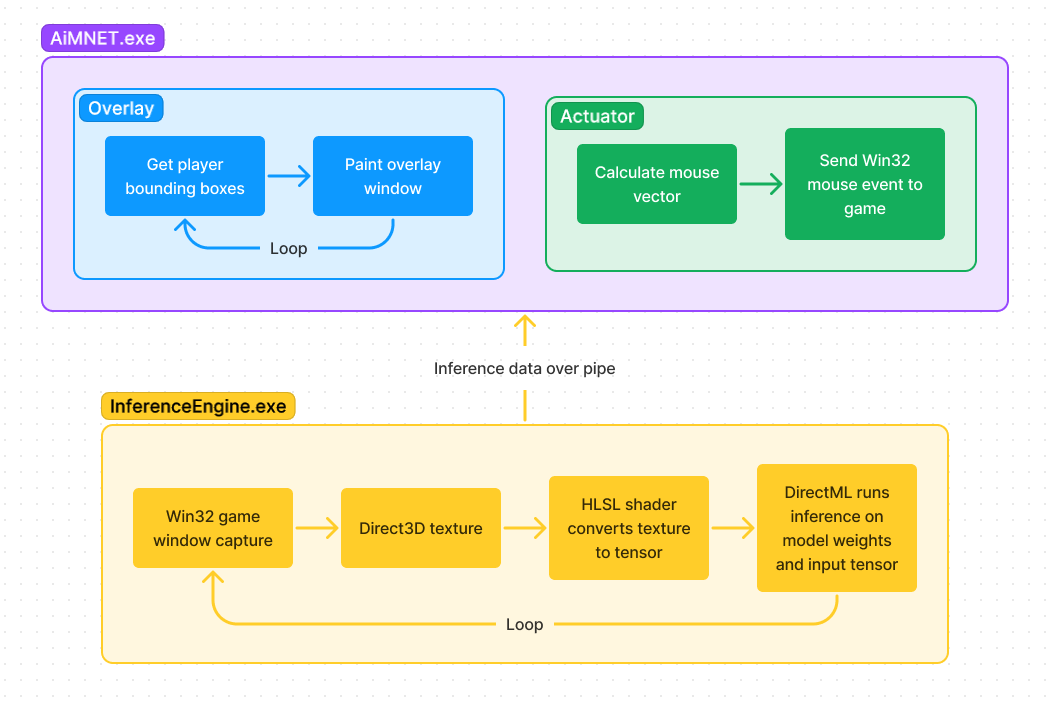
AiMNET Architecture Overview

This document describes the overall design of AiMNET, as well as a brief overview of the code.

# Software Components

AiMNET is broken up into two independent programs, the inference engine (server), and the GUI overlay and configuration window (client). The following diagram gives an overview of their interaction:



InferenceEngine.exe is usually launched by the client, but it can be run independently. It follows a simple event loop, shown above. It captures frames from the game window (via the Win32 PrintWindow() function), uses DirectML to run the Yolov4 model inference on the GPU, and then outputs the player positions that it finds (along with some other statistics) in a plain-text format to standard output. When the server is run on its own (or from Visual Studio) you can see the messages it writes out.

When the client is started, it spawns an InferenceEngine.exe process as well. The client event loop is more complicated because it has to handle GUI events as well, but it basically boils down to reading the data that the server sends it and updating a list of player positions. The overlay window then continually updates, rendering the player bounding boxes onto the screen.

When a bounding box is found to intersect with the aiming reticle, the actuator kicks into action. In a separate thread, the “ActuatorWorker” is constantly waiting for instructions. When told to shoot, it calculates the proper mouse movement based on the position of the target in the game and sends a Win32 mouse event to the game.

# Code Overview

The server code lives in InferenceEngine\, and the client code lives in Client\. The .qml files in the client describe the GUI, they aren’t responsible for any of the main logic. The client’s C++ files are quite simple, and should be relatively easy to understand. They are broken up into classes which match the names of the components described above.

The server code is a bit harder to understand. The DirectX 12 code is quite dense. The interesting files are WindowCapture.h and .cpp, and yolov4.h and .cpp. yolov4.cpp in particular is very complicated, but the interesting methods are:

* InferenceEngine::TakeAndUploadScreenshot() which handles converting the captured window image to a Direct3D texture
* InferenceEngine::Render() which runs every frame and handles outputting the results of the Yolo inference and setting up the input for the next iteration of the model

# Optimizations

A lot of work went into optimizing the server code to run as quickly as possible. Here’s a short overview of our work to get here:

1. Inference code in Python running entirely on CPU: **~3 FPS.**
2. Python inference code running on Nvidia GTX 1050: **~11 FPS**. Better, but doesn’t work on AMD graphics cards.
3. Inference code re-written in C++ using DirectML: **~11 FPS**. Not much progress, but runs on AMD cards now.
4. Optimized texture uploading in DirectML code: **~15 FPS**. A bit better, getting decent results now.
5. Parallelized all the CPU processing using OpenMP. This means that the program will automatically spawn as many threads as it needs and spread the work out between them, eliminating most of the CPU overhead. Also took advantage of SIMD (single instruction multiple data) instructions to vectorize all the calculations on CPU: **~22 FPS**

That’s where we are currently. 22 FPS gives very good results, and most of the improvements still needed aren’t related to the machine learning, but to what the actuator actually does with the data.