I spend the most of my free time working on personal projects that need a diverse set of skills. I have had experience from working with computer graphics (openGL, ray-tracing), to computer vision (openCV), to, most recently, full stack application development (webapp and android app). You can check out some of my projects’ demos in the Project tab above.

Setting up the frontend

For the frontend, I picked ReactJS as the framework for the making this webapp. I used bot TailwindCSS and CSS for styling and CSSTransition for mounting/unmounting animations. For video streaming, I utilized SocketIO to send and receive messages from the server.



Text

Description automatically generated

Setting up Backend

For the server, I opted to use Flask. This was due to the fact that, as compared to NodeJS, it was simpler to get my facial detection program up and running. I also used the Model-View-Controller-Service design pattern because most of the models were already implemented in my previous project and I only needed to add the controller and the service layer.

To integrate my facial detection program to the backend. I compiled my python scripts into a library and upload that to PyPI. From there, I can just install the library using pip.

“pip install --upgrade glyFacialDetection”

Then, when a frame arrived at the server, I can just pass that through my program and send back to the client the processed frame.

Text

Description automatically generated

ADDITIONAL SETUP

To reduce video latency, I also provided the option to only send the coordinates of the faces in the image to the clients rather than the entire processed images. This also meant that I had to have the frontend draw the faces’ bounding boxes on the original frames before displaying them.

NEVA SETTLE

Well that was it huh. The frames the clients sent were processed and returned and displayed. The app is now working fine. GGEZ.

SIKE!

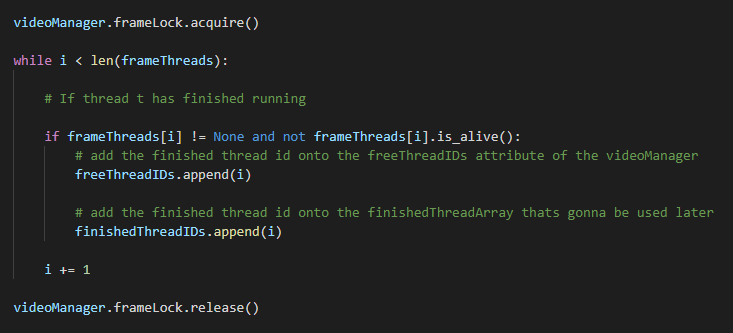
It takes a varied length of time for each frame to complete the round trip. Frames are only returned after the server has completed the facial detection on that frame. And the time it takes for that frame to be transported back to the client varies due to the nature of the internet. As a result, frames might get delivered to clients out-of-order (problem 1), or the video displayed isn't smooth since the intervals among frames change (problem 2).

Out of order frames

To address this, I created a class named ServerVideoManager (SVM) in the backend to generate a unique object SVM for each client. These SVMs exist during their clients’ connections. SVM manages all the parallel threads that are doing face detection on the frames, allowing the server to skip sending late frames to the clients.

When an incoming frame with frameID arrives, the SVM accepts that frame, launches a new thread to detect faces on the frame, and keeps track of the status of that thread. SVM also has a background thread that runs after every interval to monitor all of the facial detection threads.

When the background thread finds a thread/threads that finished running, it collects all the finished threads.



Then it extracts the latest frames among all of them. If the latest frame has frameID that is less than the previous sent back frameID, it is a late frame. As a result, the SVM sends nothing to the client. If the latest frame has frameID after the last sent back frameID, SVM will send the latest frame information to the client. Finally, SVM clears off all completed threads to create room for incoming frames.

This helps eliminate out-of-order frames. Even though the client loses some information, with the constant stream of input coming into the server, that skipped frame is insignificant.

Text

Description automatically generated

The idea behind fixing the inconsistent time interval between frames is simple. Whenever a frame arrives from the server, I examine the frameID and determine the time that frame intended to be drawn based on the time of the previous displayed frame. To do this, I constructed a class called VideoSocketService (VSS) in the frontend to handle transmitting, receiving, and displaying the frames. Each client will have one instance of an VSS object (each connection with the server).

SVS used the following formula to find the correct time to draw each frame:

Let supposedDrawnTime = Math.max(now, lastDrawnTime + (frameID – lastFrameID) \* frameTimeInterval);

This ensures that if the frame was returned too soon, it will be displayed at an even interval following the previous one. And if the frame was returned late, it will be displayed immediately.

However, notice how I said “VSS used …”. The formula above only cares about spacing the next frame to be exactly (frameID – lastFrameID) intervals away. This, combined with the fact that each late frame adds some amount of delay, causes the overall latency of the video to skyrocket (2 seconds of delay after 10 seconds of playing).

To counteract this, I engineered a solution and implemented it in the VSS class.

For every frame, I keep track of the time it was first sent from the client to the server and the time when it arrives at the client. The difference between the two timestamps is the delay time of that frame.

When a frame arrives, VSS compares that frame actualDelay to the desiredDelay (desiredDelay = prevDelay – delayReducingOffset). The delayReducingOffset value (which I set to be 18ms) is the key factor that helps counteract the cumulative delays added by late frames. It lowers the desiredDelay by a small amount for each incoming frame, which helps smoothing out the reducing-overall-delay process that VSS is doing.

If desiredDelay > currDelay, then draw the frame after (desiredDelay – currDelay)

If desiredDelay <= currDelay , then draw the frame right away.

As a result, the cumulative delay is no longer an issue, and the frame time variance between frames is kept to a minimum, eliminating jitter/stutter. All of this comes at the cost of slightly higher average latency/delay.

Text

Description automatically generated