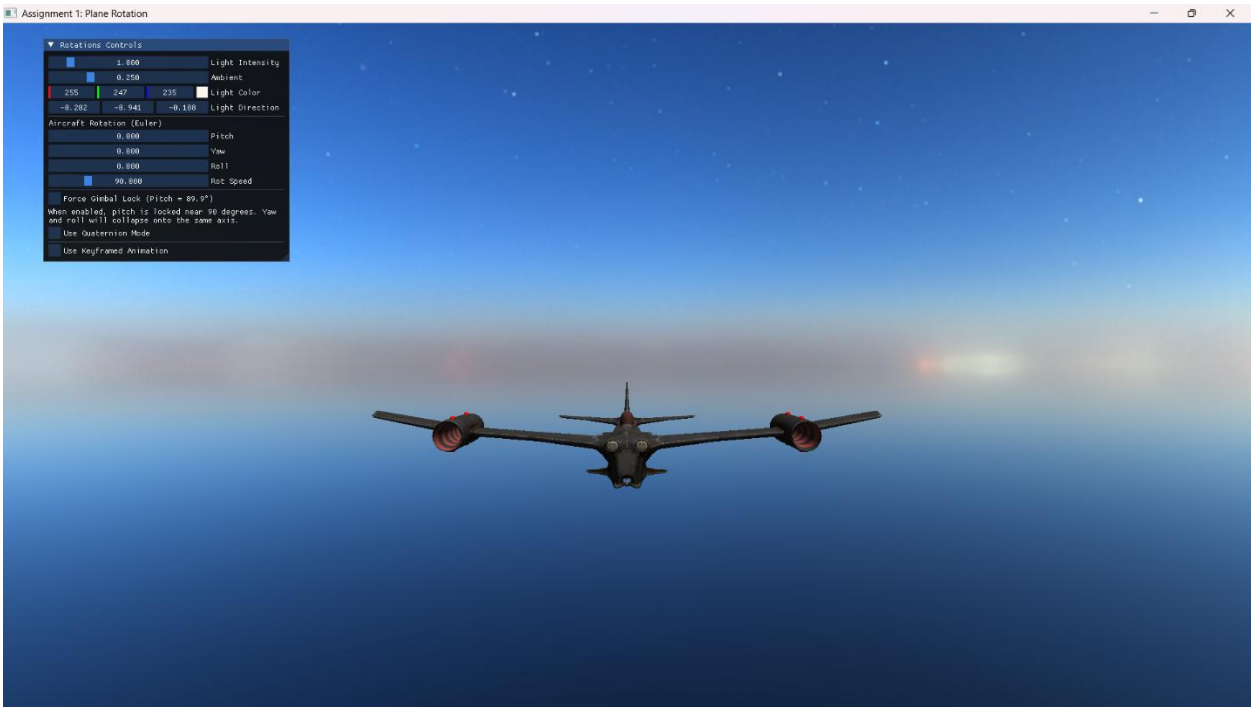
CS7GV5 Real-Time Animation

GitHub Link for full code: https://github.com/Norged-Out/CS7GV5-RTA

This project implements an interactive aircraft model animation system in OpenGL, designed to demonstrate rotation control using Euler angles (pitch/yaw/roll), the gimbal lock failure case, and a keyframed flight trajectory. The implementation is extended with quaternion-based rotations to overcome gimbal lock, smooth interpolation of rotations using SLERP, and motion smoothing techniques (damping and controlled ease-in/ease-out usage). The work is structured around my custom reusable OpenGL engine base while keeping assignment-specific logic in Main.cpp.



Core Features

1. Aircraft model with Euler-angle rotation control

The aircraft model is loaded using the Model class and rendered via a standard render loop. Euler rotation control is implemented using three explicit state variables stored in an interactive (using Dear Imgui) tweakable parameter struct, with keyboard controls adjusting these angles continuously using dt-scaled increments to ensure frame-rate independent rotation:

```
// Aircraft Euler state (degrees)
float pitchDeg = 0.0f; // X axis
float yawDeg   = 0.0f; // Y axis
float rollDeg  = 0.0f; // Z axis
```

The aircraft's orientation is applied using a Euler-to-quaternion conversion method, allowing the model's transform to remain quaternion-based while still supporting Euler-driven control. This provides clean integration and avoids building Euler matrix composition directly into rendering. Euler rotations depend on the order in which rotations are composed.

The system uses `RotationOrder::YXZ`, chosen for aircraft-style control: Yaw first (Y): sets heading direction (turn left/right), Pitch next (X): tilts nose up/down relative to heading, Roll last (Z): banks around the forward axis. This order produces intuitive behavior for aircraft motion and makes the gimbal lock singularity easy to demonstrate when pitch approaches ±90°.

```
glm::quat MathUtils::eulerToQuat(float pitchDeg, float yawDeg, float rollDeg,
                                 RotationOrder order) {
    glm::quat qX = glm::angleAxis(glm::radians(pitchDeg), glm::vec3(1, 0, 0));
    glm::quat qY = glm::angleAxis(glm::radians(yawDeg),   glm::vec3(0, 1, 0));
    glm::quat qZ = glm::angleAxis(glm::radians(rollDeg),  glm::vec3(0, 0, 1));
    switch (order) {
        case RotationOrder::YXZ: return qZ * qX * qY;
        … // other cases
    }
}
```

A "Force Gimbal Lock" mode was implemented by clamping pitch near 90° (at 89.9°). With pitch forced, yaw and roll controls visibly collapse, producing unintuitive coupled rotation. This provides a clear, visual demonstration of why Euler angles are not robust for all orientations.

This demonstration was intentionally designed to be observable:

- Pitch is constrained near the singularity,
- Yaw/Roll inputs remain active,
- The user can observe the aircraft losing independent control axes.

```
float change = params.rotSpeed * dt;
… // Input Mapping: Pitch (I/K), Yaw (J/L), Roll (U/O)
// Gimbal Lock Demo
if (params.forceGimbalLock) params.pitchDeg = 89.9f;
// Clamp pitch to avoid singularity in Euler angles
params.pitchDeg = glm::clamp(params.pitchDeg, -89.9f, 89.9f);
model.setRotationEuler(
        params.pitchDeg,
        params.yawDeg,
        params.rollDeg,
        order
);
```

2. Rotation UI feedback

Dear ImGui is used to build a debug/control window that displays and edits rotation parameters in real time. The UI includes:

- Numeric display / drag controls for `pitchDeg`, `yawDeg`, `rollDeg`
- DIsplay/control of rotation speed
- Toggle for gimbal lock forcing
- Toggle for quaternion mode
- Toggle for spine-based figure-of-eight motion keyframe animation

3. Keyframed flight path animation

A keyframe struct was defined as a discrete “state” at a specific time:

```
struct Keyframe {glm::vec3 position; glm::quat rotation; float time; // seconds};
```

The animation system maintains `animTime` that advances each frame by `dt`. The current segment is determined by finding the two keyframes that bound the current time.

```
// Advance animation time and loop back to start when we reach the end
animTime += dt;
float duration = keys.back().time;
animTime = fmod(animTime, duration);
// Find the current keyframe interval [a, b]
int i = 0;
while (i + 2 < (int)keys.size() - 1 && animTime > keys[i + 1].time) ++i;
i = glm::clamp(i, 1, (int)keys.size() - 3); // Ensure we have k0 and k3 for
Catmull-Rom
const Keyframe& k0 = keys[i - 1];
const Keyframe& k1 = keys[i];
const Keyframe& k2 = keys[i + 1];
const Keyframe& k3 = keys[i + 2];
// Normalized time between keyframes [0, 1]
float t = (animTime - k1.time) / (k2.time - k1.time);
```

This structure demonstrates that keyframes drive animation by defining discrete states, and interpolation produces the continuous motion between them. To achieve an advanced trajectory, the aircraft’s keyframe positions were arranged into a looped figure-of-eight pattern. This path is visually identifiable and includes curvature and repeated crossings, making it ideal for demonstrating the limitations of linear interpolation and the benefits of spline interpolation.

The initial baseline approach used linear interpolation between keyframe positions:

```
glm::vec3 pos = glm::mix(k1.position, k2.position, t);
```

This provides correct motion but produces sharp corners and discontinuous velocity at keyframe boundaries, especially visible in curved trajectories like a figure-of-eight.

To improve path smoothness, the approach was switched to use spline-based interpolation:

```
glm::vec3 pos = MathUtils::catmullRom(k0.position, k1.position, k2.position,
k3.position, t);
```

Where:

- `k1`-> `k2`is the current segment,
- `k0` and `k3` are neighboring control points that define tangents and curvature.

```
glm::vec3 MathUtils::catmullRom(const glm::vec3& p0, const glm::vec3& p1, const
glm::vec3& p2, const glm::vec3& p3, float t) {
    // Precompute powers of t for efficiency
    float t2 = t * t;    // t squared
    float t3 = t2 * t;   // t cubed

    return 0.5f * (
        // When t = 0, this term ensures the curve starts at p1
        (2.0f * p1)
        // Linear term: controls initial direction (tangent at p1)
        + (-p0 + p2) * t
        // Quadratic term: controls curvature
        + (2.0f * p0 - 5.0f * p1 + 4.0f * p2 - p3) * t2
        // Cubic term: ensures smooth arrival at p2
        + (-p0 + 3.0f * p1 - 3.0f * p2 + p3) * t3
    );
}
```

Catmull-Rom was chosen because:

- It is an interpolating spline (passes through keyframe points),
- Tangents are generated automatically from neighbor points (unlike pure Hermite which requires manual tangents),
- It yields smooth position and smooth direction changes suitable for animation paths,
- Guarantees C1 continuity, meaning velocity is continuous across segment boundaries.

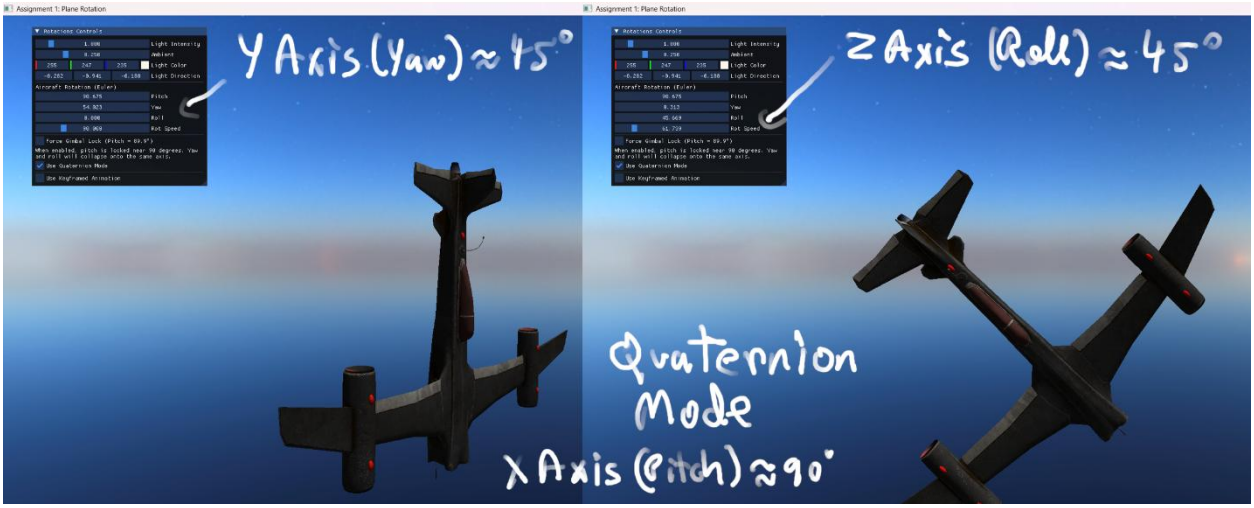
Timed keyframes require monotonic increasing timestamps. Instead of duplicating points at runtime (which breaks time ordering), additional control keyframes are authored with shifted times before and after the loop interval. This supplies valid neighbor points for Catmull-Rom at the boundaries and ensures the loop is smooth and stable.

```
std::vector<Keyframe> keyframes = {
    // Control point before start
    { glm::vec3( 0.0f,  0.0f,  0.0f), glm::quat(), -2.0f },
    // Actual animation starts here
    { glm::vec3( 0.0f,  0.0f,  0.0f), glm::quat(), 0.0f },
    // Left loop
    { glm::vec3(-10.0f,  4.0f, 5.0f), glm::quat(), 2.0f },
    { glm::vec3(-10.0f,  -4.0f, -5.0f), glm::quat(), 4.0f },
    // Back through center
    { glm::vec3( 0.0f,  0.0f, 0.0f), glm::quat(), 6.0f },
    // Right loop
    { glm::vec3( 10.0f,  4.0f, 5.0f), glm::quat(), 8.0f },
    { glm::vec3( 10.0f,  -4.0f, -5.0f), glm::quat(), 10.0f },
    // End at center
    { glm::vec3( 0.0f,  0.0f,  0.0f), glm::quat(), 12.0f },
    // Control point after end
    { glm::vec3( 0.0f,  0.0f,  0.0f), glm::quat(), 13.0f }
}
```

Extra Features

(a) Quaternion-based rotations to overcome gimbal lock

Quaternions are used as the underlying rotation representation in the model transform. Euler input is converted into quaternions when in Euler mode; quaternion mode is also supported for direct quaternion composition and for smooth orientation control during animation.




```
float angle = glm::radians(params.rotSpeed * dt);
… // Input Mapping: Pitch (I/K), Yaw (J/L), Roll (U/O)
aircraftQuat = glm::normalize(aircraftQuat);
model.setRotationQuat(aircraftQuat);
```

Quaternions solve gimbal lock because they do not represent orientation as three sequential axis rotations. They represent a single rotation in 4D space that remains valid for all orientations.

This provides a clear contrast:

- Euler mode: gimbal lock demonstrable at pitch ~90°
- Quaternion mode: stable for all orientations

(b) Smooth interpolation of rotations

Although the keyframe struct contains rotation values, final orientation during spline animation is computed dynamically from velocity to ensure the aircraft always aligns with the path direction. However, this derived orientation can still change sharply, especially in curved motion. Therefore, SLERP is used to smooth orientation changes over time:

- compute a target orientation targetRot from direction of motion,
- blend from prevRot toward targetRot using SLERP.

This produces rotational damping: smooth turns towards desired direction rather than snapping.

```
// Store previous state for velocity calculation
static glm::vec3 prevPos = keys[0].position;
static glm::quat prevRot = keys[0].rotation;
// velocity for look rotation
glm::vec3 velocity = pos - prevPos;
glm::quat targetRot = prevRot;
if (glm::length(velocity) > 0.001f) {
    glm::vec3 forward = glm::normalize(velocity);
    glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f);
    // Stable look rotation
    glm::mat4 look = glm::lookAt(glm::vec3(0.0f), -forward, up);
    targetRot = glm::quat_cast(glm::inverse(look));
}
// Interpolate rotation with SLERP
glm::quat rot = MathUtils::slerp(prevRot, targetRot, 0.15f);
```

(c) Motion smoothing techniques

The problem with per-segment easing: applying easeInOut(t) directly to every keyframe interval causes the aircraft to decelerate near every keyframe. This results in stop-and-go motion, which looks incorrect for an aircraft following a continuous flight path.

```
float MathUtils::easeInOut(float t) {
    t = clamp01(t);
    return t * t * (3.0f - 2.0f * t);
}
```

To address this, instead of applying global easing to every keyframe segment, smoothing was introduced in a controlled manner, targeting orientation changes and loop boundaries separately:

1. **Damping via SLERP (primary smoothing):** The aircraft's orientation is smoothed using SLERP in each frame. This avoids abrupt motion changes in rotation (turning).
2. **Controlled ease-in/out near the loop seam (secondary smoothing):** Ease-in/ease-out is applied only near the animation loop boundary (start/end of the cycle), preventing abrupt wraparound behavior without introducing slowdowns at every keyframe.

```
//Seam-based easing (only near loop start/end)
float seamTime = 0.5f; // seconds of smoothing near start/end
bool nearStart = animTime < seamTime;
bool nearEnd   = animTime > duration - seamTime;
// Apply easing only withing start/end seam regions
if (nearStart || nearEnd) t = MathUtils::easeInOut(t);
```

This produces:

- continuous flight speed throughout the figure-of-eight,
- smooth entry/exit behavior at the loop seam,
- stable turning due to rotational damping.