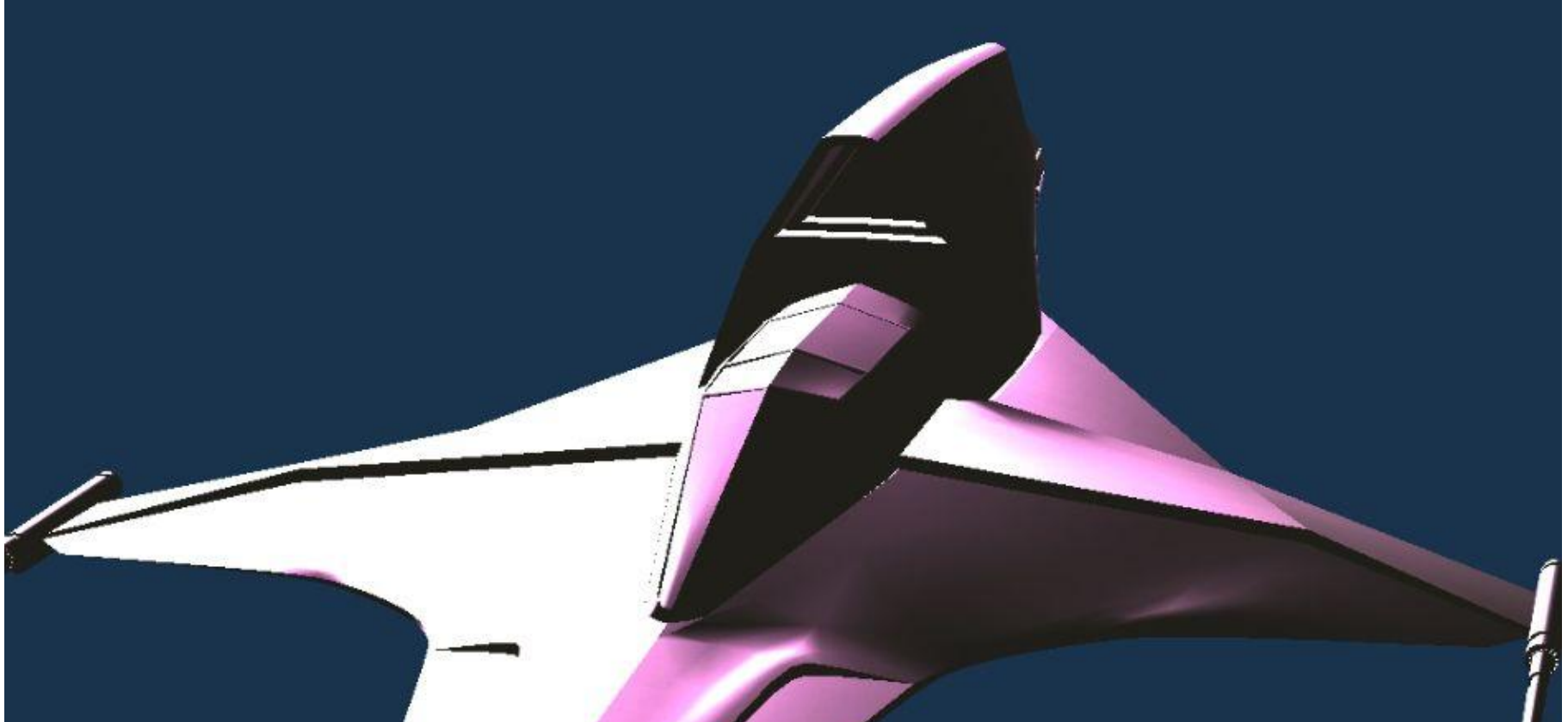


Exam Project, Sixten Björling



A Brief Presentation!

With my exam project I first stated that I was gonna develop a complete alpha for a multiplayer dogfight game for the latest Android devices using technologies like plain C/C++ with OpenGL. I later learnt that I shouldn't have water above my head through repeating such brave mission plans. However, through trying my best and concluding my knowledge, forced to spare plans to finish off something a least a little bit cool looking - I came to the point where I would be able to show an airplane model at least in 3D space with a bunch of effects along with a pink sun. The player is able to rotate the camera around with a game controller connected to the Android phone. I at least became somewhat satisfied with what I did and I would love to end up busy with similar technology for the future.

Java Native Interface! Kotlin calling C/C++ libraries?

The Java Native Interface is the bridge that lets Kotlin or Java code communicate with native C or C++ libraries. On Android, this is essential when you want high-performance code, direct access to hardware, or a custom rendering pipeline written with the NDK. Instead of rewriting everything in Kotlin, JNI allows the app to load a native library and call its functions as if they were part of the regular Android codebase. The idea is simple: Kotlin declares a function as “external,” and the native side provides the actual implementation compiled into a shared library.

In practice, this means the Android app starts in Kotlin, but whenever it needs to run heavy tasks — like rendering, physics, or math — it hands control over to the C/C++ layer. The native library is loaded at startup, and the two sides exchange data through well-defined function signatures. This keeps the architecture clean: Kotlin handles lifecycle and many times UI as well, while C/C++ handles performance-critical work. JNI is the glue that makes this possible, allowing both worlds to work together inside the same application.

Renderer! What is such thing?

A renderer is essentially the part of an application responsible for turning mathematical data — positions, rotations, colors, and camera information — into actual pixels on the screen. Instead of thinking of it as a single function, it's more like a small engine that prepares the graphics context, uploads data to the GPU, and repeatedly draws each frame. On desktop platforms this is often handled by large frameworks or engines, but on Android, especially when using the NDK, you build this pipeline yourself in C or C++, which gives you full control over how the graphics are produced.

Shaders! Have you heard about those?

Shaders are small programs that run directly on the GPU and define how your 3D scene is transformed into pixels on the screen. Instead of being written in C or C++, shaders use GLSL, a C-like language designed specifically for parallel execution on graphics hardware. The idea is that the CPU prepares the data — positions, colors, matrices, textures — and the GPU uses shaders to process that data extremely quickly, one vertex or one pixel at a time. In OpenGL ES 3.0, this usually means having at least two shaders: a vertex shader that handles the geometry, and a fragment shader that determines the final color of each pixel

OBJ files! A model that is?

An .obj file is a simple 3D model format written entirely in plain text, which makes it easy to inspect and load without relying on heavy libraries or binary parsing. The file consists of lines that describe different kinds of geometric data, such as vertex positions, texture coordinates, normals, and faces. Each line begins with a small keyword that tells you what type of data follows, and the rest of the line contains the numerical values. Reading an .obj file is therefore mostly a matter of scanning each line, checking what keyword it starts with, and storing the values in the appropriate buffers. Once the data is collected, it can be sent directly to the GPU through OpenGL or a similar API. Because the format is text-based and predictable, it's a straightforward way to load simple models when building your own rendering pipeline.

Quaternions, Matrices, MATH! Could you explain briefly?

Quaternions are a compact way to represent rotation in 3D space. Instead of using three angles that can twist and lock up (gimbal lock), a quaternion stores rotation as a four-component value that can smoothly rotate objects in any direction. They're lightweight, stable, and perfect for animations, camera movement, and anything that needs smooth orientation changes.

A matrix is basically a small grid of numbers that transforms coordinates. In 3D graphics, matrices are used to move, rotate, scale, and project objects onto the screen. A single 4×4 matrix can describe a camera view, a perspective projection, or the position of an object in the world. GPUs love matrices because they can apply them extremely fast to thousands of vertices at once.

Together, quaternions and matrices form the backbone of real-time 3D rendering. Quaternions describe *how* something is rotated, and matrices describe *where* it is

Thank you for reading!

This project have learnt me that I shall never again propose to be finished with goals such as multiplayer gameplay support released in alpha versions to Google Play to speak of first! However, I managed to "render" a 3D model with lights, shadows, colour effects along with a sun, being able to rotate around it with a game controller. That said, to my conclusion is that development of native applications in C/C++ needs time and effort as well as knowledge to be effective. I hope by completing something that demonstrates willingness to learn an document, I will receive an internship with related subject fields, learning through repeating, reading, listening and testing. But it has been a lot of fun and interesting to be active with this exam project. I hope that you found this presentation the be concise and descriptive.

Regards, Sixten!