



Exam Project, TDD & Docs

09.01.2026

Sixten Björling

Forsbergs School, Kingdom of Sweden

Stockholm, Sweden

contact@sibjor.com

Overview

This is the documentation of my exam project at Forsbergs School studying Game Programming. Here we will explain how I've implemented my final .apk Android executable Native Application with technologies such as Kotlin, OpenGL ES 3.0, C and C++.

Goals

The goals for this project were not really delivered as planned as I learned that it would require far more time to implement all my dream goals for the "game" application. But aside from that I got to a point in which I could demonstrate a lot of learnt materials, like the all self made: .obj parser/loader, basic shaders, window context, renderer and final native library to read from Kotlin. I learnt a lot and I think the end result got very pretty, showcasing an un-textured airplane model correctly with several effects and shaders along with a pink sun.

Specifications

Lets go through the code active in the project, this will act as an index:

1. Kotlin & Java Native Interface
2. Renderer
3. Window
4. Math
5. .obj
6. Native Library
7. Shaders
8. Game Controller

1. Kotlin & Java Native Interface

JNI (Java Native Interface) acts as the bridge connecting your Android application's Java/Kotlin layer with C++ game engine. In this project, the implementation is centered around native-lib.cpp, which exposes specific functions that the Android activity calls in response to lifecycle events and user input.

When the application surface is created (onSurfaceCreated), a dedicated rendering thread is spawned. This thread initializes the Game logic and the Renderer (passing the native window and asset manager) and then enters a continuous loop where it updates the game state and draws frames as long as the surface exists.

The bridge also handles input in real-time. Functions like onJoystickMoved and onThrottleChanged are called directly from Kotlin when UI controls are manipulated. These calls immediately update the atomic state variables within the C++ Game instance, ensuring

that your flight controls feel responsive without blocking the main UI thread. Essentially, native-lib.cpp is the traffic controller that ensures your Java UI commands are executed by the C++ engine and that the C++ engine has a window to draw on.

```
import android.annotation.SuppressLint
import android.content.res.AssetManager
import android.os.Bundle
import android.view.InputDevice
import android.view.KeyEvent
import android.view.MotionEvent
import android.view.Surface
import android.view.SurfaceHolder
import android.view.View
import androidx.appcompat.app.AppCompatActivity
import com.example.myflight.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity(), SurfaceHolder.Callback {

    private lateinit var binding: ActivityMainBinding

    @SuppressLint("ClickableViewAccessibility")
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        // Gör ytan fokuserbar så att den kan ta emot knapptryckningar
        binding.surfaceView.isFocusableInTouchMode = true
        binding.surfaceView.requestFocus()
    }
}
```

```
// Lyssna på när ytan skapas/förstörs
binding.surfaceView.holder.addCallback(this)
}

// Hanterar joystick-rörelser
override fun onGenericMotionEvent(event: MotionEvent?): Boolean {
    if (event?.source?.and(InputDevice.SOURCE_JOYSTICK) ==
InputDevice.SOURCE_JOYSTICK &&
        event.action == MotionEvent.ACTION_MOVE) {

        val xAxis = event.GetAxisValue(MotionEvent.AXIS_X)
        val yAxis = event.GetAxisValue(MotionEvent.AXIS_Y)

        onJoystickMoved(xAxis, yAxis)

        return true
    }

    return super.onGenericMotionEvent(event)
}

// Hanterar knapptryckningar (för gasen)
override fun onKeyDown(keyCode: Int, event: KeyEvent?): Boolean {
    if (event?.source?.and(InputDevice.SOURCE_GAMEPAD) ==
InputDevice.SOURCE_GAMEPAD) {
        if (keyCode == KeyEvent.KEYCODE_BUTTON_A) {
            onThrottleChanged(1.0f) // Öka gasen
            return true
        }

        if (keyCode == KeyEvent.KEYCODE_BUTTON_B) {
            onThrottleChanged(-1.0f) // Minska gasen
            return true
        }
    }
}
```

```
    }

    }

    return super.onKeyDown(keyCode, event)
}

// -- Metoder för SurfaceHolder.Callback --

override fun surfaceCreated(holder: SurfaceHolder) {
    onSurfaceCreated(holder.surface, assets)
}

override fun surfaceChanged(holder: SurfaceHolder, format: Int,
width: Int, height: Int) {
    onSurfaceChanged(width, height)
}

override fun surfaceDestroyed(holder: SurfaceHolder) {
    onSurfaceDestroyed()
}

// -- Externa JNI-funktioner som implementeras i C++ --

private external fun onSurfaceCreated(surface: Surface,
assetManager: AssetManager)

private external fun onSurfaceChanged(width: Int, height: Int)
private external fun onSurfaceDestroyed()
private external fun onJoystickMoved(x: Float, y: Float)
private external fun onThrottleChanged(change: Float)
```

```
companion object {  
    // Laddar vårt 'myflight'-bibliotek när appen startar  
    init {  
        System.loadLibrary("myflight")  
    }  
}  
  
// --- JNI-brygga ---  
extern "C" {  
    JNIEXPORT void JNICALL  
Java_com_example_myflight_MainActivity_onSurfaceCreated(JNIEnv* env,  
jobject, jobject surface, jobject jAssetManager) {  
        ANativeWindow* window = ANativeWindow_fromSurface(env,  
surface);  
  
        AAssetManager* assets = AAssetManager_fromJava(env,  
jAssetManager);  
  
        if (window != nullptr) {  
            game_instance = std::make_unique<Game>();  
            rendering = true;  
            render_thread = std::thread(renderLoop, window, assets);  
        }  
    }  
  
    JNIEXPORT void JNICALL  
Java_com_example_myflight_MainActivity_onSurfaceDestroyed(JNIEnv *,  
jobject) {  
        rendering = false;  
        if (render_thread.joinable()) {  
            render_thread.join();  
        }  
    }  
}
```

```
    }

    game_instance.reset(nullptr);
}

JNIEXPORT void JNICALL
Java_com_example_myflight_MainActivity_onSurfaceChanged(JNIEnv*,
jobject, jint width, jint height) {
    if(renderer_instance) {
        renderer_instance->onSurfaceResized(width, height);
    }
}

JNIEXPORT void JNICALL
Java_com_example_myflight_MainActivity_onJoystickMoved(JNIEnv *,
jobject, jfloat x, jfloat y) {
    if(game_instance) {
        game_instance->onJoystickMoved(x,y);
    }
}

JNIEXPORT void JNICALL
Java_com_example_myflight_MainActivity_onThrottleChanged(JNIEnv*,
jobject, jfloat change) {
    if(game_instance) {
        game_instance->onThrottleChanged(change);
    }
}
}
```

2. Renderer

The renderer is the core component bridging your C++ logic with the device's graphics hardware through OpenGL ES 3. It handles the initialization of the graphics context via EGL, creating a drawing surface directly on the native window provided by Android. The rendering loop is straightforward: for each frame, it calculates the necessary matrices (view and projection) to simulate a camera perspective and issues draw calls for the active scene objects, such as the airplane and the sun. It utilizes shaders—small programs running on the GPU—to determine how vertices are positioned (.vert) and how pixels are colored (.frag). The renderer also manages asset loading, reading the raw .obj model data and compiling the shader source code from the assets folder into executable programs for the graphics card. Essentially, it translates your game state (positions, rotations) into the visual image displayed on the screen.

Here parts of "renderer.cpp":

```
const Vec3 OBJECT_COLOR = {0.6f, 0.65f, 0.7f};

const Vec3 LIGHT_COLOR = {1.0f, 0.9f, 0.7f};

const Vec3 SUN_POSITION_WORLD = {-150.0f, 40.0f, -250.0f};

// Solen är nu dubbelt så stor (var 9.0f)

const float SUN_SIZE_WORLD = 18.0f;

void Renderer::drawFrame(const Vec3& plane_pos, const
Quaternion& plane_rot, const Vec3& camera_pos, const Vec3&
camera_up) {

    float view[16], proj[16];

    matrix_look_at(view, camera_pos, plane_pos, camera_up);

    matrix_perspective(proj, 45.0f,
(float)surfaceWidth_/(float)surfaceHeight_, 0.1f, 1000.0f);

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Korrekt ritordning för 3D

    drawSun(view, proj);

    drawPlane(view, proj, camera_pos, plane_rot, plane_pos);

    eglSwapBuffers(eglDisplay_, eglSurface_);
```



```

}

bool Renderer::initOpenGL(AAssetManager* assetManager) {
    std::string planeVS_src = readAssetFile(assetManager,
"plane.vert"); std::string planeFS_src =
readAssetFile(assetManager, "plane.frag");

    std::string sunVS_src = readAssetFile(assetManager,
"sun.vert"); std::string sunFS_src =
readAssetFile(assetManager, "sun.frag");

    if (planeVS_src.empty() || planeFS_src.empty() ||
sunVS_src.empty() || sunFS_src.empty()) return false;

    planeProgram_ = glCreateProgram();

    GLuint vs = loadShader(GL_VERTEX_SHADER,
planeVS_src.c_str()); GLuint fs =
loadShader(GL_FRAGMENT_SHADER, planeFS_src.c_str());

    glAttachShader(planeProgram_, vs);
glAttachShader(planeProgram_, fs);
glLinkProgram(planeProgram_);

    glDeleteShader(vs); glDeleteShader(fs);

    planeMvpLoc_ = glGetUniformLocation(planeProgram_, "u_mvp");
planeModelLoc_ = glGetUniformLocation(planeProgram_,
"u_model");

    planeLightPosLoc_ = glGetUniformLocation(planeProgram_,
"u_lightPos_world"); planeViewPosLoc_ =
glGetUniformLocation(planeProgram_, "u_viewPos_world");

    planeColorLoc_ = glGetUniformLocation(planeProgram_,
"u_objectColor"); planeLightColorLoc_ =
glGetUniformLocation(planeProgram_, "u_lightColor");

    sunProgram_ = glCreateProgram();

    vs = loadShader(GL_VERTEX_SHADER, sunVS_src.c_str()); fs =
loadShader(GL_FRAGMENT_SHADER, sunFS_src.c_str());

```

```

    glAttachShader(sunProgram_, vs); glAttachShader(sunProgram_,
fs); glLinkProgram(sunProgram_);

    glDeleteShader(vs); glDeleteShader(fs);

    sunViewMatLoc_ = glGetUniformLocation(sunProgram_,
"u_view"); sunProjMatLoc_ = glGetUniformLocation(sunProgram_,
"u_proj");

    sunPosLoc_ = glGetUniformLocation(sunProgram_,
"u_sun_pos_world"); sunSizeLoc_ =
glGetUniformLocation(sunProgram_, "u_sun_size");

    glClearColor(0.1f, 0.2f, 0.3f, 1.0f);

    glEnable(GL_DEPTH_TEST);

    glEnable(GL_CULL_FACE);

    glCullFace(GL_BACK);

    if (!loadObjModel(assetManager)) { LOGE("Kunde inte ladda
modellen."); return false; }

    return true;
}

```

3. Window

A window context on Android essentially acts as the bridge between your native rendering code and the actual surface the user sees on the device. Unlike desktop platforms where you might rely on GLFW, SDL, or platform-specific windowing APIs, Android exposes a very minimal interface through **ANativeWindow**, which is basically a handle to a drawable surface provided by the system. Because of this, the implementation becomes fairly direct: you receive the window pointer from the Java side, and from that point on you are responsible for creating an EGL context, binding it to the window, and preparing OpenGL so it can draw into that surface.

The structure here follows that idea. A few constants define colors and world-space positions used by the renderer, and the **Renderer** class itself simply manages the lifecycle of the EGL context. When the renderer is created, nothing heavy happens; the actual initialization is deferred to **init()**, which takes both the asset manager (for loading shaders or models) and the native window. The initialization is

straightforward: first set up EGL using the window, then initialize OpenGL resources. If either step fails, the renderer reports failure and avoids entering an invalid rendering state. When the renderer is destroyed, the EGL context is cleaned up to ensure the system doesn't leak GPU resources.

This keeps the rendering pipeline predictable: Android gives you a window, you attach EGL to it, and once the context is active, all drawing commands go directly to the screen through OpenGL ES. Below is the implementation that handles this setup.

```
#include <android/native_window.h>

#define LOG_TAG "Renderer"

Renderer::Renderer() = default;
Renderer::~Renderer() { destroyEGL(); }

bool Renderer::init(AAssetManager* assetManager, ANativeWindow*
window) {
    if (!initEGL(window)) return false;
    if (!initOpenGL(assetManager)) return false;
    return true;
}

bool Renderer::initEGL(ANativeWindow* window) {
    eglDisplay_ = eglGetDisplay(EGL_DEFAULT_DISPLAY);
    eglInitialize(eglDisplay_, nullptr, nullptr);

    const EGLint attribs[] = {EGL_RENDERABLE_TYPE,
EGL_OPENGL_ES3_BIT, EGL_SURFACE_TYPE, EGL_WINDOW_BIT,
EGL_BLUE_SIZE, 8, EGL_GREEN_SIZE, 8, EGL_RED_SIZE, 8,
EGL_DEPTH_SIZE, 24, EGL_NONE};

    EGLConfig config; EGLint numConfigs;
    eglChooseConfig(eglDisplay_, attribs, &config, 1,
&numConfigs);
```

```

    eglSurface_ = eglCreateWindowSurface(eglDisplay_, config,
window, nullptr);

    const EGLint context_attribs[] =
{EGL_CONTEXT_CLIENT_VERSION, 3, EGL_NONE};

    eglContext_ = eglCreateContext(eglDisplay_, config, nullptr,
context_attribs);

    if (eglMakeCurrent(eglDisplay_, eglSurface_, eglSurface_,
eglContext_) == EGL_FALSE) { LOGE("eglMakeCurrent failed");
return false; }

    onSurfaceResized(ANativeWindow_getWidth(window),
ANativeWindow_getHeight(window));

    return true;
}

void Renderer::destroyEGL() {

    glDeleteBuffers(1, &plane_vbo_pos_); glDeleteBuffers(1,
&plane_vbo_norm_);

    glDeleteProgram(planeProgram_);
glDeleteProgram(sunProgram_);

    if (eglDisplay_ != EGL_NO_DISPLAY) {

        eglMakeCurrent(eglDisplay_, EGL_NO_SURFACE,
EGL_NO_SURFACE, EGL_NO_CONTEXT);

        if (eglContext_ != EGL_NO_CONTEXT)
eglDestroyContext(eglDisplay_, eglContext_);

        if (eglSurface_ != EGL_NO_SURFACE)
eglDestroySurface(eglDisplay_, eglSurface_);

        eglTerminate(eglDisplay_);

    }

    eglDisplay_ = EGL_NO_DISPLAY; eglContext_ = EGL_NO_CONTEXT;
eglSurface_ = EGL_NO_SURFACE;
}

```

4. Math

A small header like this provides the essential building blocks needed for handling rotations, directions, and camera transforms in a 3D application. While there are larger libraries available that cover every possible case, I wanted something minimal and transparent, where each operation can be understood directly from the code. Everything here is written in plain C/C++ without external dependencies, which makes it easy to embed into small engines or tools without worrying about overhead or initialization steps.

The implementation is straightforward: quaternions are represented as four floats, vectors as three floats, and matrices as simple 4×4 float arrays. Rotations are created either from axis-angle pairs or by multiplying quaternions together, and vectors can be rotated by treating the quaternion as a compact rotation operator. The matrix functions follow the same philosophy — a perspective matrix is built from the field of view and aspect ratio, a look-at matrix is constructed from eye and target positions, and quaternion-to-matrix conversion simply expands the quaternion into its corresponding rotation matrix.

Since everything is stored in raw float buffers, the data can be passed directly to OpenGL, Vulkan, or any other graphics API without additional conversion. This keeps the pipeline clean: compute the values in C/C++, push them into a buffer, and let the GPU consume them as-is. Below is the header containing these utilities:

```
#pragma once

#include <cmath>
#include <cstring> // For memcpy

#ifndef M_PIf
#define M_PIf 3.14159265358979323846f
#endif

struct Vec3 { float x, y, z; };
struct Quaternion { float x, y, z, w; };

inline Quaternion normalize(const Quaternion& q) {
```

```

    float mag_inv = 1.0f / sqrtf(q.x*q.x + q.y*q.y + q.z*q.z +
q.w*q.w);

    return {q.x*mag_inv, q.y*mag_inv, q.z*mag_inv, q.w*mag_inv};
}

inline Quaternion operator*(const Quaternion& a, const
Quaternion& b) {

    return {

        a.w*b.x + a.x*b.w + a.y*b.z - a.z*b.y,
        a.w*b.y - a.x*b.z + a.y*b.w + a.z*b.x,
        a.w*b.z + a.x*b.y - a.y*b.x + a.z*b.w,
        a.w*b.w - a.x*b.x - a.y*b.y - a.z*b.z

    };

}

inline Quaternion create_from_axis_angle(const Vec3& axis,
float angle_deg) {

    float half_angle_rad = angle_deg * (M_PIf / 180.0f) * 0.5f;
    float s = sinf(half_angle_rad);

    return {axis.x * s, axis.y * s, axis.z * s,
cosf(half_angle_rad)};

}

inline Vec3 cross(const Vec3& a, const Vec3& b) {

    return { a.y * b.z - a.z * b.y, a.z * b.x - a.x * b.z, a.x *
b.y - a.y * b.x };

}

inline Vec3 rotate_vector_by_quaternion(const Vec3& v, const
Quaternion& q) {

    Vec3 u = {q.x, q.y, q.z};

```

```

    Vec3 t = cross(u, v);

    t.x *= 2.0f; t.y *= 2.0f; t.z *= 2.0f;

    Vec3 t2 = cross(u, t);

    return { v.x + q.w * t.x + t2.x, v.y + q.w * t.y + t2.y, v.z
+ q.w * t.z + t2.z };
}

inline void matrix_multiply(float* dest, const float* a, const
float* b) {
    float res[16];

    for (int i = 0; i < 4; i++) for (int j = 0; j < 4; j++) {
res[j*4 + i] = 0.0f; for (int k = 0; k < 4; k++) res[j*4 + i]
+= a[k*4 + i] * b[j*4 + k]; }

    memcpy(dest, res, 16 * sizeof(float));
}

inline void matrix_look_at(float* mat, const Vec3& eye, const
Vec3& center, const Vec3& up) {
    Vec3 f = {center.x - eye.x, center.y - eye.y, center.z -
eye.z};

    float inv_len = 1.0f / sqrtf(f.x*f.x + f.y*f.y + f.z*f.z);
    f.x *= inv_len; f.y *= inv_len; f.z *= inv_len;

    Vec3 s = cross(f, up);

    inv_len = 1.0f / sqrtf(s.x*s.x + s.y*s.y + s.z*s.z);
    s.x *= inv_len; s.y *= inv_len; s.z *= inv_len;

    Vec3 u = cross(s, f);

    mat[0] = s.x; mat[1] = u.x; mat[2] = -f.x; mat[3] =
0.0f;

```

```

    mat[4] = s.y; mat[5] = u.y; mat[6] = -f.y; mat[7] =
0.0f;

    mat[8] = s.z; mat[9] = u.z; mat[10] = -f.z; mat[11] =
0.0f;

    mat[12] = -(s.x*eye.x + s.y*eye.y + s.z*eye.z);
    mat[13] = -(u.x*eye.x + u.y*eye.y + u.z*eye.z);
    mat[14] = (f.x*eye.x + f.y*eye.y + f.z*eye.z);
    mat[15] = 1.0f;
}

inline void matrix_perspective(float* mat, float fov, float
aspect, float near_p, float far_p) {
    float f = 1.0f / tanf(fov * (M_PI / 360.0f));
    memset(mat, 0, 16 * sizeof(float));

    mat[0] = f / aspect; mat[5] = f; mat[10] = (far_p + near_p)
/ (near_p - far_p); mat[11] = -1.0f;

    mat[14] = (2.0f * far_p * near_p) / (near_p - far_p);
}

inline void matrix_from_quaternion(float* m, const Quaternion&
q) {
    float xx = q.x*q.x, yy = q.y*q.y, zz = q.z*q.z, xy =
q.x*q.y, xz = q.x*q.z, yz = q.y*q.z, wx = q.w*q.x, wy =
q.w*q.y, wz = q.w*q.z;

    m[0]=1-2*(yy+zz); m[4]=2*(xy-wz); m[8]=2*(xz+wy); m[12]=0;
    m[1]=2*(xy+wz); m[5]=1-2*(xx+zz); m[9]=2*(yz-wx); m[13]=0;
    m[2]=2*(xz-wy); m[6]=2*(yz+wx); m[10]=1-2*(xx+yy); m[14]=0;
    m[3]=0; m[7]=0; m[11]=0; m[15]=1;
}

```

5. .obj Parser and Loader

An .obj file is a 3D model consisting of coordinates of different variants written in plain text. If you would use a binary format, launching and playing an application with such models would possibly result in better framerates, but since to my knowledge so far don't know how binary formats are read with computers, I used the .obj format. It's a pretty straight forward implementation, simply you read strings or characters from the .obj file and push it to a buffer where it's later delivered and read by the hardware through its OpenGL support or similar. Each "variant" of a coordinate would consist of values describing either a triangle or "quad" that said - three of four values, separated by comma. Thus you read each line ending starting with the name of such "coordinate naming variant" and separate its values by commas, sending it further in C/C++ space. Here's a part of an .obj file:

```
# Exported from Wings 3D 1.5.1
#778 vertices, 776 faces
v 0.20458359 0.19793898 -0.51668685
v 1.5410231e-2 0.13989404 0.88424362
v 4.1158740e-2 0.14159951 0.85076779
v -9.1397396e-5 0.13913730 0.89149761
v 0.21377173 0.18073259 -4.8865800e-2
v 0.21332466 0.18156824 -6.8344798e-2
v 0.20188896 0.19057371 -0.77500838
v -9.1397396e-5 0.39246479 -0.10113870
v -9.1397396e-5 0.37022855 0.12884735
v -9.1397396e-5 0.14603175 0.88020278
v 1.9285702e-2 0.14732825 0.86990342
v 0.13165289 0.34423930 0.10423873
```

The "vertices" mentioned above at the top would act as 3D world destination points, and the "faces" would be values combining those vertices into either triangles or quads as mentioned before. When quads are used, implementation is different in a pretty different way to a non mathematical genius such as myself due to being read twice instead of only once, as the triangles. Check this example:

```
else if (lineType == "f") {
    std::string v1_str, v2_str, v3_str, v4_str;
    lineStream >> v1_str >> v2_str >> v3_str;
```

```

    auto parseFaceIndex = [&](const std::string& s, int& p_idx,
int& n_idx) { size_t slash = s.find("//"); p_idx =
std::stoi(s.substr(0, slash)) - 1; n_idx =
std::stoi(s.substr(slash + 2)) - 1; };

    int p[4], n[4];

    parseFaceIndex(v1_str, p[0], n[0]); parseFaceIndex(v2_str,
p[1], n[1]); parseFaceIndex(v3_str, p[2], n[2]);

    int indices[] = {0, 1, 2};

    for (int i : indices) {

        final_positions.insert(final_positions.end(),
{temp_positions[p[i]*3], temp_positions[p[i]*3+1],
temp_positions[p[i]*3+2]});

        final_normals.insert(final_normals.end(),
{temp_normals[n[i]*3], temp_normals[n[i]*3+1],
temp_normals[n[i]*3+2]});

    }

```

(The code above is part of the renderer.cpp file)

6. Native Library

The native library is a final destination combining the header and .cpp files responsible for other parts of the implementation. It's responsible for communicating with Kotlin via the "Java Native Interface", delivering the final stuff. Here you get a peak of what it looks like:

```

#include <jni.h>

#include <thread>

#include <atomic>

#include <memory>

#include <chrono>

#include <android/native_window_jni.h>

#include <android/asset_manager_jni.h>

#include "game.h"

#include "renderer.h"

```

```
// --- Globala pekare och tillstånd ---
std::unique_ptr<Game> game_instance = nullptr;
std::unique_ptr<Renderer> renderer_instance = nullptr;
std::atomic<bool> rendering = false;
std::thread render_thread;

void renderLoop(ANativeWindow* window, AAssetManager*
assetManager) {
    renderer_instance = std::make_unique<Renderer>();
    if (!renderer_instance->init(assetManager, window)) {
        // Rensa och avsluta om initiering misslyckas
        renderer_instance.reset(nullptr);
        ANativeWindow_release(window);
        return;
    }

    auto lastTime = std::chrono::high_resolution_clock::now();
    while (rendering.load()) {
        auto currentTime =
std::chrono::high_resolution_clock::now();

        float deltaTime =
std::chrono::duration<float>(currentTime - lastTime).count();

        lastTime = currentTime;

        if (game_instance) {
            game_instance->update(deltaTime);

            // Uppdaterat anrop: Hämta värden från game och
skicka till renderer

            renderer_instance->drawFrame(
```

```

        game_instance->getPlanePosition(),
        game_instance->getPlaneRotation(),
        game_instance->getCameraPosition(),
        game_instance->getCameraUp()

    );
}

}

// Rensa upp Renderer och EGL-kontext
renderer_instance.reset(nullptr);
ANativeWindow_release(window);
}

// --- JNI-brygga ---
extern "C" {

    JNIEXPORT void JNICALL
    Java_com_example_myflight_MainActivity_onSurfaceCreated(JNIEnv*
    env, jobject, jobject surface, jobject jAssetManager) {

        ANativeWindow* window = ANativeWindow_fromSurface(env,
        surface);

        AAssetManager* assets = AAssetManager_fromJava(env,
        jAssetManager);

        if (window != nullptr) {

            game_instance = std::make_unique<Game>();

            rendering = true;

            render_thread = std::thread(renderLoop, window,
            assets);

        }

    }
}

```

```
JNIEXPORT void JNICALL
Java_com_example_myflight_MainActivity_onSurfaceDestroyed(JNIEnv*
*, jobject) {
    rendering = false;
    if (render_thread.joinable()) {
        render_thread.join();
    }
    game_instance.reset(nullptr);
}

JNIEXPORT void JNICALL
Java_com_example_myflight_MainActivity_onSurfaceChanged(JNIEnv*
*, jobject, jint width, jint height) {
    if (render_instance) {
        render_instance->onSurfaceResized(width, height);
    }
}

JNIEXPORT void JNICALL
Java_com_example_myflight_MainActivity_onJoystickMoved(JNIEnv*
*, jobject, jfloat x, jfloat y) {
    if (game_instance) {
        game_instance->onJoystickMoved(x, y);
    }
}

JNIEXPORT void JNICALL
Java_com_example_myflight_MainActivity_onThrottleChanged(JNIEnv*
*, jobject, jfloat change) {
    if (game_instance) {
        game_instance->onThrottleChanged(change);
    }
}
```

```
}  
}
```

As you can see we are using C++ 17, with components such as `std::unique_ptr`. We can also see how the renderer is being activated in a time related implementation, as well as the window client being implemented. There is a lot of implementation taken from the NDK (Native Development Kit) which is maintained by Google providing several C/C++ functions, headers etc targeting Android "native" development. It makes stuff easier in a way - if you wouldn't be the one with the deepest of C knowledge being able to create drivers running on the hardware directly. And I'm also not sure if Google or Samsung would accept such code being active in applications available on their store platforms. The "JNI" in "JNIEXPORT" stands for the "Java Native Interface", which is explained in another part of this documentation.

7. Shaders

Here I'll explain how the shaders were implemented. But first a brief explanation of what a shader is:

- In this project, a shader consist of code written in GLSL, the "Open GL Shader Language"
- It's running on GPU devices that support the specific version of either OpenGL or in this case OpenGL ES 3.0, which targets mobile and embedded devices.
- With the technologies you are able to implement several graphical contexts and effects such as light, shadows, glow and colour fades.
- Shaders in OpenGL are either so called: fragment shaders or vertex shaders.
- Vertex shaders consist of attributes describing invisible contexts of e.g. a model.
- Fragment shaders contains mathematical values of e.g. colours, glows, lights and shadows.

Let's look at some shader code from the (sun) project:

```
precision mediump float;  
  
in vec2 v_uv;  
out vec4 fragColor;
```

```
void main() {  
    // Beräkna avståndet från mitten (0.5, 0.5)  
    float dist = distance(v_uv, vec2(0.5));  
  
    // FÄRGER:  
    // En väldigt ljus, nästan vit-rosa kärna för att simulera  
intensitet  
    vec3 coreColor = vec3(1.0, 0.95, 0.92);  
    // En djupare rosa/magenta färg för glorian  
    vec3 glowColor = vec3(1.0, 0.3, 0.65);  
  
    // FORM:  
    // Kärnan: En intensiv punkt i mitten som avtar snabbt  
    float coreIntensity = 1.0 - smoothstep(0.0, 0.4, dist);  
    coreIntensity = pow(coreIntensity, 4.0); // Gör kärnan mindre och  
skarpare  
  
    // Gloria: Mjukare avtoning ut mot kanten  
    float glowIntensity = 1.0 - smoothstep(0.2, 0.5, dist);  
  
    // KOMBINEARA:  
    // Blanda färgerna baserat på kärnans intensitet.  
    // Ju närmare mitten, desto mer av den ljusa "coreColor".  
    vec3 final_color = mix(glowColor, coreColor, coreIntensity);  
  
    // Alpha: Solen ska vara genomskinlig längst ut i hörnen  
    float alpha = smoothstep(0.5, 0.25, dist);  
  
    // Applicera premultiplied alpha
```

```
fragColor = vec4(final_color * alpha, alpha);
}
```

I'm sorry if the Swedish syntax comments could be confusing to the reader - it's simply how I've tried memorizing the technique for myself in future. In the code, from the file "sun.frag" you notice we find many components utilizing "vec3" - which translates to "Vector3", that said - a Vector x 3 math context. It's useful (as you can possibly imagine) with 3D contexts due to being divided in 3 different values such as X, Y or Z. "Smoothstep" is an interpolation function in GLSL which (interpolates) floating point values. That said - a middle value is returned as a result of being "between" two others. Very useful.

8. Game Controller

Input handling is managed through standard Android callbacks tailored for physical game controllers. The MainActivity captures motion events (onGenericMotionEvent) from connected joystick devices. The raw axis values (X and Y) are read directly from MotionEvent.AXIS_X and MotionEvent.AXIS_Y and immediately forwarded to the native C++ engine via the onJoystickMoved JNI function.

Throttle control is handled via key events (onKeyDown), listening for button presses (like Button A or B) on the gamepad, which trigger the onThrottleChanged native function. This setup bypasses the need for on-screen touch controls, providing a direct, low-latency interface for physical hardware.

Inside the C++ engine (game.cpp), these input values are stored in atomic variables to ensure thread safety. The game loop reads these values each frame to update the camera's orbital rotation, creating a smooth and responsive flight experience controlled entirely by the physical joystick.

```
const float JOYSTICK_SENSITIVITY = 2.0f;
const float JOYSTICK_DEAD_ZONE = 0.05f;
```

```
void Game::update(float /*deltaTime*/) {
    // Läs joystick-input
    float jx = joystick_x_raw_.load();
```



```
float jy = joystick_y_raw_.load();  
  
if (abs(jx) < JOYSTICK_DEAD_ZONE) jx = 0;  
if (abs(jy) < JOYSTICK_DEAD_ZONE) jy = 0;  
  
// Uppdatera kamerans rotation  
rotation_y_deg_ += jx * JOYSTICK_SENSITIVITY;  
rotation_x_deg_ += jy * JOYSTICK_SENSITIVITY;  
  
// Begränsa vertikal rotation (+/- 89 grader) för att  
tillåta titt underifrån  
if (rotation_x_deg_ < -89.0f) rotation_x_deg_ = -89.0f;  
if (rotation_x_deg_ > 89.0f) rotation_x_deg_ = 89.0f;  
}  
  
void Game::onJoystickMoved(float x, float y) {  
    joystick_x_raw_ = x;  
    joystick_y_raw_ = y;  
}
```



Thank you for reading! Regards, Sixten.