NaviRice: A Holographic Navigation Heads Up Display for Cars

A Major Qualifying Project Report submitted to the faculty of WORCESTER POLYTECHNIC INSTITUTE in partial fulfillment of the requirements for the degree of Bachelor of Science By:

Can Alper, Alexander Gaines, Binam Kayastha, Yang Liu

April 26, 2018

Project Number: MQP MXC 0562

Advisor: Professor Michael J. Ciaraldi

Abstract

The purpose of this research project is to help drivers navigate by displaying holographic images on their windshield as a heads-up display (HUD). With the HUD, drivers can keep their eyes on the road, without looking down to their phone for directions. To do this, our project uses a large monitor on the car's dashboard to display images on the windshield. A Microsoft Kinect is used to track the head. A phone application is used to enter directions and acts as a GPS receiver. The information from the Kinect and the phone app is sent to the rendering service. Finally, the rendering service creates a warped image on the monitor which accounts for the curvature of the windshield, displaying an image to the user. Throughout the project, we explored different methods to achieve this and aimed to get holographic directions to display at least 90 Hz.

Contents

1 Introduction

Our project is to create a Holographic Augmented Reality using a Heads Up Display (HUD) to help automobile drivers navigate. We display 3D holographic objects in real-world positions in the perspective of the driver. The authors of this research paper did everything from the concept to the end product. We fully implemented a functioning system in a car by removing the dashboard, mounting all the necessary devices, and writing the software which connects them. To our knowledge, there are no laws in the state of Massachusetts preventing us from doing this. This research paper shows all of our attempts, both unsuccessful and successful.

The purpose of this project is to help drivers keep their eyes on the road without having to look down to their phone or GPS device for navigation. In the two seconds they look away from their windshield for directions, a driver going at 65 mph can pass two-thirds of a football field [1]. A HUD allows them to receive information while focusing on the road, and can potentially save them from getting into a car accident or crash. Future work can be done to further aid the driver, such as highlighting the correct lane to be on for a turn, highlighting destination buildings, blinkers for blind spots, and more described in detail in the Future Work section 5.4.

To our knowledge, no project displays AR objects using the windshield of a consumer car and keeps the objects at the same holographic 3D locations using head tracking. We talk about similar projects in the background.

2 Background/Related Work

2.0.1 Giving objects a 3D illusion by tracking the head

Johnny Chung Lee developed a project where he created the illusion of 3D objects staying in the same physical location on a TV screen. When you place your finger in front of you and move your head, the background moves in relation to the finger. This is the same effect the objects on Lee's screen mimic: As the head moves, the objects on the screen try to stay in the same position in the real world, either in front or behind the TV. This can easily be seen in his recording of the project [2]. The details of how he tracks the head is explained in Section 3.2.1.2.

2.0.2 Heads Up Displays

A Head Up Display, or HUD, is a display that lets the driver see information without looking down, as the information is usually projected on the windshield. Unlike any other system, we use our own novel method explained in the paper but felt it was important to note similar projects.

2.0.2.1 HUDWAY Cast

HUDWAY Cast is a device which connects to your phone and uses a mini projector to display navigation information on a piece of glass set on the dashboard. The basic idea was to give drivers "navigation-critical information" while keeping their eyes on the road [1]. Note that the HUDWAY Cast only covers a small area, due to its reliance on a separate reflective piece of glass. Also, note that it does not create a 3D illusion of objects, and thus doesn't need to know the position of the driver's head.

2.0.2.2 Fighter Jets

Fighter jets utilize a HUD to help the pilot view various information. This information can be either overlaid in the real world or static to the pilot's view and it might include things like friendly or enemy jets, bomb drop calculations, the direction of the plane and so on. Since the pilot's head is assumed to be at a fixed location the display properties do not change. This is because the holographic objects are so far away and the head location is insignificant in the process of creating the illusion and can be ignored. However fifth generation fighter jets, such as Lockheed Martin F-35, utilizes a full view Augmented Reality helmet that allows seamless pilot-informatics integration.

2.0.3 Previous work on Head Detection

The importance of creating a 3D illusion is described in Section 2.0.1. Currently, there are many different methods that exist. We build on some of these solutions, but ultimately end up creating our own head tracking program via machine learning as described in the methodology section 3.2.

2.0.3.1 Using Wii Remote

One method of tracking the head location is by using the Wii Remote as Johnny Chung Lee demonstrated [2]. The location of the Wii Remote can be determined by the sensor bar and vice-versa. Since the sensor bar is just 2 IR dots, we can swap it with two IR lights put on some glasses. Then we place the Wii Remote where the sensor bar would normally go, and move our head with the glasses, which allows us to get a 3D location of it relative to the Wii Remote. An algorithm can compute the depth based on how far apart the dots are.

In Section 3.2.1.2, in our methodology, we further explain why this method doesn't meet our standards.

2.0.3.2 Open TLD

OpenTLD is a method where you can specify what to track by drawing a box around it and is able to continue tracking that piece of the image. It was written in C++ and unfortunately is no longer being worked on. We were unable to rerun the program or replicate the results [3].

2.0.3.3 Haar Cascades

Haar Cascades are an old way of detecting or recognizing certain images. It is similar to machine learning in that it needs to be trained. Haar Cascades work by matching multiple features (set of gradients) which define the target of recognition (in this case, a face). Since Haar Cascades can include upwards of 160 thousand features, OpenCV attempts to optimize their implementation. The Haar Cascades are run on multiple scaled versions of the image, from small to large. By running them on small images, most pixels can be ignored before running it on the original image, therefore saving computation time.

The computation speed of Haar Cascades decreases with the increase of pixels in the image. The performance also decreases as the number of times the Haar Cascades are scaled increases. The accuracy of the Haar Cascades increases if they are run on more scaled images. There is an inverse relationship between speed and accuracy with the number of scaled images for Haar Cascades. A detailed explanation can be found on the OpenCV site [4].

2.0.3.4 Machine Learning

Machine learning programs for face detection exist, but are extremely slow and require a large amount training data. We still can pursue this because our use case is very specific: Detecting a head in a car, which limits the number of possible faces (no need to detect small faces or baby faces, etc.)

2.0.3.4.1 Convolutional Neural Networks

In recent years Convolutional Neural Networks (Conv Nets) became very popular in the Computer Vision field. This is because Conv Nets try to replicate the visual cortex found in the eyes of humans and many other animals. This feature of the Conv Nets provides superior learning capabilities over regular Neural Networks for vision-related problems.

Unlike regular Neural Networks, Conv Nets focus on specific areas of an image. The term for this area is called a kernel (aka Sliding Window). Kernels are small images extracted from the original image. For example, let's say we have an image with a size of 100x100 with three channels, which makes this an image of 100x100x3. If we were to extract kernels of 25x25x3 from this image we would get a data structure of 75x75x25x25x3 as seen in Figure 1. The 75x75 portion of the structure comes from the fact that an image of 25x25 can only be shifted through an image of 100x100 a total of 75 times.

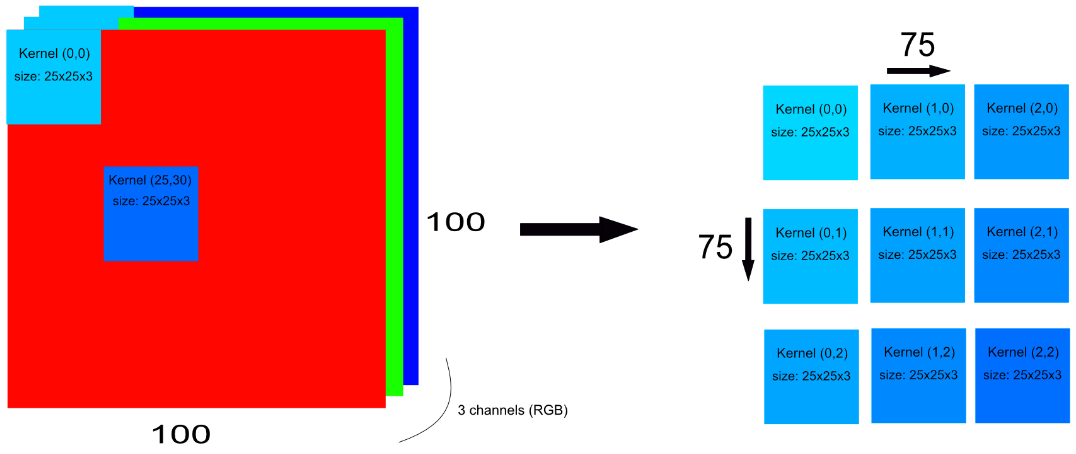
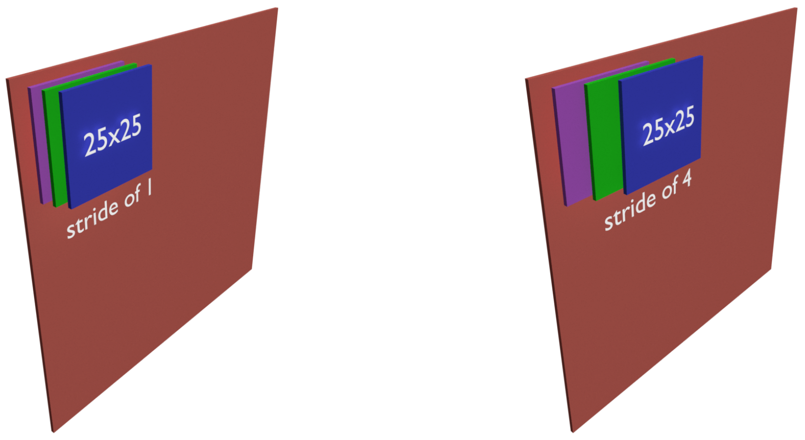


Figure 1: Kernel extraction

Another common term used in Conv Nets is stride. Stride defines the distance of each kernel from one another. In Figure 1, kernels are extracted with a stride of 1, which means that the top left coordinate of the first kernel is (0,0) and the top left coordinate of the neighbor kernels are (1,0), (0,1) and (1,1). If we used a stride of 4 we would get neighbour coordinates of (3,0), (0,3) and (3,3) as seen in Figures 2b and 2a. When the stride changes from 1, the data structure of the generated kernels change too. For example, an image of 300x300x3, a kernel size of 10x10x3, and a stride of 3x3 will create a data structure of 96x96x10x10x3.



(a) Stride of 1 visualization (b) Stride of 4 visualization

Figure 2: Examples of kernels extracted with different strides on an image

One of the recent discoveries with Conv Nets is the ReLU (Rectified Linear Unit) activation function. Activation functions are nonlinear functions that allow us to update synapse weights by providing the Gradient Descent algorithm with the required gradients. The gradients are calculated by finding the error between the prediction from an input and the correct label for that input. However when the synapse weights reach 0, ReLU becomes ineffective and to prevent this Leaky-ReLU is utilized. Before ReLU, Sigmoid and Hyperbolic Tangent functions were the most popular activation functions. All of the mentioned functions can be seen in Table 1.

Activation Function	Equation
Sigmoid	$\frac{1}{1-e^{-x}}$
Hyperbolic Tangent	$\frac{e^x-e^{-x}}{e^x+e^{-x}}$
ReLU	$\max(0, x)$
Leaky-ReLU	$\max(0.01 * x, x)$

Table 1: Different types of activation functions

2.0.4 Rendering

2.0.4.1 Real-time vs Offline

The speed at which an image is rendered by a computer is either considered "real-time" or "offline". If the image can be produced faster than 24 frames per second, it is generally considered "real-time", as it is rendered fast enough to be dynamic and responsive to input from a user. Most computer games are rendered in real-time, although there are some exceptions [5]. "Offline" renders are generally a lot more advanced and detailed, and in some cases can take days or even weeks to complete a single image.

2.0.4.2 Raytracing vs Rasterization

There are two primary ways of rendering a computer generated 3D scene to an image. The first, and most widely used for real-time, is rasterization.

The basic idea of rasterization is to take a shape, find where it "lands" on the screen, and then fill in the pixels that it covers. For axis-aligned rectangles, this is very easy. Find two of the corners, say at pixel coordinates 5,7 and 13,11, and then just fill in the pixels in that range. For triangles, it is a little more complex, but still relies on the basic principle of finding where the polygon fits on the screen and then "filling in" the pixels that it covers, much like a children's coloring book.

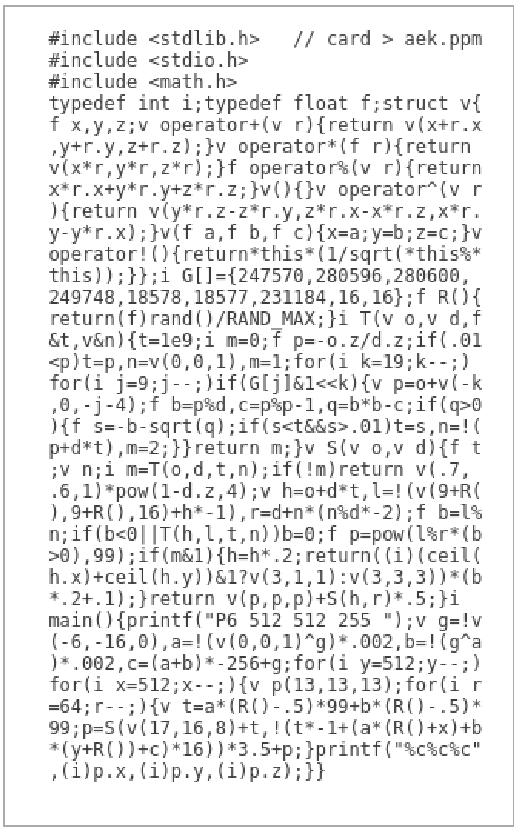
Modern rasterization works on polygons, specifically triangles. Modern graphics processing units only draw triangles. In order to draw a rectangle, or "quad", one would utilize two triangles sharing a single edge.

The flow of a basic rasterizer is this:

```
For every triangle in the scene {
    Figure out where the 3 vertices align on the screen
    Determine rectangular bounds of those three vertices (such as 5,7 and 13,11)
    For every pixel in that rectangular bounding box {
        if pixel is part of the triangle & pixel is "foreground" {
            Color that Pixel
        }
    }
}
```

Modern graphics hardware has transistors specifically designed to do exactly this. They can do this process at over 16 billion triangles per second [6].

The second method is raytracing. Raytracing is almost the inverse of rasterization, as the process now works on a per-pixel basis instead of a per-triangle basis. Instead of taking a triangle and determining the pixels that it covers, raytracing takes a pixel and determines which triangles cover it.



```
#include <stdlib.h>   // card > aek.ppm
#include <stdio.h>
#include <math.h>
typedef int i;typedef float f;struct v{
f x,y,z;v operator+(v r){return v(x+r.x
,y+r.y,z+r.z);}v operator*(f r){return
v(x*r,y*r,z*r);}f operator%(v r){return
x*r.x+y*r.y+z*r.z;}v(){}v operator^(v r
){return v(y*r.z-z*r.y,z*r.x-x*r.z,x*r.
y-y*r.x);}v(f a,f b,f c){x=a;y=b;z=c;}v
operator!(){return*this*(1/sqrt(*this%*
this));}};i G[]={247570,280596,280600,
249748,18578,18577,231184,16,16};f R(){
return(f)rand()/RAND_MAX;}i T(v o,v d,f
&t,v&n){t=1e9;i m=0;f p=-o.z/d.z;if(.01
<p)t=p,n=v(0,0,1),m=1;for(i k=19;k--;)
for(i j=9;j--;)if(G[j]&1<<k){v p=o+v(-k
,0,-j-4);f b=p%d,c=p%p-1,q=b*b-c;if(q>0
){f s=-b-sqrt(q);if(s<t&&s>.01)t=s,n=!(
p+d*t),m=2;}}return m;}v S(v o,v d){f t
;v n;i m=T(o,d,t,n);if(!m)return v(.7,
.6,1)*pow(1-d.z,4);v h=o+d*t,l=!(v(9+R(
),9+R(),16)+h*-1),r=d+n*(n%d*-2);f b=l%
n;if(b<0||T(h,l,t,n))b=0;f p=pow(l%r*(b
>0),99);if(m&1){h=h*.2;return((i)(ceil(
h.x)+ceil(h.y))&1?v(3,1,1):v(3,3,3))*(b
*.2+.1);}return v(p,p,p)+S(h,r)*.5;}i
main(){printf("P6 512 512 255 ");v g=!v
(-6,-16,0),a=!(v(0,0,1)^g)*.002,b=!(g^a
)*.002,c=(a+b)*-256+g;for(i y=512;y--;)
for(i x=512;x--;){v p(13,13,13);for(i r
=64;r--;){v t=a*(R()-.5)*99+b*(R()-.5)*
99;p=S(v(17,16,8)+t,!(t*-1+(a*(R()+x)+b
*(y+R())+c)*16))*3.5+p;}printf("%c%c%c"
,(i)p.x,(i)p.y,(i)p.z);}}
```

Figure 3: A minimalist raytracer, from [7]

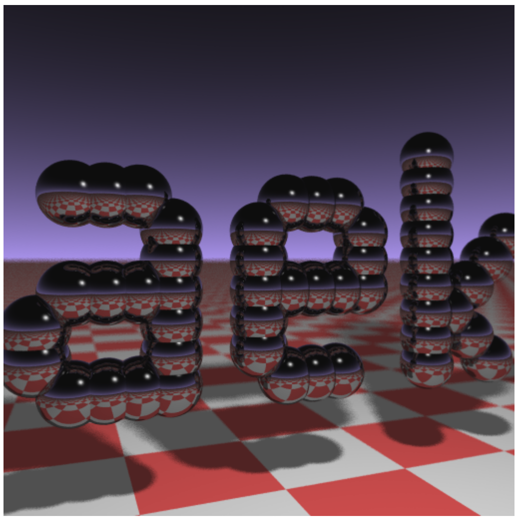


Figure 4: The image which the code produces, from [7]

Ray-tracing can be a very simple algorithm to implement. The business card raytracer is a C++ program that ray traces an entire scene, complete with shadowing, reflections, and depth-of-field effects that fits its source code entirely on the back of a business card [7, 8].

The basic flow of a raytracer is this:

```
For every pixel on the screen {
        "cast a ray":
            For every surface in the scene {
                Determine whether the ray hits the surface
            }
        pick the intersecting surface closest to the ray's start
        color that pixel (Potentially casting more rays and "bouncing"
            off the surface to simulate reflection, refraction,
            or lighting effects. This can be done easily with recursion.)
    }
```

The main benefit of rasterization is its computation speed. The process of "filling in" a triangle is a very cache-friendly algorithm. Raytracing, however, has many issues with cache friendliness when rays "diverge". The entire process of checking all triangles for every ray is also very slow on its own, and there is a large amount of ongoing research to speed this process up.

Doing any sort of complex effects or simulations with rasterization requires many "hacks". Shadows, reflections, light scattering, refraction, or any other properties of light are rather difficult to realistically render using rasterization and may require many passes to supplementary buffers, fake geometry, or other "tricks". Generally, all rasterized effects are just a convincing method of approximating or faking something that light does. For example, accurate reflections using only the rasterization is impossible. To have any sort of accuracy on a surface that isn't perfectly flat requires supplementary raytracing in addition.

Pure raytracing, on the other hand, can do these all by just casting more rays, seemingly "tracing" the path of a photon as it flies from the light source to the simulated camera. However, the sheer number of rays that are needed to be cast to do these accurate effects with acceptable quality is quite large.

Raytracing is also not limited to triangles. Many raytracers use basic shapes, or "primitives" such as spheres, cubes, cylinders, etc. These are actually simpler to compute intersections with, compared to the "advanced" triangle. When developing our naive raytracing method, we experimented with using a mathematically defined curve to express the curve of the windshield.

3 Methodology

3.1 System Architecture

3.1.1 Overview

There are four main components to this project:

- Head Tracking: Detecting where the driver's head is in 3D space inside the car.
- Navigation: Detecting where the car is in relation to where the driver wants to go, and feeding the data to the rendering server.
- Rendering: Rendering objects on the windshield through the reflection of a monitor on the dashboard, based on the head position to give the object the 3D feel.
- Hardware: The hardware needed in order to achieve the three above.

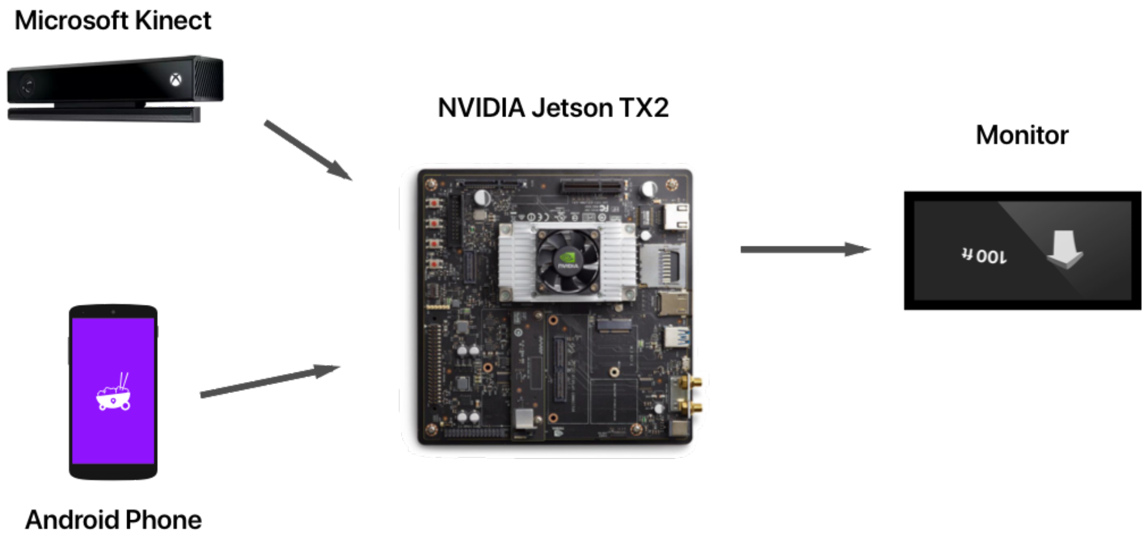


Figure 5: Overview of the components of our project

3.1.2 Microservices Architecture

Our project requires many moving parts to communicate with each other. We were inspired by the microservices architecture because it allows each service to swap, which makes it easier to experiment with different techniques. One example of this is when we tried using Haar Cascades and Machine Learning: It was very easy for us to swap in the Machine Learning model and replace Haar Cascades.

Another benefit of using microservices is that we could efficiently utilize our various skillsets, and program in different languages. Our project was written in C, C++, Python3, and Kotlin.

One thing to note is that we did not have an API Gateway as it would incur too much latency in our time-sensitive program. Therefore, we simply used a Client-Server architecture between the different services for communication. It is also worth noting that only the connection between the Phone and Rendering Service was over Wi-Fi, and everything else was directly connected to each other via ports/Ethernet cables. All of the connections used the TCP/IP protocal.

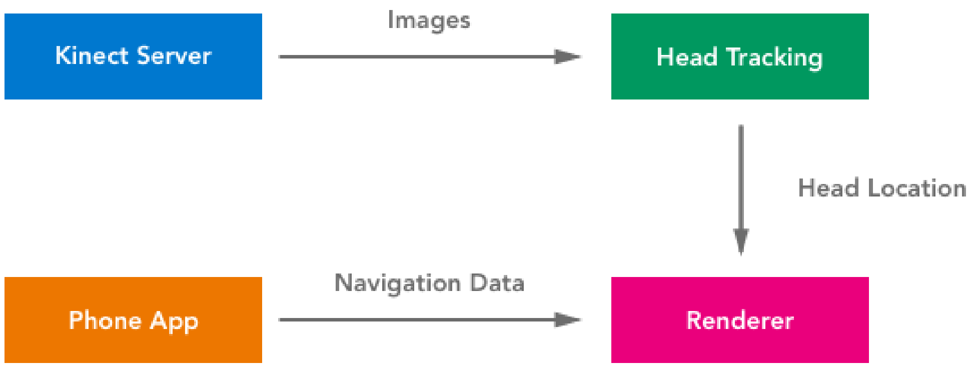


Figure 6: Microservices

3.1.3 Protobuf

In order to communicate effectively with these services across different languages, we decided to use Google's Protobuf, which stands for Protocol Buffer. Google provides libraries in different languages which read Protobuf files in those respective languages. It also allows us to store data in a structured byte form easily.

Protocol Buffer allows us to define the structure of the packets in *.proto* files. Our *.proto* files can be found in Appendix B. These files can then be read into the programming languages of our choice. It also provides helper functions to print out the string representation of the packet, greatly reducing the amount of time we spent on debugging.

3.1.3.1 Issues with Protobuf

Protobuf did not have good support for the C language. As the rendering service was all written in C, we faced many difficulties getting it hooked up with the rest of the applications. Our solution was to create a C++ wrapper around our C code which would communicate to and from Protobuf, changing the Protobuf data into structs.

We also ran into an issue where two different versions of Protobuf were required to communicate between our services, and we had to be careful how we installed them on the shared device (Jetson) which ran both the HeadTracking and Rendering services.

3.1.4 Kinect Server

We created a streaming server to distribute the images. This server utilizes Protobuf. To acquire the images from the Kinect2, we used the open source library called libfreenect2. We also threaded and optimized the server to use minimal resources and to be performant.

3.1.5 Positional Server

Our positional server relayed the last position of the driver's head to the clients. In our specific use, the Machine Learning program is the server and the rendering service is the client. To achieve this, the server needed to send three float values. These float values are the x and y positions between -1.0 to 1.0, -1.0 being the most left side of the Kinect's view and 1.0 being the most right side. Similarly, -1.0 is the bottom and 1.0 is the top. The third float value is the distance of the detected head from the Kinect in millimeters. Since this is very simple information we decided to not utilize Protobuf for this server-client relationship. When the client asked for the service our server simply flushed the three floating point values mentioned to the TCP/IP socket.

3.1.6 Phone App Client to Rendering Server

We used the following *.proto* file to serialize current navigation step into bytes and sent it from the phone App to the rendering server.

```
// step.proto
syntax = "proto2";

package navirice.proto;

message Step {
  required double latitude = 1;
  required double longitude = 2;
  required string description = 3;
  required string icon = 4;
}
```

The latitude and longitude represent the ending GPS location of the current step. The description contains the turn by turn instruction of the step. The icon field contains the corresponding id of the icon saved in the rendering server.

The size between each Step message varies due to the length of instruction string and icon name. It is very challenging for the server on the other side of the network to reconstruct the message without knowing its type and length. Thus, we also designed a request header message to help overcome this issue.

```
// step.proto
syntax = "proto2";

package navirice.proto;

message RequestHeader {
     enum Type {
          CURRENT_STEP = 0;
          CURRENT_LOCATION = 1;
          CURRENT_ACCELERATION_FORCE = 2;
          CURRENT_ROTATION_RATE = 3;
          ON_BUTTON_CLICK = 4;
     }
  required Type type = 1;
  required uint32 length = 2;
}
```

3.2 Head Tracking

To create the illusion of 3D objects, the rendering service displays images which appear to stay in the same physical location in the real world based on the driver's head location. The head location needs to be mapped to a 3D coordinate in the rendering service. We chose a device which allows us to know the relative distance of the head from the device. We then explored methods which allow us to get the location of the head using the chosen device. Our goal was to get the head location to update so the rendering service can display images at a rate of least 60 fps, at which the average human cannot perceive flickering [9]. Ideally, we aimed for 90 fps, which is what VR developers use as the baseline to avoid distortion/nausea [10].

3.2.1 Imaging and Tracking Devices

In order to track the driver's head in 3D space, we needed devices with sensors which can detect depth. There are two main types of devices which can do this - imaging and tracking devices.

3.2.1.1 Imaging device vs Tracking device

Tracking devices are fast and give an exact location and depth of the head, however, they require the user to put on a wearable device. Imaging devices give an image and a depth map. The location of the head must be found by a separate detection program. The depth can be found by mapping the detected location to the depth map. Because a custom detection program is needed with Imaging devices, it requires more processing power, but the users head can be detected without wearing any extra devices. Using an imaging device also gives us the freedom to detect any part of the head, face, eye, etc.

3.2.1.2 Wii Remote - Tracking

We explain Lee's methods of tracking the head using a Wii Remote in our background section 2.0.3.1.

One issue with this method is that it relies on the head (and 2 IR dots) facing directly towards the Wii Remote. Turning the head may cause the Wii Remote tracking algorithm to think the head suddenly got really close to the monitor, as the two IR dots would be near each other. One way to combat this is to use 2 Wii remotes in slightly different locations.

3.2.1.3 Kinect2 - Imaging

The Kinect2 captures three types of images: RGB, IR, and Depth. The differences are shown in the table below:

Type	Size	FPS	Horizontal FOV	Vertical FOV
RGB	1920x1080	15-30	84.1°	53.8°
IR	512x424	30	70.6°	60°
Depth	512x424	30	70.6°	60°

Table 2: Kinect2 Specifications

IR and Depth data is extremely useful for our project, as RGB images rely on visible light. IR and Depth data will be the same regardless of how much sunlight is present, which allows us to track the head even at night. The RGB data is also generated by a camera that is offset from the IR and Depth camera, which would need extra work and computing power to correct for the visual shift.

Generally, the Kinect2 only transfers data to the XBox, so we bought an extra extension block which allowed us to send the data to computing devices. We informally call this the "adapter brick".

The Kinect2 can detect the depth of objects within a range 0.3 to 5 meters, which is enough to detect the driver's head from its mounted location on the dashboard of the car.

The Kinect2 is also not a health hazard at close ranges, compared to the Kinect1. The Kinect1 used many infrared lasers to create a pattern to detect depth, which could cause eye damage if a user stayed close to the device for extended periods of time. The Kinect2 also uses infrared, but uses time of flight instead of structured light, so there are no dangerous lasers.

3.2.1.4 Intel RealSense - Imaging

Since the Kinect2 can only provide us with 30 fps, we researched another device called the Intel RealSense, which has similar image captures to the Kinect2, but at higher framerates.

Type	Size	fps
RGB	1920x1080	15-30
IR	1280x720	90
Depth	1280x720	90

Table 3: Intel RealSense Specifications

Intel released RealSense in early 2018. Unfortunately, the kit was back-ordered and we never received the device.

3.2.1.5 Chosen Device

We decided to go with the Kinect2 which allowed us to implement our own head detection algorithm and avoided the problem with the user having to use eyewear. Check the hardware section 3.5.5 with regards to the positioning of the Kinect2. The Kinect2 will be referred to as the Kinect in the rest of the paper.

3.2.2 Problems with existing head detection software

As discussed on in the Background section, there were multiple issues with the current state of the art head detection software.

- **RGB/grayscale vs IR**

  All head detection software based on images were either based on RGB or grayscaled images. Our project needs IR data so we can detect the driver's head at night.

- **Time Sensitivity**

  Most existing detection software are not time sensitive. For example, facial recognition is generally run on static images such as Facebook profile images or iPhone unlocking. The closest time sensitive facial recognition we found were Snapchat filters; however, the current filters have some latency in overlaying the image on the face, and the software is closed source.
  Our project requires extremely low latency and high framerate as this affects the 3D illusion feeling when in the car.

- **Face vs Head**

  Most existing detection software has the goal to track a face. Our project requires us to track the driver's head regardless of whether they are facing the camera.

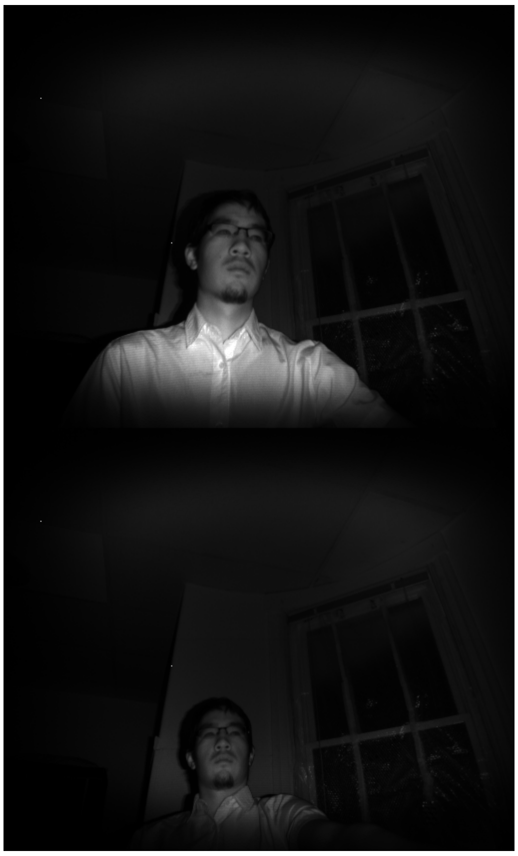
  For these reasons, we decided to create our own head tracking application. We attempted to use two major methods: Haar Cascades and Machine Learning.

3.2.3 Preprocessing IR images

We aimed to make IR images similar to grayscale images to easily build off of existing technologies.

Value Ranges: IR images are represented as floats. We try map IR images to the grayscaled values of 0-255. An example of an IR image being mapped to 0-255 can be seen in Figure 8.

High Exposure: IR images that are created by the Kinect have high exposure, which causes features of a face hard to notice by our programs. The images would look very washed out, either turning out to be very dark or very light. In order to account for this, we created our own autoexposure, which would map the lowest to highest float value in IR to the 0-255 scale. See Figure 7 to see how the autoexposure helps brighten or darken the image when needed.


(a) Image input with no autoexposure

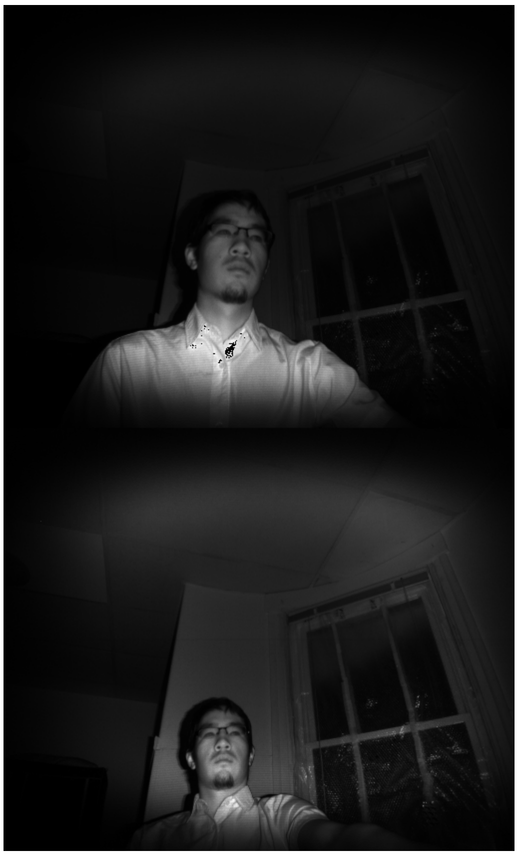
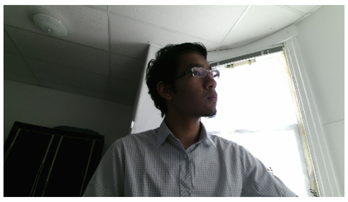
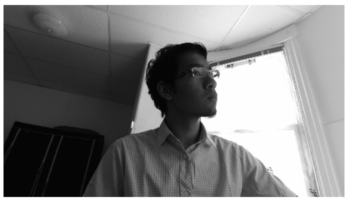

(b) Image input with autoexposure

Figure 7: Comparison before and after using autoexposure program (both versions were run at the same time to take these images)

Smoothing Exposure: With our autoexposure code, we got noticeable flickering due to the quick changes in the highest value of IR. This was prevalent when there was a reflection of IR caused by objects like glare from glasses. In order to account for this, we took the average of a few previous highest IR values, so it would/slowly change the exposure over time.

With our auto exposure system, we can see that the images are much more similar to each other than before. However, there are still some noticeable differences, such as the eyes which tend to blend the iris, pupil, and sclera together. There is also variation in grayscale vs IR image values.


(a) rgb


(b) grayscale

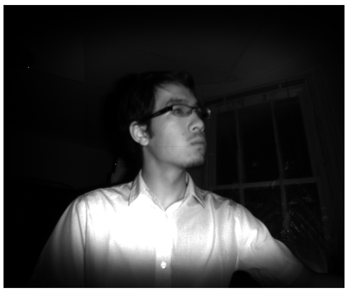

(c) ir

Figure 8: Comparison between rgb, grayscale, and ir images

3.2.4 Method 1: Haar Cascades

Initially, we grayscaled the RGB images from the Kinect camera to run Haar Cascades on in order to get the head location. We then decided to change to IR, so head tracking could be done at both night and day, and we could get a higher and more consistent framerate, as shown in Table 2.

In order to get the location with Haar Cascades, we called the following function implemented by OpenCV, which returned a list of head locations. Each head location was in the form of 4 values representing a box covering the head.

```
boxes = cascade.detectMultiScale(
    numpy_image, scaleFactor = 1.1, minNeighbors=5, minSize=(20, 34),
    flags = cv2.CASCADE_SCALE_IMAGE)
# Taking the most confident head location detected by
# Haar Cascades in the form of a box:
(top_left_x, top_left_y, box_width, box_height) = boxes[0]
```

From these values, we took the center of the box as the head location. This position in the IR image was mapped to the depth image to get the depth value from the Kinect. The x, y position of the head is then mapped to the ranges of -1.0 to 1.0. Finally, the mapped values of x and y, and the depth value are sent to the rendering server where they are changed to a position in a 3D coordinate system.

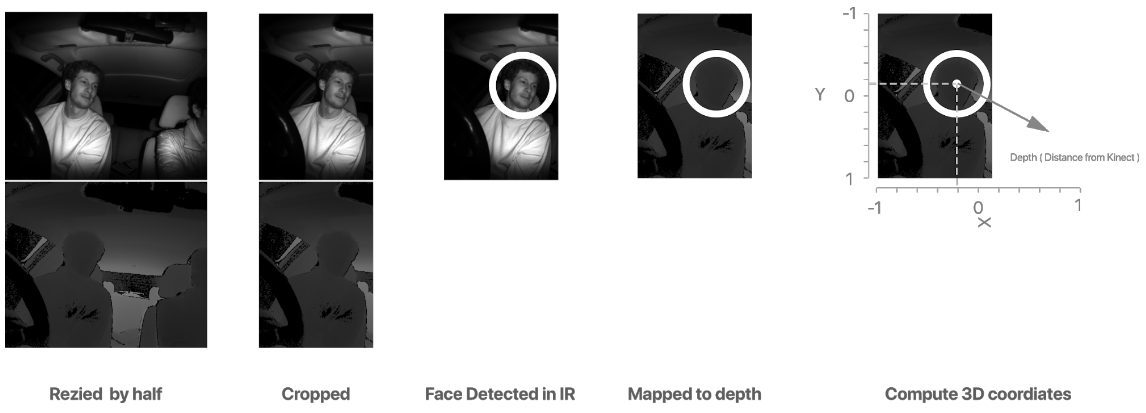


Figure 9: Diagram of Haar Cascades with Optimizations

3.2.4.1 Optimizations

Haar Cascades are pixel based, which means a reduction in pixel count decreases computation, and increasing speed. Because of this the `scaleFactor` parameter effects the accuracy and speed for detecting heads. The lower the value, the more times it scales the image, increasing accuracy, but decreasing speed (and vice-versa for a higher value). We found the best compromise between speed and accuracy to be a `scaleFactor` of 1.1. We also used other methods to increase accuracy and speed as listed below:

- **Scale down the image**

  Scaling down the image by half gave us enough resolution to detect the head, and gave us a performance boost. One potential issue is that if the head is further away, it could no longer be detected. This was a non-issue because the head in the driver's seat would always be within one and a half meters from the Kinect.

- **Crop the image to only include the driver** Since the Kinect is positioned in the middle of the car (reasoning in section 3.5.5), it can occasionally detect the head of the front seat passenger if they lean over to the left. This is problematic, so we crop half of the input image from the Kinect. It also has the added bonus of removing more pixels, increasing the speed.

- **Add more classifiers to increase accuracy**

  It is important for us to be able to detect a head even when it is faced away from the Kinect. Unfortunately, a Haar Cascade which detects a frontal face does not detect a profile or semi-profile face very accurately. We circumvent this issue by applying multiple Haar Cascades which can detect the head, and also one cascade which can detect profile faces. We use these classifiers from the OpenCV repository [11].

```
haarcascade_frontalface_default.xml
haarcascade_frontalface_alt.xml
haarcascade_frontalface_alt2.xml
haarcascade_profileface.xml
```

- **Stop when the first cascade detects a face**

  Since we use multiple Haar Cascades, after the first one detects a face, we do an early exit and don't run the other ones.

- **Use the last successful Haar classifier to try again first (of the four)**

  Since each Haar Cascade is slightly different, we assume that after one Haar classifier has found a head location, that it is most likely classifier to find a head in the next image given by the Kinect. We use a priority queue which moves the previous successful Haar classifier to the top of the queue.

- **Multi-threading attempt**

  Our head tracking software was synchronous which meant that the program would experience a spike in latency if a head couldn't be detected by any of the four Haar Cascades. This spike in latency is caused by the fact that all Haar Cascades must run as we can't make an early exit. To combat this, we created a queue which gets populated at 30 fps by the Kinect and has multiple worker threads which run and try to extract head locations from these images. If a thread is able to detect a head location on a newer image faster than another thread working on an older image, then we invalidate the head location from the older image and delete all previous images in the queue. This allows us to avoid being bogged down by a previous attempt which may take 4 times as long to detect a head. We attempted to use threading which allows us to run our program on multiple incoming images. Each worker thread would take the image, and try to find a head. If a more recent thread found a head in its image, the system would discard any older threads and their images, as they were considered out of date.

  Unfortunately, OpenCV's implementation of Haar Cascades does its own multi threads at the pixel level, which made our multithreading at the image level ineffective, especially when running the HeadTracking Service with the other parts of the project (like the KinectServer and Rendering Service).

3.2.4.2 Issues with OpenCV's Haar Cascade Implementation

- **Lack of confidence values**
  This list which the function `cascade.detectMultiScale` returns, is ordered by confidence, meaning the first box in the list is most likely to be the head. However, the value of the confidence is not given. This was problematic because we would encounter situations where Haar Cascades would return a head location with lower confidence. Since we're not given the confidence value we cannot try to get a more confident head location.

  For example, imagine a scenario where these were the values returned by the cascades:

haarcascade name	list of head locations with confidence values
haarcascade_frontalface_default.xml	None
haarcascade_frontalface_alt.xml	[box1 with 50% confidence]
haarcascade_frontalface_alt2.xml	[box1 with 80% confidence, box2 with 50% confidence]
haarcascade_profileface.xml	None

  With our algorithm, the box1 from frontalface_alt would be chosen, as we would have no idea that the box only had 50% confidence.

- **Doesn't give control on a per pixel basis**

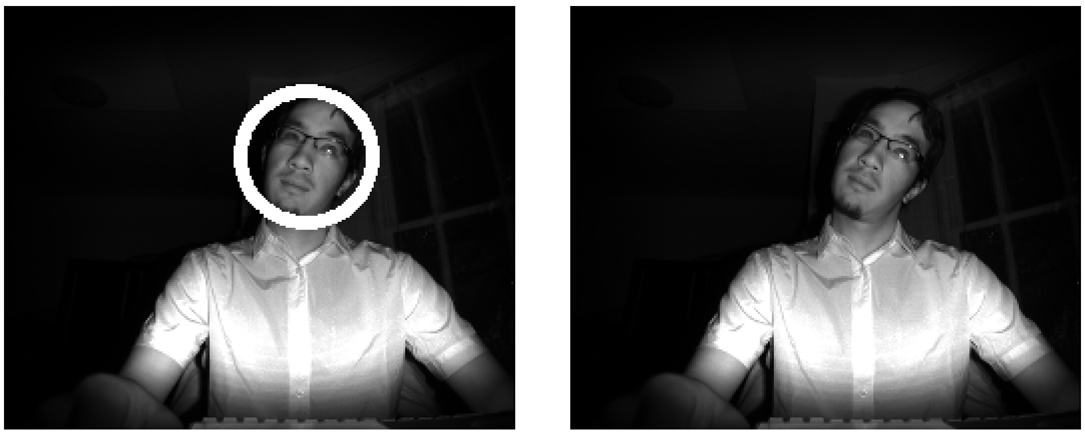
  Haar Cascades run per pixel. However, OpenCV does not allow us to control which pixels the Cascades are run on. This did not allow us to use certain optimizations such as skipping every other pixel or using background removal (which was used in Section 3.2.5.4.1).

- **Does not cache resized images**

  Every time a Haar Cascade is run, it is run on multiple scales - determined by the `scaleFactor` option - of an image. Unfortunately, the same image is rescaled per classifier, and there is no way for OpenCV to cache the scaled images in order to effectively run multiple classifiers.

- **Head Tilt**

  Because Haar Cascades match features by certain locations, it would not be able to detect the head when slightly tilted as shown in figure 10. On top of this, the Kinect is slightly tilted when mounted in the car. This could be accounted for by tilting the image by a certain number of degrees to match the tilt of the head, but this would take too much time and computation power.


(a) Haar Cascades able to detect head at certain degree.

(b) Haar Cascades unable to detect head after a certain tilting degree.

Figure 10: Limit of how much the head can be tilted with Haar Cascades.

3.2.4.3 Result of Haar Cascades

In addition to the host of other problems explained above, the performance to accuracy ratio of Haar Cascades is not sufficient for our use case. Acceptable accuracy results in 50-60 ms per frame of processed data, which is 3 times too slow.

3.2.5 Method 2: Machine Learning

We've tried various convolutional neural network architectures. For all of our Machine Learning we utilized Tensorflow while using a GPGPU to do head tracking. Since they are highly parallelizable and customizable we predicted that this would resolve the issues we faced with the Haar Cascades method. We used Leaky_ReLU for all of the hidden and input layers and Sigmoid for the output layers on all of our architectures.

3.2.5.1 Attempt 1: Densely connected encoder network

- Architecture:
  - Input: 1/4 scaled down Kinect output(s).
  - Learning label: Bitmap mask generated by OpenCV using Haar Cascades.
  - Output: Bitmap mask of possible head location.
- Result:
  - Network overfits to the last training epoch.
  - Network can only get as good as Haar Cascades.

- Network does not have a confidence output, assumes there is a head present every frame.

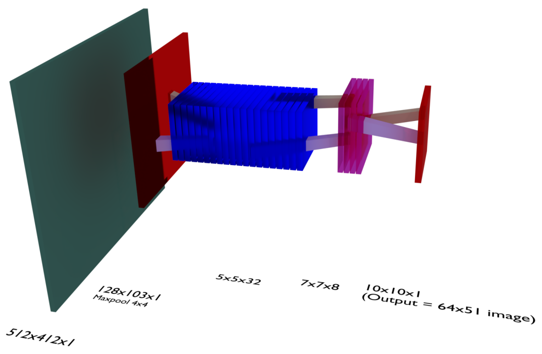


Figure 11: Model 1

For our first attempt, we decided to create an encoder network that would give us a confidence map as seen in Figure 11. The confidence map was 1/8 th of the image (64x51), with values between 0 and 1, where 1 is 100% confidence that the pixel is part of a head, and 0 is 0% confidence.

To create the labels for the training and test sets, we computed a bitmap from OpenCV's Haar Cascades. We generated a black image that was the same size as the output of the network (64x51). Then we found the head location using OpenCV's Haar Cascades, and filled the head location with 1s as shown in Figure 12 and 13.

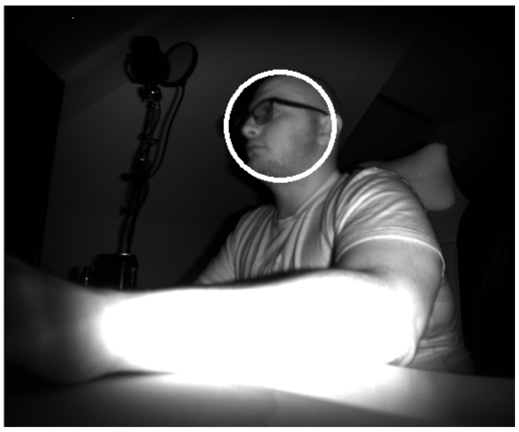


Figure 12: IR image of the training input for model 1 with HAAR cascade detection drawn on face

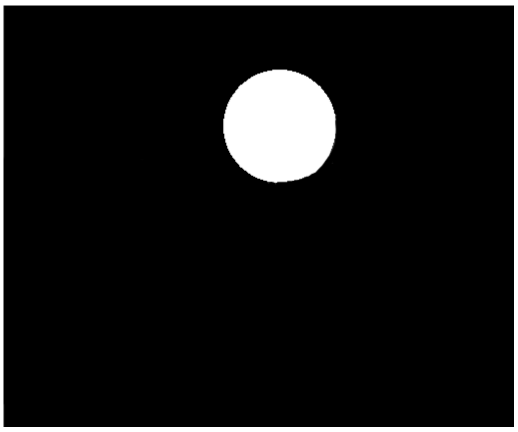


Figure 13: Training label for model 1

We tried a combination of inputs for this network:

- Depth image
- IR image
- IR and Depth image

We observed multiple problems with this architecture. Our biggest problem was that the generated image did not give us accurate information about where the head could be as seen in Figure 14. We wanted to use blob detection from OpenCV, but this method proved to be unstable because the neurons fired too randomly to create an accurate blob.

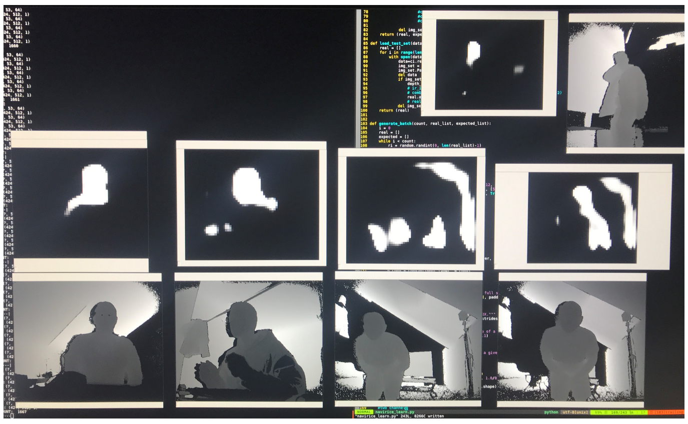


Figure 14: Model 1 output for various images

3.2.5.2 Attempt 2: Regression network

- Architecture:
  - Input: 1/4 scaled down IR image.
  - Learning label: Bitmap mask generated by OpenCV using Haar Cascades.
  - Output: Bitmap mask of possible head location.
- Result:
  - Regression network with a small number of nodes. Is only able to detect the head in a certain location due to lack of nodes.
  - Regression network with too many nodes, processing speed decreased (used up all resources on our device).

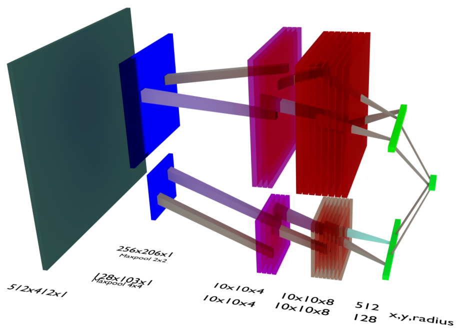


Figure 15: Model 2

3.2.5.2.1 Explanation:

For our second attempt, we decided to integrate two different components based on our first attempt.

- regression
- scaled layers

Since we didn't know how to interpret the output data from our first attempt we decided to add a regression layer to our model. This allowed us to interpret the convolutions directly so that the model could give us x, y, and radius of the face. We accomplished this by swapping out the old output layer from attempt 1 with two flat layers. Once we started training this new model, we realized that the regression data output by our model was affected by the other objects that happened to be on the image. In other words, the model was overfitting and it could only predict cases that it has seen before.

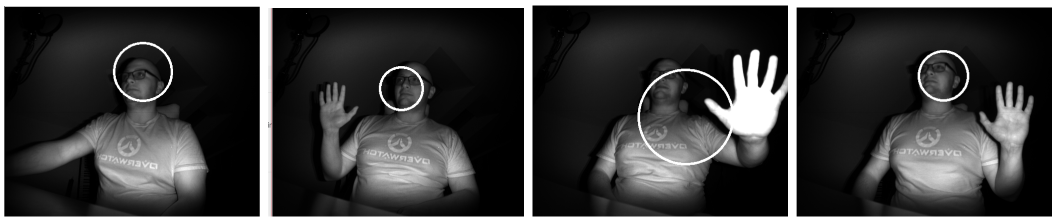

(a) Model trains on the head with no hands (b) Model correctly detects the head with right hand present (c) Model fails to detect head with other hand present (d) Model retrained on head with other hand present

Figure 16: Models reaction to hands

For example, if the last view images the model trained on had the head located at a certain coordinate, it would only detect the head when it was within a certain range of that coordinate. To overcome this issue we added a secondary convolutional layer set that was scaled down to half of the original convolutional set. This method provided the network with more scale invariant data so it helped with the overfitting issue to an extent.

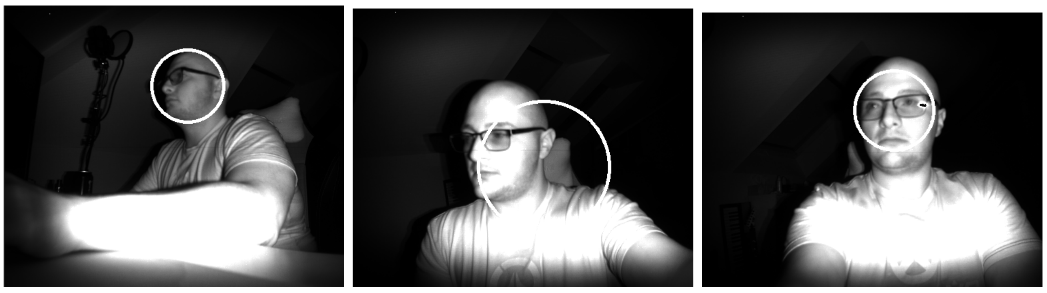

(a) Example input with Haar Cascade label (b) Example output of model with single convolutional path (c) Example output of model with dual convolutional path

Figure 17: Adding dual convolutional path

3.2.5.2.2 Training

To train this model we used Haar Cascades just like our attempt 1. However, instead of generating a bitmap image, we trained directly on Haar Cascade's output. This means that the model could only be as accurate as Haar Cascades.

Training steps:

- Acquire the latest cached image from Kinnect Server.
- Generate the label from the image using OpenCV.
- Feedforward the image to the model.
- Backpropagate using the generated label from OpenCV.

3.2.5.2.3 Challenges

This architecture assumes that there is a face in the given image at all times. This means that the model has no way of filtering common background objects. Since this model depends on Haar Cascades to train, which are not perfect, we couldn't find a better way of representing a confidence value.

With all the adjustments made to this architecture, it performed at a rate of 20ms to 27ms on Nvidia Jetson TX2 over a 1/4th scaled down version of the IR image. This means that the device can process 40 frames per second. We can reduce this to 30 frames per second, as it is the fastest the Kinect can retrieve images. However, even at 30 frames per second the model uses most of the TX2's resources and does not leave enough for other programs to run effectively. We tried scaling down the input image even more, but it caused the model to perform at an unacceptable accuracy.

3.2.5.3 Attempt 3: Cascading Convolutional Neural Networks

- Architecture:
  - Input: 12x12 patches (first pass) 1/8 scaled IR image and 24x24 patches (second pass) of the 1/4 scaled IR image
  - Learning label: Probability of head vs Probability of not head
  - Output: Patches of potential head locations
- Result:
  - Able to run at 10 ms with a 1/4 scaled input image
  - It occasionally detects both the head and the body as head, most likely due to WIDER dataset not representative of realistic data (RGB instead of IR, random faces instead of faces in the car).

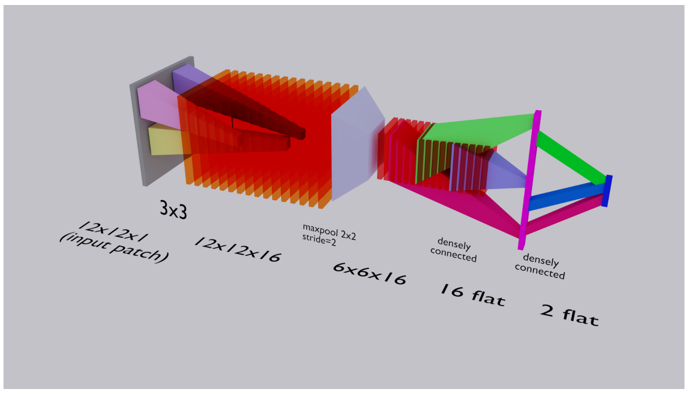


Figure 18: First stage of the Cascading Convolutional network, Model 3

3.2.5.3.1 Explanation:

Densely scanning the whole image for a head was too slow. We split the image into multiple patches, and eliminate patches where the head is unlikely to be to decrease computation. 18

For our third attempt, we got inspiration from Li et al. [12]. They used Cascading Convolutional networks to scan the image and quickly eliminate areas with no interest so they can speed up the process. Since we use the IR image to track the driver's head we implemented the 12x12 net and 24x24 net mentioned in Li et al. (2015) with only one channel and other slight modifications.

3.2.5.3.2 Cascading Convolutional Neural Networks

When the Machine Learning task is to track a given area in an image, Convolutional Neural Networks (Conv Nets) is not able to perform at desired speeds, as shown in the past two attempts. This is because regular Conv Nets use the entirety of the image to draw conclusions, which means that the image is densely scanned and all the Conv Net filters are densely scanned as well. To prevent the unnecessary computation, Cascading Conv Nets are introduced.

Cascading Conv Nets eliminate the unlikely regions of the image and stop computation at those locations. All the regions that pass the first stage of cascading are then sent to a more complex model to be evaluated. Here are the steps that we take to process the image with Cascading Conv Nets:

- 512x424 IR image is separated into patches of 48x48 with a stride of 16. (total of 128x106 48x48 patches).
- All of the 48x48 patches are scaled down to 12x12 using bicubic filtering.
- All of the 12x12 patches are evaluated by our first stage Conv Net 18.
- If the first stage Conv Net is confident that there is a head in a given patch, the patch location is sent to the second stage.
- The passing 48x48 patches are scaled down to 24x24.
- If the second stage is confident that there is a head inside the image, then the patch location is sent to an algorithm that groups the patches together to decide the final locations. The bounding boxes of the heads are also computed here.

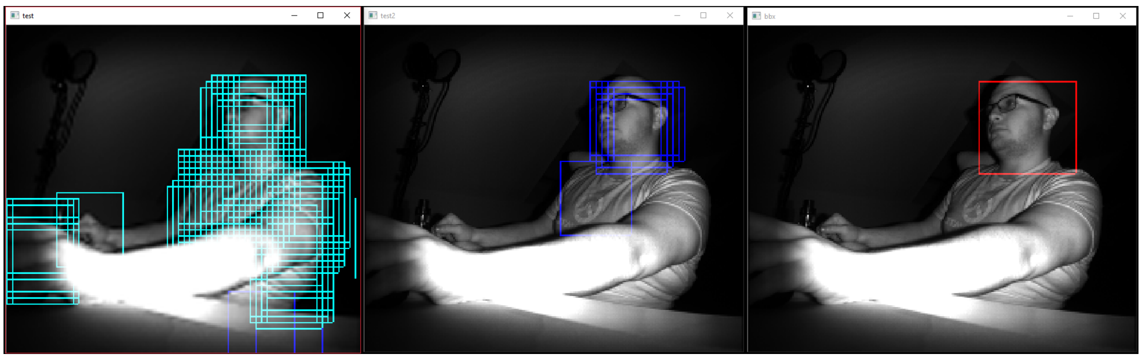


Figure 19: Cascading Neural Network outputs
left image: first stage passes
middle image: second stage passes
right image: final head location

3.2.5.3.3 Training with WIDER dataset:

WIDER dataset contains 32,203 images with 393,703 labeled faces. All the faces in the WIDER dataset are labeled as bounding boxes. In order to generate 12x12 and 24x24 patches, we load a random image from the dataset and crop the face area by the maximum of the width and height. The cropped images are then scaled down to the corresponding patch sizes.

Second pass network is identical to the first pass network (Figure 18) except for the number of filters and the number of neurons in the flat layer.

- convolutional layer 1: 32 filters
- convolutional layer 2: 64 filters
- flat layer: 256 neurons

3.2.5.4 Optimizations

3.2.5.4.1 Background detection

One method of optimization we experimented with was background detection. Because the Kinect was securely mounted to the inside of the car, we could consider the car interior itself to be "background" and fixed. The only object that changes in the perspective of the Kinect is the driver's body. Any object in front of the "background" could then be detected by the Kinect, as we have access to the depth image and could compare against a reference scan.

When the Kinect is first initialized, it takes a reference depth snapshot without the driver sitting in the driver's seat. As soon as the driver gets in the seat, the depth values for those pixels will be less than the reference, and the driver can easily be distinguished from the background. This optimizes both the speed and accuracy because the background can be ignored in the computation.

Initially, we used NumPy to do this on a per-pixel basis. Later, however, we developed a system inside the KinectServer itself in C++ to do it on a per Patch Basis. This was beneficial, as we were planning on using it to feed directly into the Patch based machine learning system to provide an early-out for what could not possibly be a head. This implementation was also significantly faster in performance, as it was written in C++ and did not need to check every single depth pixel in the 512x424 image.

3.3 Phone Application

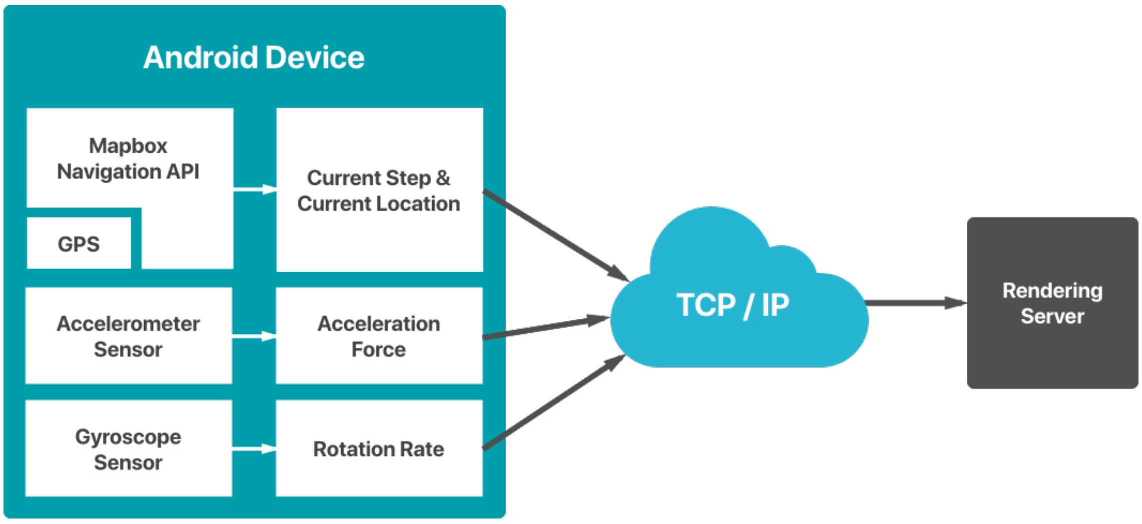
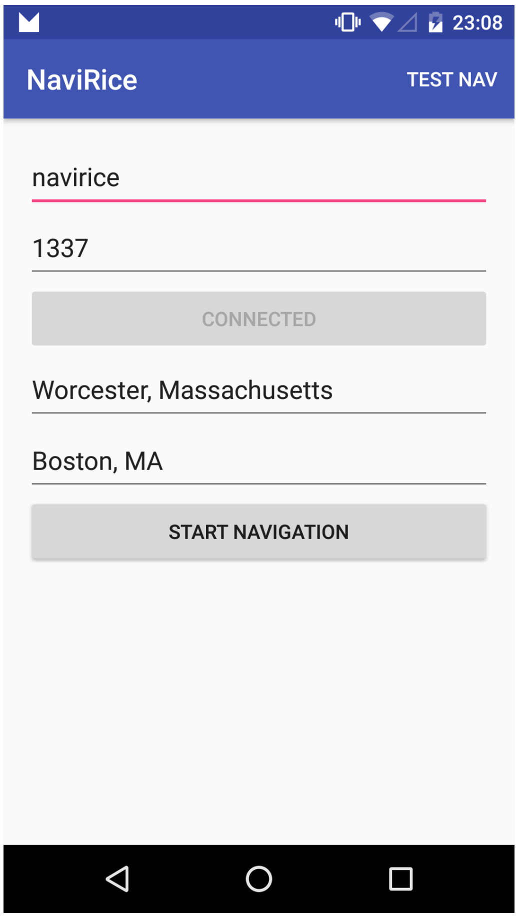


Figure 20: Phone App overview

We created an Android App to stream the upcoming step, the acceleration force and the rotation rate to the rendering server.



navirice

1337

CONNECTED

Worcester, Massachusetts

Boston, MA

START NAVIGATION

Figure 21: App launch screen

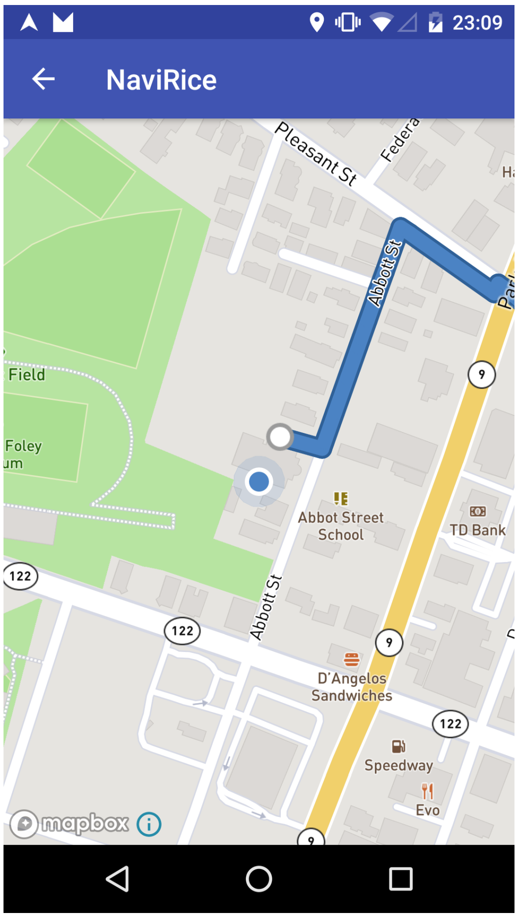
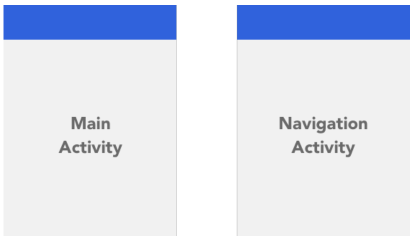


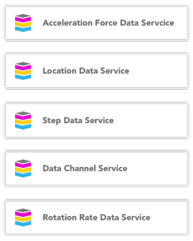
Figure 22: App navigation screen

The App determines the source location by collecting current latitude and longitude from the Global Positioning System (GPS) sensor. It then prompts the driver to provide the name of their destination and resolves it into GPS coordinates through Google Play Geocoding API. The coordinates are then passed to online map APIs to calculate the possible navigation paths between them. The App picks the shortest path and starts computing milestones, navigation progress, and the upcoming steps after the user presses the *START NAVIGATION* button. The APIs also snap the GPS coordinates of the current location to the closest street and gradually moves to the next step each time the car reaches the end of the current step. The navigation activity synchronizes the map with navigation service and shows the current route on the screen. The navigation service also notifies the current step data service and sends the directions and instructions to the rendering server.

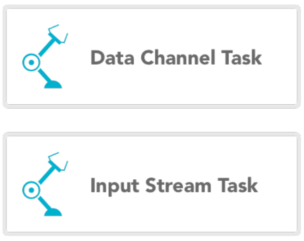
3.3.1 App Architecture



(a) Activities



(b) Data services



(c) Asynchronous Tasks

The App has two activities, 6 core services, 6 data services and 2 asynchronous tasks. The activities are responsible for displaying connection status, highlighting the navigation route, and handling user interactions. The core services are responsible for getting sensor data and performing navigation. The asynchronous tasks are mainly used for managing I/O streams in the background thread.

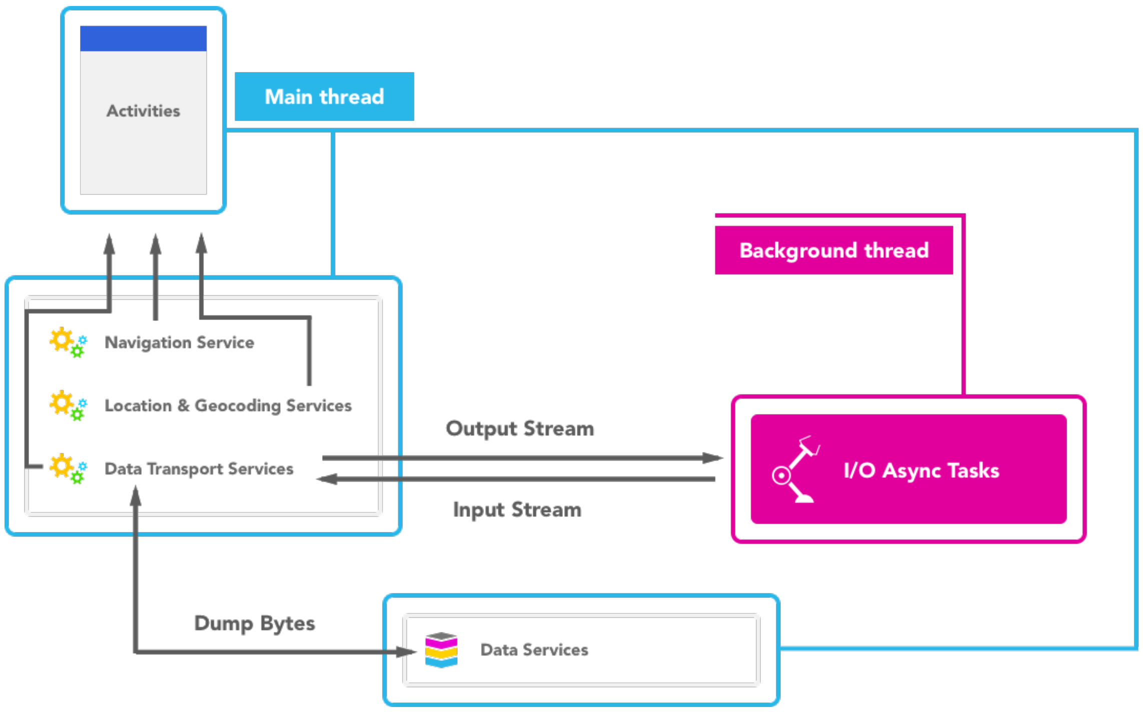


Figure 24: Phone App thread model and data flow

We used a *TCP* server and client architecture to transfer data between the Android App and the rendering server in real-time. An easy way to implement the TCP client is to block and do *I/O* operations on the current thread. However, since Android does not allow performing any blocking operation on the main thread (to avoid freezing the user interface (*UI*)), we created a special background thread for the client to communicate with the rendering server.

Android also has its own unique thread safety model, which prohibits other threads from interacting with the UI directly. To solve this problem, we developed two special asynchronous task managers to switch between UI and network thread when updating UI elements.

Different from desktop applications, an Android App may be killed after a long period of inactivity or be paused when using other Apps. Unfortunately, Android offered a special feature, called Service, to keep background services running, such as playing music. We encapsulated the TCP client into a background data Service to transfer data between the App and the rendering server in the background.

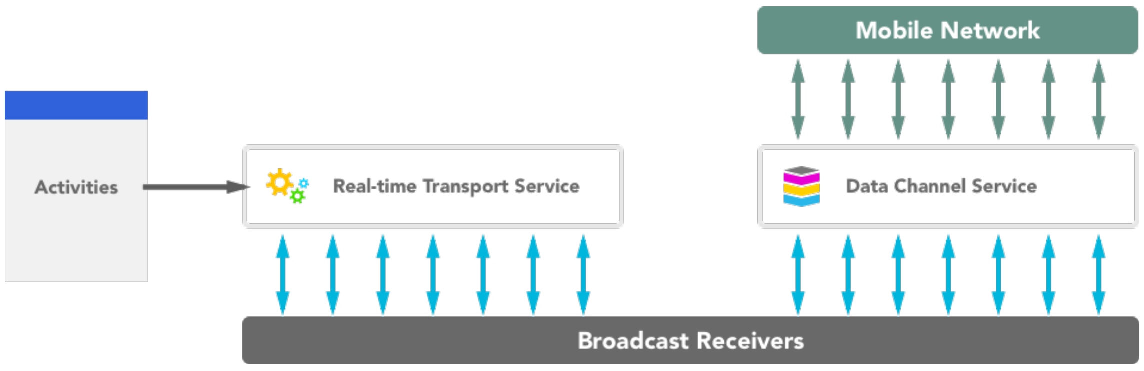
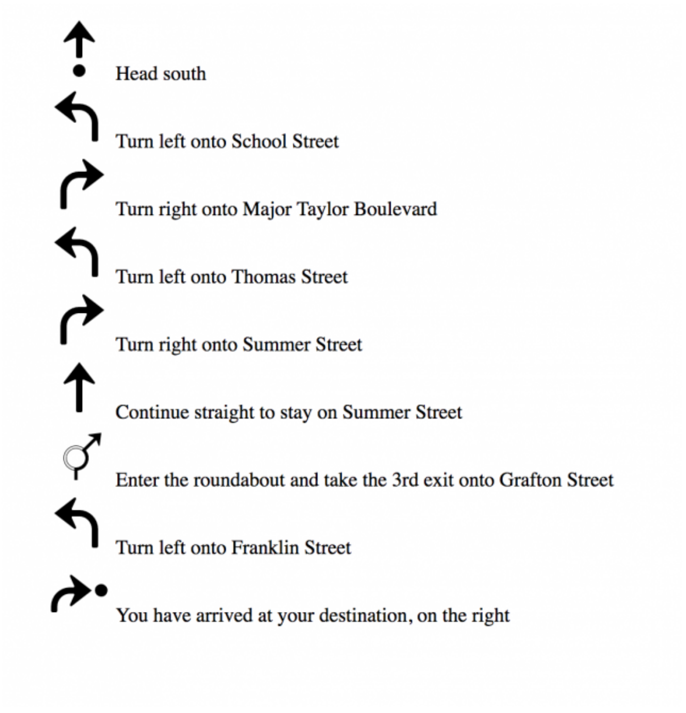


Figure 25: Data transport services and Broadcast Receivers

By default, Android only binds a service to a single activity to pass data between them. To allow both of the main activity and the navigation activity to interact with the client, we also developed a collection of *BroadcastReceivers* to send messages to the App. To send the upcoming step to the rendering server, the navigation activity sends the bytes representing the step to Real-time Transportation Service. The service then broadcast the data within the scope of the App. The Broadcast receiver in the service then receives the broadcasted data and passes it to the TCP client.

3.3.1.1 Getting Navigation Data

We experimented with the Google Map API to compute the current navigation step from the current location. It has very good documentation and has a larger developer community compared to other APIs. To determine the current step, we gather all of the waypoints and use the car's GPS coordinates as well as the previous step to determine the driver's progress along the path. However, due to the complexity of the curvy path and automatic rerouting algorithms, we eventually decided to move forward with the Mapbox Navigation SDK instead. This SDK provides us with both the current step and the navigation icon names. The Mapbox Navigation SDK also allows faster navigation updates without payment, unlike Google's API.



Head south

Turn left onto School Street

Turn right onto Major Taylor Boulevard

Turn left onto Thomas Street

Turn right onto Summer Street

Continue straight to stay on Summer Street

Enter the roundabout and take the 3rd exit onto Grafton Street

Turn left onto Franklin Street

You have arrived at your destination, on the right

Figure 26: Direction icons complimenting MapBox Navigation SDK

3.3.1.2 Getting Acceleration Force and Rotation Rate

The GPS data is good for estimating the coarse location of the car. However, it does not update frequently enough to accurately determine the car's instantaneous location because it takes time for the GPS sensor to receive signals broadcasted by the satellites. Within the 200 milliseconds waiting for the signal, a car running at 40 mph can potentially move to another place 3 meters away from the estimated location. On top of the GPS sensor data, the mobile App also collects the current acceleration force and rotation rate in the x, y, and z axes to better predict the instantaneous location of the car.

The Android sensor manager provides a unified way of collecting sensor data. The acceleration sensor measures the acceleration applied to the device, including horizontal acceleration and acceleration due to gravity. The gyroscope sensor measures the rotation rate around the devices x, y and z axis. Both of them represent data in form of three floats.

Sensor Name	Data	Description	Unit
Acceleration Sensor	value[0]	Acceleration Force along x axis	$\mathrm{m\,s^{-2}}$
	value[1]	Acceleration Force along y axis	
	value[2]	Acceleration Force along z axis	
Gyroscope Sensor	value[0]	Rotation Rate along x axis	$\mathrm{rad\,s^{-1}}$
	value[1]	Rotation Rate along y axis	
	value[2]	Rotation Rate along z axis	

3.4 Rendering

The service to create the visuals actually displayed on the screen is the Renderer. It uses a virtual 3D coordinate system, where the location of the driver's head, Kinect, windshield, and monitor are all accurately mapped. To accurately display the overlaying image of navigation information in the real world we need to undo the warping caused by the reflection, which is affected by the curvature of the windshield. From the 3D objects in these coordinates, we work backwards to see how these objects map from the virtual world to the windshield, and then finally to the screen.

3.4.1 Windshield measurements

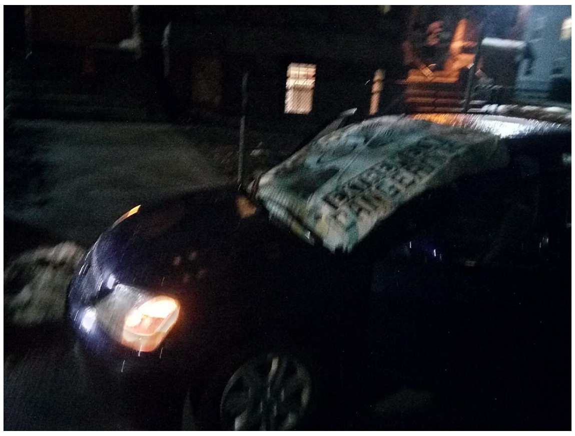


Figure 27: Scanning the windshield of the car

In order to accurately undo the warping caused by the windshield, we need an accurate model of the curvature of the windshield. Without it, the image would still be warped and incorrectly projected, and it would ruin the holographic illusion. To get an accurate model of the windshield, we use the Kinect to scan the surface. Because the Kinect uses time of flight IR to get a depth image, it can not correctly measure transparent or reflective surfaces. To actually get a scan of the shape of the windshield itself, even though the windshield is transparent, we used a slightly damp blanket to cover the windshield. This forced the Kinect to measure the depth on the surface of the blanket, a few millimeters above the windshield, instead of the interior of the car. We then corrected for this slight displacement in CAD software. The dampness of the blanket helped it stick closely to the surface of the windshield, and helped improve the accuracy of the scan.

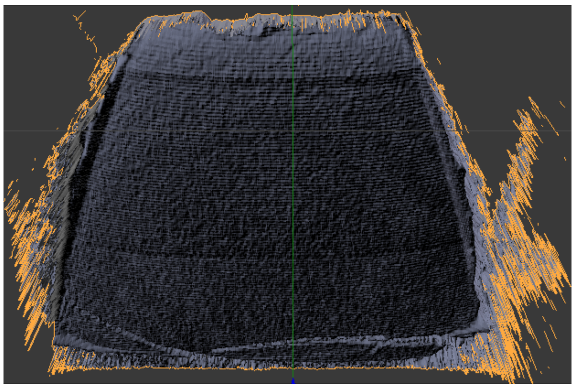


Figure 28: Scanned windshield model in blender

After gathering the scanned data, we reversed the perspective caused by the Kinect to generate an accurate mesh.

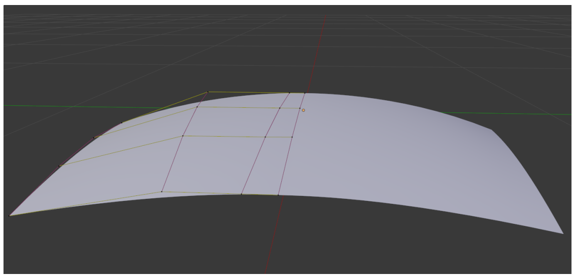


Figure 29: A NURBS surface of the windshield

We then used 3D CAD tools to create a curved surface based on the scans. The curves are made using NURBS surfaces, which are analytic curves that are later converted to a mesh.

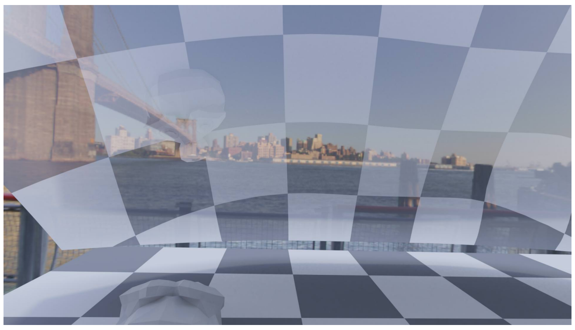


Figure 30: A preview of what the windshield's reflection would look like

Due to the precision and lack of calibration of the Kinect, we are unsure of whether the model of the windshield is actually correct. A slight imperfection in the curve can lead to a very large difference in the way the image is warped, so having very accurate models is very important. This has lots of room for further research. Note the slight irregularities in the reflections on the right side of the in Figure 30. This is likely caused by a inconsistency of only a millimeter or two in the virtual windshield.

3.4.2 Optimization

Optimization was very important for the rendering server, as it needs to update at 90 Hz in order to not have a jarring effect on the user. The original and naive method was to use millions of rays, reflecting them off of the windshield's mesh, and then find where they fell on the Virtual display. Each ray would carry a color, and the pixel on the virtual display that the ray finally ended on would assume the color of the ray. This method, while accurate, would be very slow, as the number of rays needed to be cast in order to completely fill the virtual display would be in the millions. Even still, many of the rays that were cast towards the virtual windshield would reflect in such a way that they would not hit the virtual display.

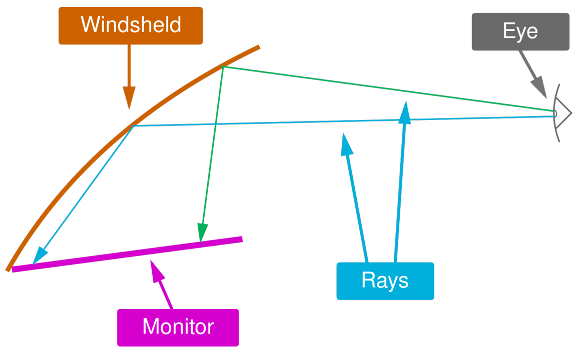


Figure 31: Naive method of tracing rays

The speed at which we estimated we could cast rays on our target hardware was approximately 10 million per second. This would be far too slow, as we would only get a few frames per second. Even state of the art raytracers would not be able to perform at usable speeds on our chosen hardware. "Realtime raytracing" is still a very young field, and is seen in the graphics community as the holy grail of realtime computer graphics.

The first advancement came as a replacement of raytracing a mesh. Instead of raytracing a mesh for the windshield, raymarch into a displacement texture. The displacement texture would have the curve of the windshield in it, and doing a check whether the ray intersected with the virtual windshield at that point was as simple as finding where the ray entered and exited a volume bounding the virtual windshield, and then checking every single pixel of the displacement texture that the ray passed over. If the ray's z component in the bounding volume was less than that of the texture at that texture coordinate, it would be considered touching the windshield at that step. The 3D position and normal could then be quickly computed. This could be further improved by techniques such as Relaxed Cone Stepping [13].
We ended up ultimately not using this technique, as we found other methods to replace it.

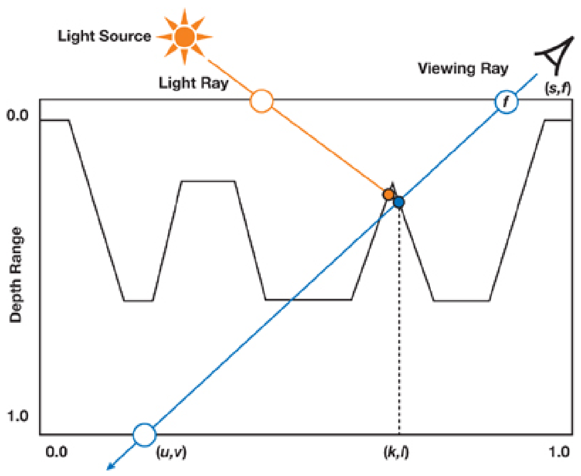


Figure 32: Raymarching through a displacement texture, from [13]

The second advancement was the "tracegrid". Instead of requiring a ray for every pixel of the target display, we realized that we could cast a grid of points, locate where they landed on the target display, and then linearly interpolate the texture coordinates of the viewed image. This was further simplified by the ability to use standard texture mapped triangles to render the warped

tracegrid out onto the screen. Instead of having to trace millions of rays, we only had to trace one ray for every point of the tracegrid. We could vary the accuracy of the reflected image by varying the resolution of the tracegrid, and we decided that a resolution greater than 100x100 had no noticeable decrease in quality. With this change, we lowered the number of rays we were tracing from a few million down to 10 thousand.
We used Evan Wallace's realtime water caustics demo as an inspiration [14].

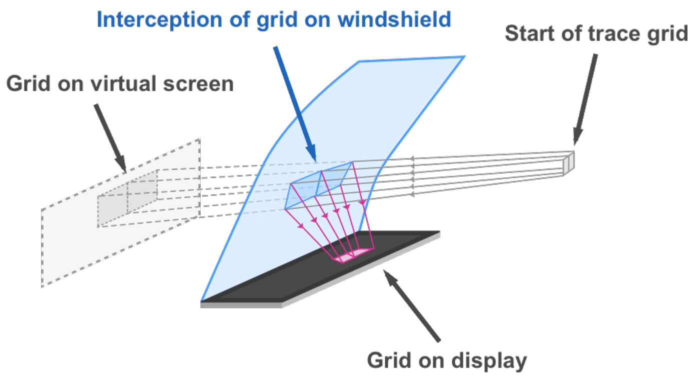


Figure 33: The tracegrid process

The next big improvement in the performance and complexity of the rendering system was the realization that the first rays could be entirely removed and replaced with a rasterization pass. Because the tracegrid was a rectangular grid, it's geometry was such that each ray's collision with the windshield could be calculated by rasterizing the windshield into a set of GBuffers (textures which store the depth, position, and normal for each pixel). Rasterizing a mesh is extremely fast on a consumer GPU, compared to raytracing, and the hardware is directly designed to be very performant at it. This change completely removed the need to do any raymarching, as the second bounce was against a flat surface (the monitor) instead of something defined by a displacement map.

To calculate the collision location of each ray, all that needs to be done is to sample the GBuffers at the ray's corresponding position in the tracegrid. Tracegrid sections that don't overlay the windshield are ignored. A form of anti-aliasing can be done at this step by using Bilinear or Trilinear filtering on the GBuffers. However, GBuffer normals will need to be re-normalized at this step if this filtering is done.

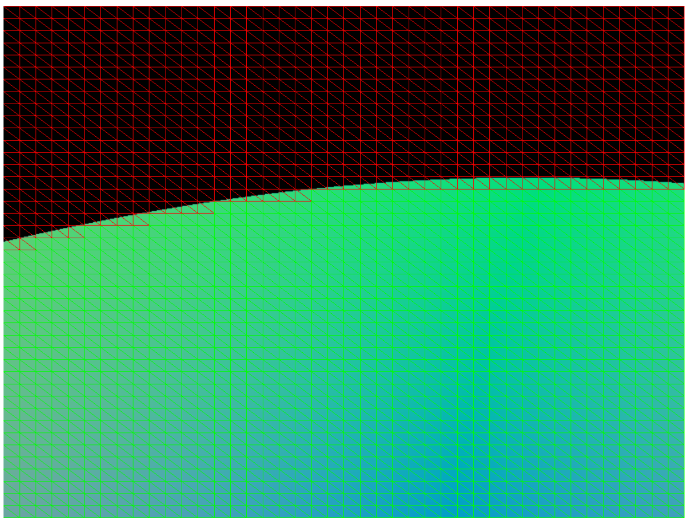


Figure 34: The tracegrid overlayed onto the position GBuffer. Ignored sections marked in red.

In our actual implementation, we rasterize the GBuffers at a resolution of 1024 by 1024 pixels, no matter the size of the tracegrid. A possible future optimization would be to adjust the resolution of these GBuffers to be 2 times the resolution of the tracegrid along each axis, and using Bilinear filtering for best results.

This improvement was inspired by the "Five Faces" realtime raytracing demo by the Fairlight demo group, which achieved realtime reflection and refraction in a fully dynamic scene [15].

The final major optimization that was done to the Renderer was implementing a dynamically adjusting viewport. Because only the objects being rendered have any effect on the actual pixels of the resulting image, the viewport can be shrunk to tightly fit the objects. Also, objects that the user would not be able to see through the windshield can be ignored, and the viewport can be clipped to the bounds of the windshield.

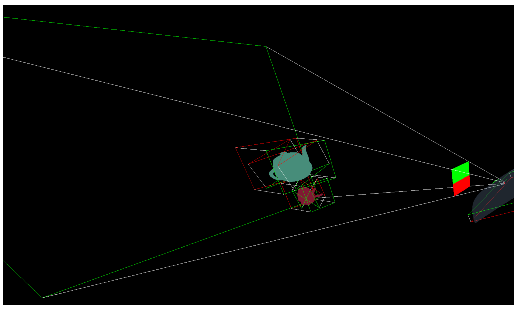


Figure 35: Debug view without dynamic viewport resizing

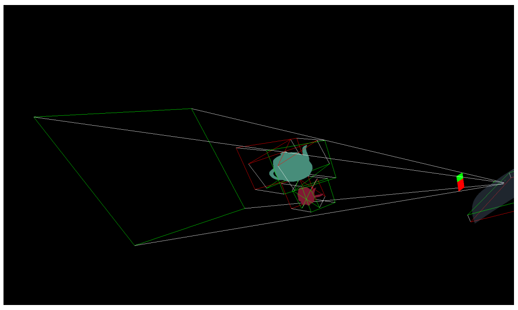


Figure 36: Debug view with dynamic viewport resizing

We accomplished this by computing bounding boxes for each of the objects being rendered, and then finding the maximum and minimum roll and yaw angle from the view for these. The viewport's zoom, aspect ratio, roll, and yaw would be adjusted to tightly fit these bounds. Because this was only adjusting the orientation of the viewport and not the position, it had no visible difference to the user and did not affect the holographic illusion in any way.

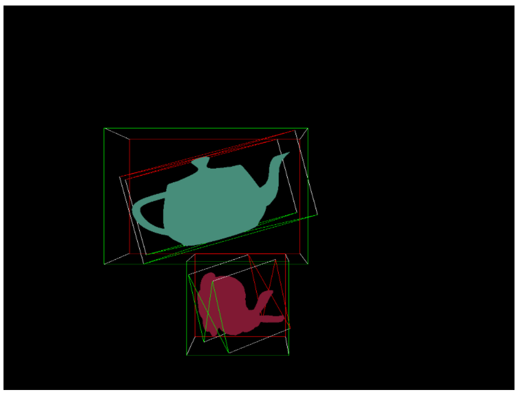


Figure 37: User's view without dynamic viewport resizing (bounding boxes shown)

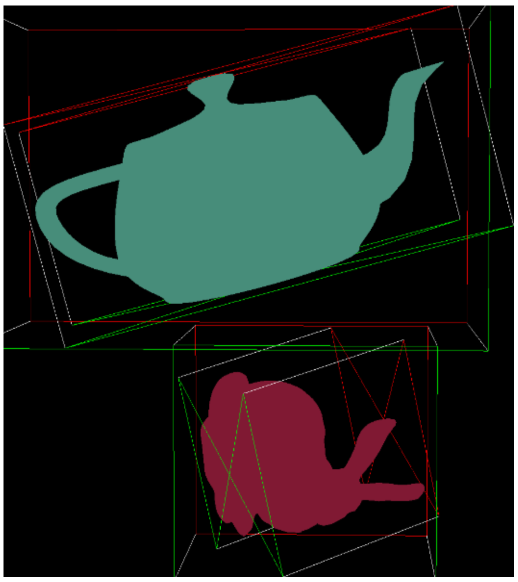


Figure 38: User's view with dynamic viewport resizing (bounding boxes shown)

This technique not only reduces the total pixels being rendered from the final tracegrid pass on the monitor but also has the added benefit of greatly improving the tracegrid's effective precision. Now, a single object in the view will have roughly the entire tracegrid across its surface, instead of just a small section of it. This further allowed us to lower the tracegrid and GBuffer resolution to further improve performance.

Further improvement can be made to this, such as dynamically clipping and creating viewports for each object individually, instead of a single viewport that includes all of the visible objects. The tracegrid could also have this same multi-object adjustment done to it.

With all of these optimization techniques, the Renderer's performance improved from less than 1 frame per second to over 800 frames per second on the Nvidia Jetson TX2. This vast improvement not only allowed the rendered objects to appear smooth and realtime but also allowed for a performance surplus that could be used for making the Head Tracking more accurate.

3.4.3 Head position latency compensation

To compensate for the relatively low update rate of the Kinect, the renderer uses a prediction method to guess where the head will be faster than 30hz. It does this by storing the last two received positions and the time it received them. When drawing each frame, it uses these positions to calculate the velocity that the head is moving at and uses that velocity to predict where the head will be, offset to the last position it received. We found that this works fairly well, as long as the head tracking data does not have a substantial amount noise to it.

3.5 Hardware

The hardware added to the car consists of 4 parts: A power system, for supplying stable power to all of the electronics, a Monitor, for projecting the hologram, a Kinect, for tracking the driver's head, and a computer. We selected the Nvidia Jetson TX2 for the computer. We also experimented with using a projector instead of a flat display.

3.5.1 Car

The car we decided to use as our platform was a 2003 model Honda Civic EX. We chose this car because of its reliability, ease of modification, and low cost. It was also a car that we had experience with modifying before. The entire dashboard had to be removed to fit the Monitor and Kinect equipment. The airbags were left in place, for safety, but most of the plastic trim around the driver's side had to be modified or removed. Additionally, some modifications to the engine bay had to be done for the power system.

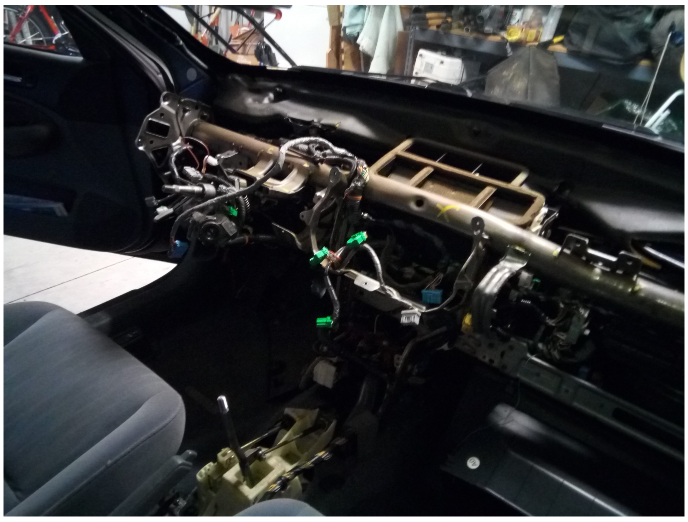


Figure 39: Dashboard removed from car

One future issue with this project could be the legality of the car in its current form, as we doubt it would pass inspection in some states. Custom made dashboards would hopefully mitigate this issue.

3.5.2 Power system

We designed a custom power system to power all of the added electronics, instead of using an off the shelf 120 volt inverter or accessory adapters. Off the shelf equipment would not work due to either the inefficiency of the power conversion, or the power draw through the car accessory jack. All of the electronics added to the car needed specific voltages as well, which would have been difficult to organize with a simple off the shelf solution. The custom power system resulted in a simple, robust, and flexible solution.

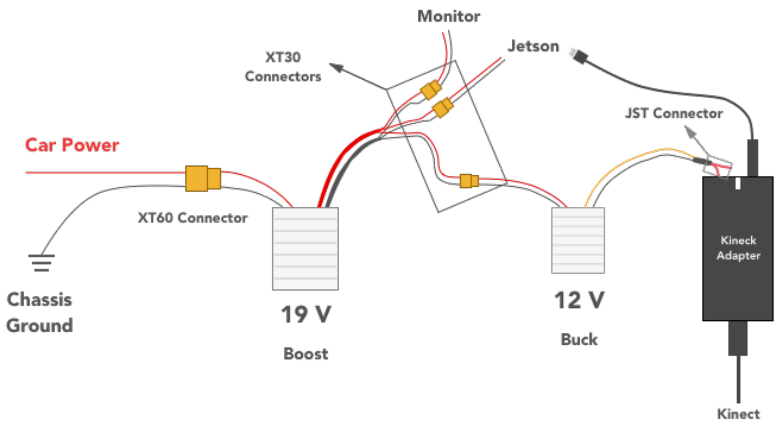


Figure 40: Diagram of the power system

The accessory jack in our project car is only rated for a maximum of 100 watts, or approximately 8 amps at 12 volts. We calculated that the total power draw of our system would exceed this, so using the accessory jack was not an option. Furthermore, most of the fuses and wiring under the dashboard of the car was only rated for 10 amps. We did not want to risk creating electrical issues in the car by tapping into the power for preexisting equipment and potentially creating a safety issue by blowing a fuse that protected vital safety equipment, such as the airbags.

To fix the issue of running too much power through the accessory jack, we decided to run our own power line directly from the battery. This technique is often used for installing aftermarket audio systems in cars. Since our project car was a Honda Civic, there was a wide variety of documentation on how to do this, as the Civic is a popular model to install aftermarket modifications on.

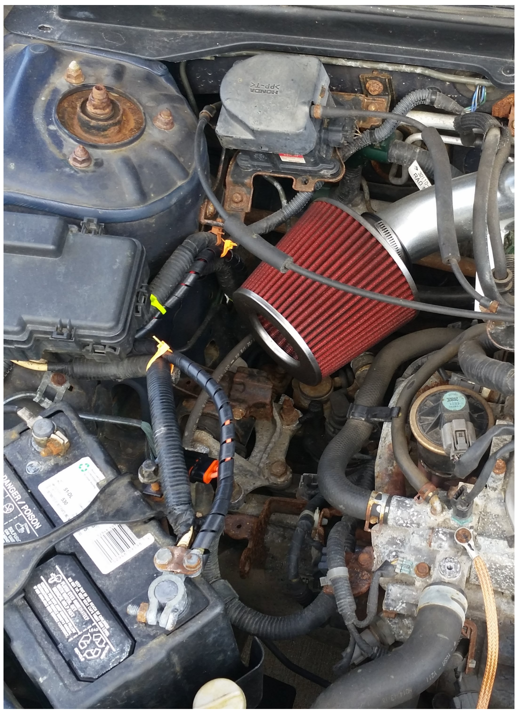


Figure 41: Inside the engine bay

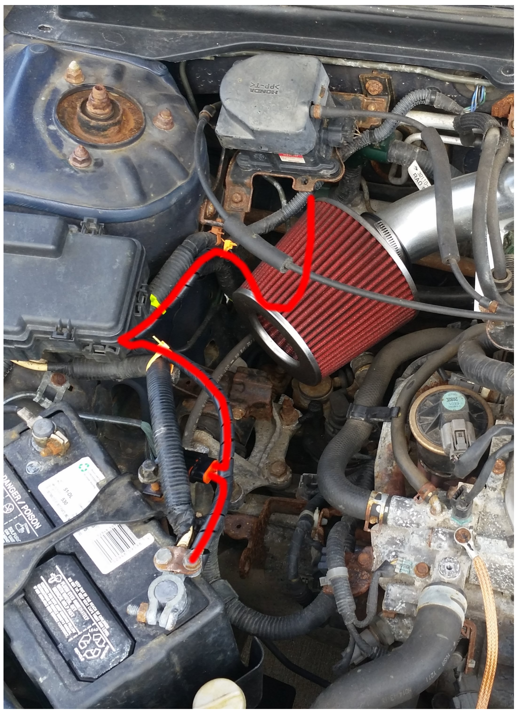


Figure 42: Highlighted power wire

The 10 gauge power wire connects to the car's battery using a crimped ring connector on the positive battery terminal. Then, it goes through a waterproof fuse holder. We used a 25 amp fuse for safety. If the wire melted in the heat of the engine bay and shorted to the chassis, it would blow the fuse instead of starting a fire. The fuse also served a purpose of being an easy way to disconnect power to the whole system when it was not in use. Next, the wire continues down the engine bay to the passenger side firewall wiring harness grommet. We used a clothes hangar to poke a hole and feed the wire through this rubber grommet, as suggested by many automotive audio enthusiasts online [16]. The wire was soldered to an XT60 connector, as well as a chassis ground under the dashboard. Finally, the entire portion of the wire, fuse holder included, in the engine bay was wrapped in a protective covering.

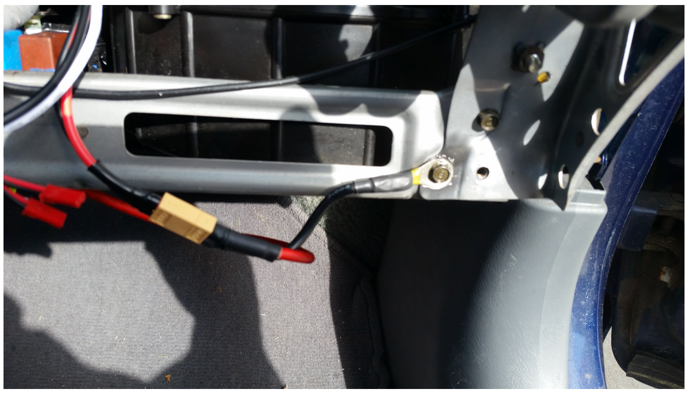


Figure 43: Main XT60 power connector, including chassis ground

System	Required input voltage	current draw
Jetson TX2	19V	4.75a
Monitor	19V	3.42a
Laptop	19V	2.25a
Kinect	12V	2.65a

Table 4: Power usage of the various electronics equipment

The rest of the power system includes a boost converter, for converting the car's 11-14 volts from the battery to 19, and then a buck converter, for converting the 19 volts back down to 12. The Kinect requires 12 volts, and we did not want to run the car's unstable voltage into it. This was a simple and cost effective way to accomplish this.

The Monitor and the Jetson TX2 require 19V for power, and they are wired to the 19V boost converter. We did this by splicing in XT30 connectors into their power cords, so they could also be used inside the car.

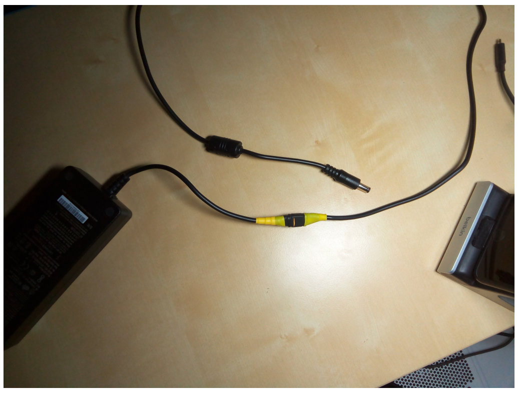


Figure 44: Jetson power cord with spliced in XT30 connector

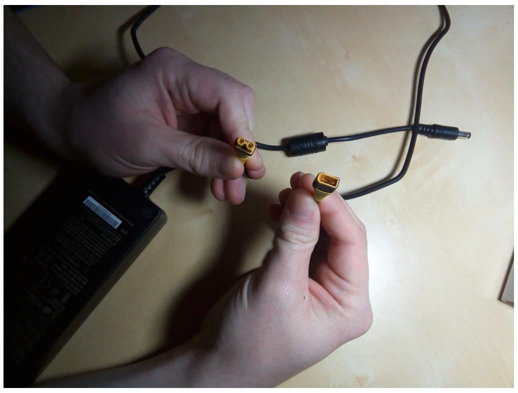


Figure 45: Closer look at the spliced in XT30 connector

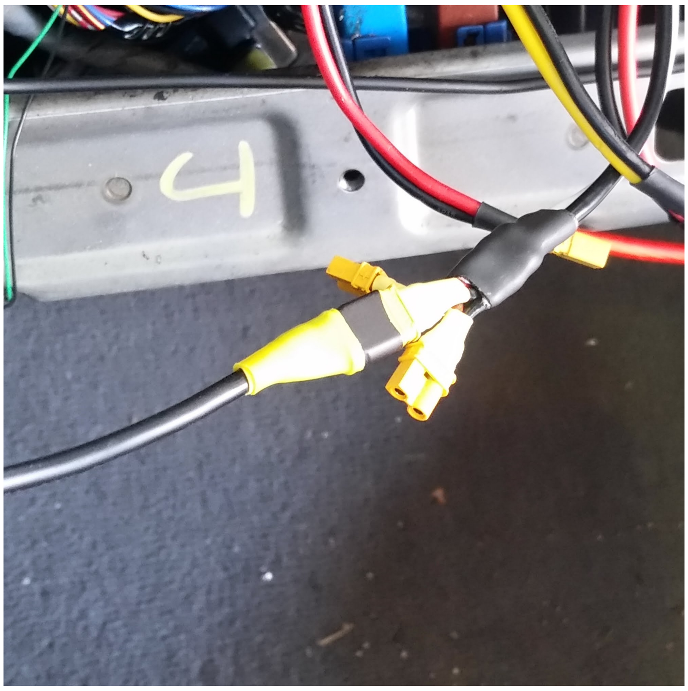


Figure 46: Jetson connected to car power system

Source	Connector	Sink
Car unregulated	XT60	19V Boost
19V Boost	XT30	12V Buck
19V Boost	XT30	Monitor
19V Boost	XT30	Jetson
19V Boost	XT30	Laptop
12V Buck	JST	Kinect adapter brick

Table 5: Connectors used

All of the 19V connections use XT30 connectors (rated up to 30 amps). All of the regulated 12V connections use JST connectors (rated up to 10 amps). Because of the high current load, the main battery connector uses an XT60 connector. An XT30 would likely have been able to handle the current, but for a large safety margin, and to help differentiate and organize the parts of the system, we decided that it would be best to use an XT60 instead.

We also added extra XT30 and JST connectors to the outputs of the power system, in case we wanted to add additional electronics after the power system was completed. We ended up using this feature to power a laptop with 19 volts for debugging.

For simplicity, and to avoid dangerous shorts, we used the female connectors for power sources, and male connectors for power sinks.

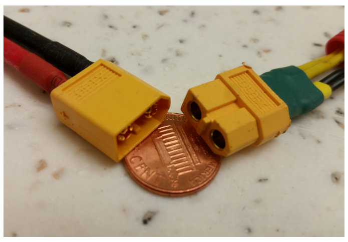


Figure 47: A pair of XT60 connectors

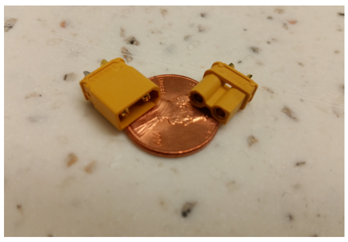


Figure 48: A pair of XT30 connectors

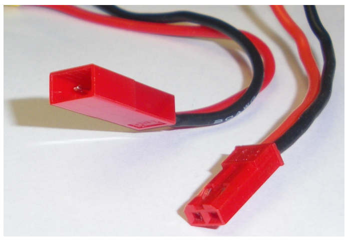


Figure 49: A pair of JST connectors [17]

Power source	Maximum current sink	Estimated current sink	Current source
Car unregulated	25 amps	16 amps	Car's alternator/battery
19V Boost	15 amps	12 amps	Car unregulated
12V Buck	10 amps	2.65 amps	19V Boost

Table 6: Power usage of the various electronics equipment

The boost converter is a 19V unit rated at 15 amps. The buck converter is a 12V unit rated at 10. Both have over-current and over-heat shutdown protection. We calculated the amount of power all of the electronics would need, then added a bit for a safety margin when deciding what converters to use. The Monitor's power brick was rated at 3.42 amps, and output 19V. The Jetson's power brick was rated at 4.75 amps and output 19V. The Kinect's was rated at 2.65 amps and output 12V. Because the buck converter was using power from the boost converter, we used the Kinect's power usage, with a safety margin to account for inefficiencies in the buck converter, in the calculations for the 19V power usage.

The Kinect adapter brick was also modified in order to accept 12V power from a JST power connector. Looking back, we could have accomplished this easier by using the same splicing method that we used for the monitor, Jetson, and laptop.

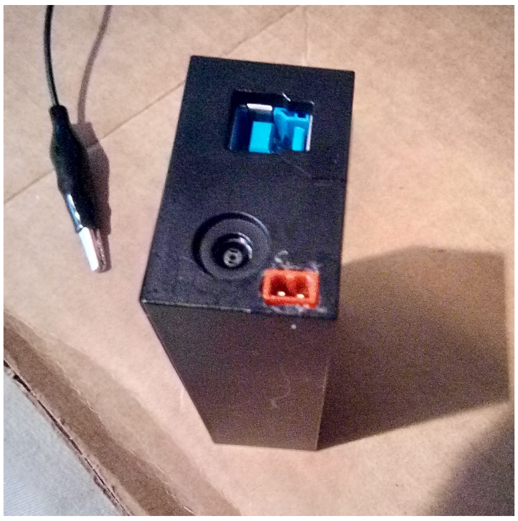


Figure 50: A kinect adapter modified with a JST connector for 12V power

Mike mahoney

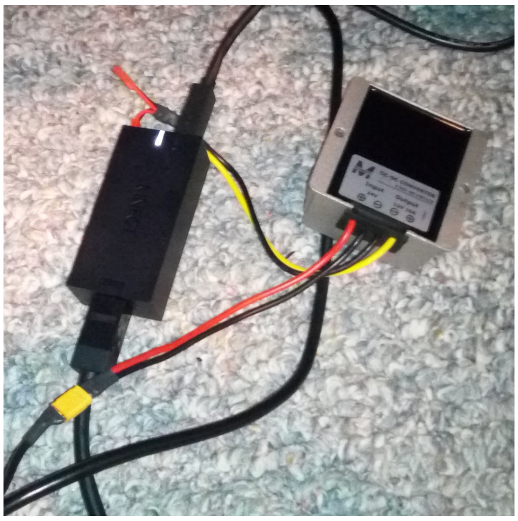


Figure 51: Testing the Buck converter with the Kinect adapter

The boost converter, buck converter, and Kinect power brick, were mounted to the dashboard support bar, where the glove box was located before it was removed.

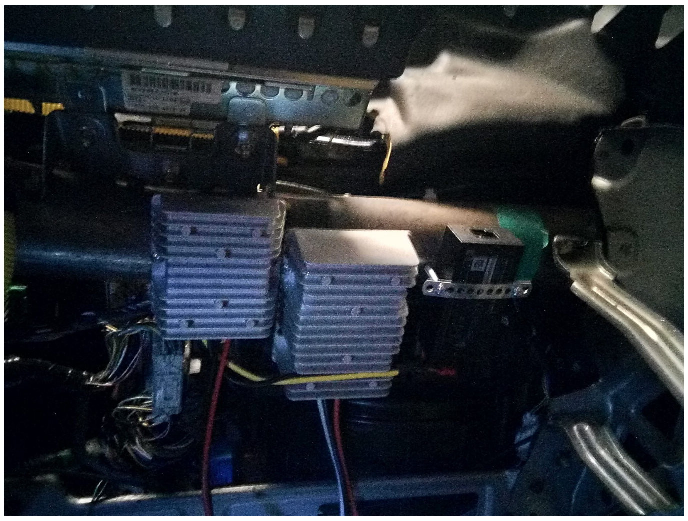


Figure 52: Mounted power system

Due to the custom power system, we have never run into any issues with drawing too much power or voltage instability.

3.5.3 Display

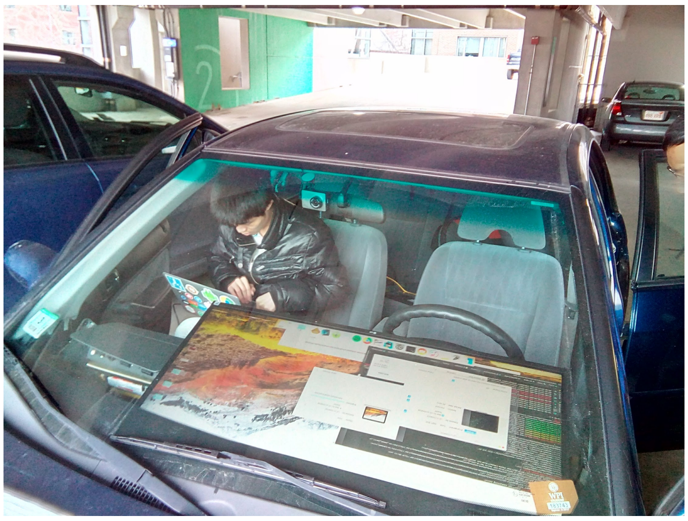


Figure 53: Monitor mounted and tested

The display used for the project is a LG 34UM69G 34 inch ultrawide. It has a resolution of 2560x1080, and can run at up to 90Hz refresh rate. We chose a 21:9 instead of a more common 16:9 display, as it allowed us to fit in a wider display without interfering with the driver's reach of the steering wheel, and increased the effective field of view of the augmented reality. The display is approximately 32.5 inches wide, and 14 inches tall, which just barely fits. The driver's side A-pillar plastic trim had to be removed for the monitor to fit without obstructing the passenger side airbag.

The monitor is mounted on custom made adjustable aluminum pillars that are bolted to the dashboard support bar. It is also secured towards the front of the monitor, directly to the frame of the car. We designed and constructed a custom frame for the back of the monitor, also out of aluminum, which these pillars bolt to. All of the edges of the frame were rounded and sanded down for safety. The pillars are adjustable, to get precise fitment of the monitor.

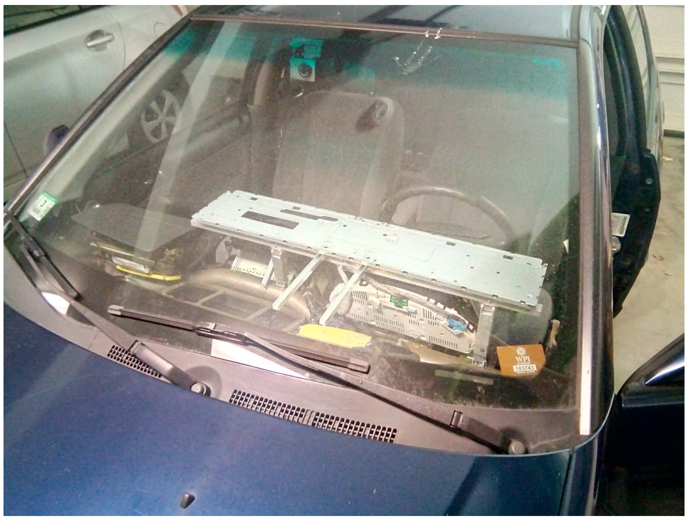


Figure 54: Monitor aluminum frame, minus the front mount

In order to fit the monitor in place, the motherboard of the monitor needed to be moved. In its original position, it would have interfered with the gauge cluster, and we would not have been able to plug either a HDMI or Displayport cable into it. It was moved from the bottom left of the monitor to the top right. New standoffs for the board were made, and attached to the monitor's frame using epoxy.

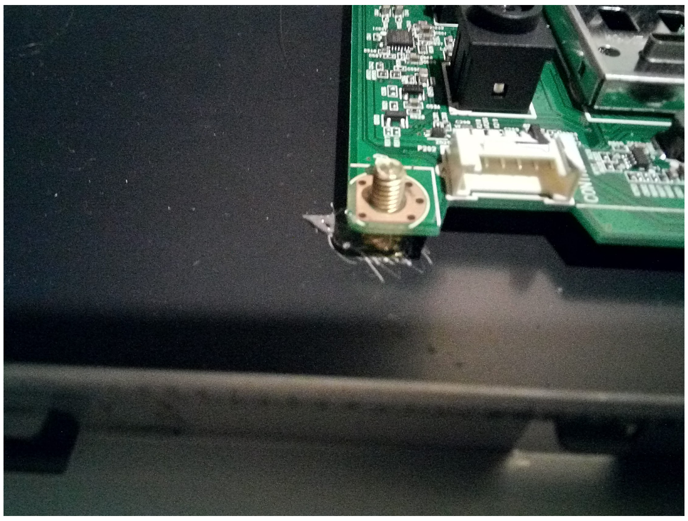


Figure 55: Modified standoffs for monitor motherboard, sans locknuts

The wires for powering the backlight were extended by splicing in Cat5 cable, and the ribbon cables that connected the motherboard to the t-con board were re-routed.

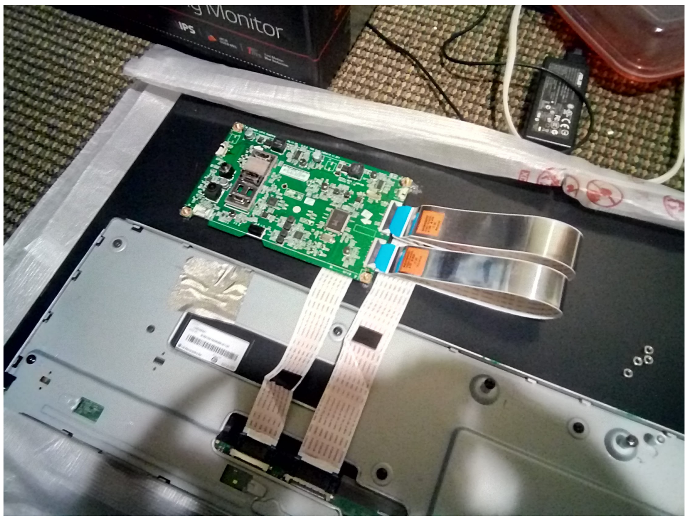


Figure 56: re-routed ribbon cables

3.5.4 Projector

Early on in the project, we experimented with the idea of using a projector instead of a computer monitor. Our plan was to project onto the surface of the dashboard, and use software to do the mapping correctly. We discovered that a projector with enough brightness to be even remotely visible during the nighttime would be rather large and bulky, and would not fit in our Honda Civic. We compared the brightness to a laptop screen, and discovered that using an LCD panel would work far better. If we had used a projector, it would have also made the raytracing in the renderer much more complex as well.

3.5.5 Kinect

Specs on the Kinect can be found on Table 2.

The Kinect was mounted onto a platform built atop the car's radio. This platform also bolted to the dashboard support bar, which had the useful side effect of giving the radio extra support, as it was not mounted very securely once the dash was removed. The Kinect's position was at a slight angle, in order to squeeze it underneath the monitor, which affected the performance of the Haar Cascades based head-tracking negatively.

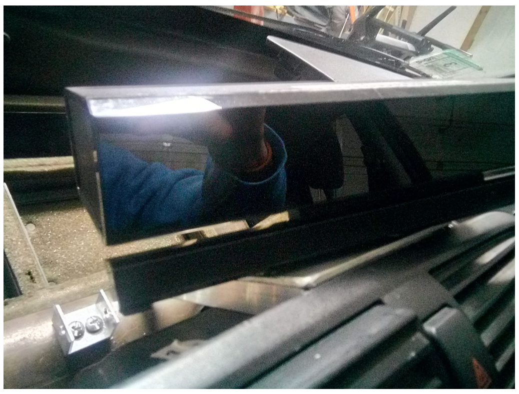


Figure 57: Kinect mounting plate in detail

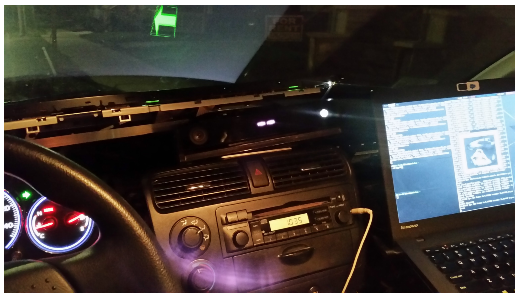


Figure 58: Kinect mounted below monitor

3.5.6 Jetson

We decided to utilize a NVIDIA Jetson TX2 because of its impressive performance with Neural Networks and parallel computing compared to consumer laptops. NVIDIA Jetson TX2 is an embedded device with fast GPGPU-like processing capabilities. It utilizes CUDA which can integrate with Tensorflow to compute high dimensional Neural Networks. Although it has a substantial amount computing power within its class of devices, TX2 is not particularly fast compared to the current Desktop GPGPUs. However, it is very suitable for our computing power needs.

Specifications:

- NVIDIA Pascal^{TM}, 256 CUDA cores (compared to 4068 for NVIDIA GTX 1080 ti)
- HMP Dual Denver 2/2 MB L2 + Quad ARM® A57/2 MB L2
- 8 GB 128 bit LPDDR4 59.7 GB/s

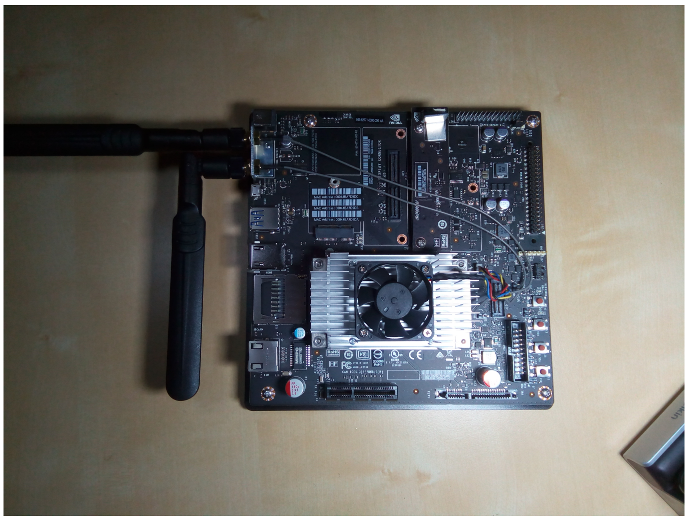


Figure 59: Nvidia Jetson TX2

4 Results

Our base goal of having 3D navigational information rendering on a windshield using head tracking was accomplished. In this section, we talk about what the final product was, and it's current limitations. In the discussions Section 5 where we explain why we got these results, and what improvements we could make to the project.

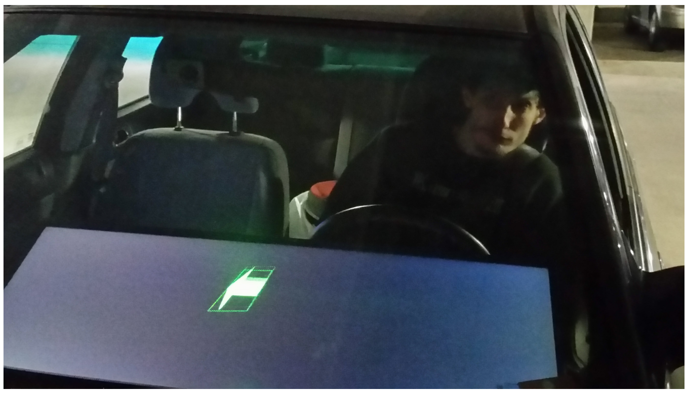


Figure 60: Driver leaning to the left

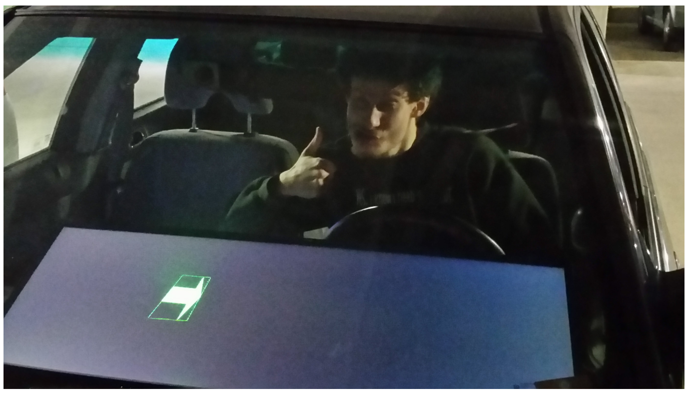


Figure 61: Driver leaning to the right

4.1 Head Tracking

By creating our own Machine Learning program to detect the head, we were able to reach faster head location updates per second than Haar Cascades. We also were able to resolve the many issues we had with Haar Cascades, as we had better control over the parameters of our detection algorithm.

Specifically, the head could be detected:

- At any angle, except for the back of the head
- At any tilt, which the Haar Cascades could not handle
- With clothing apparel such as hats
- With the face partially obscured (such as a hand)
- With a wide range of facial hair or hairstyles

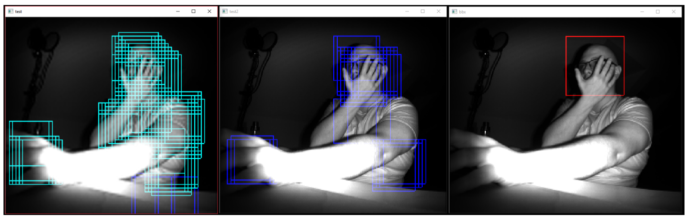


Figure 62: Head Tracking with the face partially obscured

With our final model, we were able to reach approximately 20-50 fps on the TX2, depending on how much computation was needed for the stages. This was faster than the performance of our Haar Cascades method as well.

4.2 Rendering

Modeling everything in the virtual world worked great, and we were able to see rendering in practice. While warping worked, we couldn't calibrate it to the exact values, due to the latency in head tracking. The scanning method for the windshield was also a bit imprecise, and that could have accentuated the difficulties in calibration. Finding publicly available accurate models of the windshield of the Honda Civic was impossible, and we had to resort to generating our own.

4.3 Phone App

All the required navigation data, accelerator data, and gyroscope data are transported to the rendering server in real-time. Even though the Android system had lots of limitations, such as crashing doing network operations on the main thread and crashing when manipulating the UI on the background thread, we were able to leverage the power of asynchronous tasks, the background service, and broadcast receiver to get around these issues. However, due to time constraints, the acceleration force data generated by the accelerometer and the rotation rate by the gyroscope sensor were not used in our system.

4.4 Hardware

The internals of the car, while robust and sturdy, may leave some room for improvement in the aesthetic department. The entire dashboard was removed, and while the monitor was not in the way of the driver, it did raise some safety concerns. A more advanced fabrication facility or company could produce a fully functioning dashboard with the monitor and tracking hardware cleanly installed.

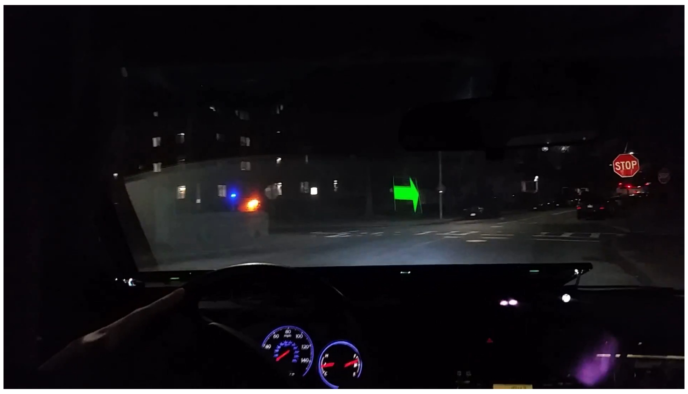


Figure 63: Arrow being rendered holographically

4.5 Driving the Car

While driving, we noticed that the monitor and the Kinect were very securely mounted, and driving over bumps did not cause any issues with the rendering or head tracking. We were also able to power everything continuously with the car's battery and alternator. Our final values while running everything on the Jetson was well over 90 fps for rendering and around 30 fps for head tracking. While the navigation arrows did move in conjunction with the head, the latency in head tracking made it difficult to see the 3D illusion well.

5 Discussion

While we were not able to construct an optimal user experience, we believe that it could easily be achieved with more hardware and software improvements.

5.1 Hardware

5.1.1 GPUs

When we developed our programs we first tested them on a regular desktop with a Nvidia Geforce GTX 1080ti. Our rendering service program achieved 800+ FPS. Our final head tracking model achieved a minimum of 90 fps. We knew that our programs would run slower on the Jetson TX2 so we ran our programs on the TX2 to tweak and measure our code. Although this workflow proved to be good enough, we believe that if we did our development purely on TX2 we could potentially achieve better results.

5.1.2 Imaging Device

We also were limited by the tracking hardware itself, as the Kinect can only produce a maximum of 30 frames per second in IR and Depth data. The back ordered Intel Realsense can produce up to 90 frames per second, which would help improve the head tracking latency and perceived throughput greatly.

5.2 Software

5.2.1 HeadTracking

Currently, our programs for Machine Learning are purely written in Python. Even though we utilize Tensorflow and Numpy, we could improve our prediction speed by switching to a lower level language like C++. Another thing to note is that our current version of Tensorflow is not integrated with TensorRT, which is an Nvidia propriety software for tensor calculations. Nvidia claims TensorRT might speed up a model's execution speed for their embedded platforms [18].

5.2.2 Rendering

One of the largest issues with the rendering and warping system was the difficulty in calibrating the windshield model. If a company were to directly incorporate this system into their cars, it would be very simple for them to include a perfectly accurate representation of the layout and windshield surface. This would fix almost all of the issues with an improperly calibrated rendering system.

5.2.3 Phone Application

One possible improvement to provide a smoother user experience is to automatically reconnect to the rendering server when the App gets disconnected. If the device goes through an area with strong wireless interference, it may potentially break the connection between the App and the rendering server. With this feature, the driver can avoid looking down the phone screen. It can also provide a more seamless user experience if the driver temporarily exited the car, such as during a break during a long trip.

5.3 Future implementations

If an automobile company were to utilized our project, they could build a car with a navigation system that is safer than current navigation technologies. The company could also integrate the project more seamlessly into the car. This could allow them to keep the hardware inside the dashboard, add extra GPUs, and also expand their HUD to the whole windshield to be perfectly aligned with the driver's view.

5.4 Future Work

Here are some potential improvements that can be made to the system:

- Reduce system latency and improve perceived throughput.
  - Use Intel Realsense instead of/in addition to Kinect.
  - Optimize headtracking process.
  - Add an external GPU onto the TX2.
  - Improve head prediction system.
- Improve headtracking accuracy: Eyetracking
- Improving the accuracy and precision of windshield measurements/calibration.
- Other helpful visuals:
  - Speedometer: GPS based, or can use OBD2 car interface.
  - Incoming texts: Prevent the user from taking their eyes off the road to read messages.
  - Radio menu.
- External API: Allow developers to display anything they want on the screen
- Virtual Buttons: Display floating buttons and use the Kinect to detect when the driver touches them.
- Overlay line for navigation: Helps the driver know when change lane, or what road is the right place to turn on.
- Pedestrian detection: Detects pedestrians who are about to get on the road, and warns the driver if they are going too fast.
- Street Sign magnification: Helps the driver read street signs by rendering a virtual one.
- Building detection: Highlights the building which to arrive in, so the driver actually knows when they have reached their destination.
- Pothole highlighting: Helps the driver avoid dangerous potholes.
- Blind spot removal/ Virtual mirrors.
- Danger/ emergency vehicle warnings.
- Self-driving car debugging.
- Integrated holographic backup camera.

References

[1] “"hudway cast",” 2018. [Online]. Available: https://hudwaycast.com

[2] J. Lee, “Projects - wii.” [Online]. Available: http://johnnylee.net/projects/wii/

[3] G. Nebehay, 2018. [Online]. Available: https://www.gnebehay.com/tld/

[4] “Face detection using haar cascades,” 2018. [Online]. Available: https://docs.opencv.org/3.4.1/d7/d8b/tutorial_py_face_detection.html

[5] [Online]. Available: https://segaretro.org/Myst

[6] J. Kampman and R. Wild, “Nvidia’s geforce gtx 1080 graphics card reviewed.” [Online]. Available: https://techreport.com/discussion/30281/nvidia-geforce-gtx-1080-graphics-card-reviewed

[7] A. Kensler, “Business card raytracer.” [Online]. Available: http://eastfarthing.com/blog/2016-01-12-card/

[8] F. Sanglard, “Decyphering the business card raytracer.” [Online]. Available: http://fabiensanglard.net/rayTracing_back_of_business_card/

[9] A. Wiltshire, “How many frames per second can the human eye really see?” 2017. [Online]. Available: https://www.pcgamer.com/how-many-frames-per-second-can-the-human-eye-really-see/

[10] “The importance of frame rates.”

[11] A. Jonsson, “Haar cascades.” [Online]. Available: https://github.com/opencv/opencv/tree/master/data/haarcascades

[12] H. e. a. Li, “A convolutional neural network cascade for face detection.”

[13] F. Policarpo and M. M. Oliveira, “Gpu gems 3,” 2007. [Online]. Available: https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch18.html

[14] E. Wallace, “Webgl water.” [Online]. Available: http://madebyevan.com/webgl-water/

[15] S. of Fairlight, “Real time ray tracing part 2.” [Online]. Available: https://directtovideo.wordpress.com/2013/05/08/real-time-ray-tracing-part-2/

[16] “Running cables through the firewall in a 2002 honda civic,” 2011. [Online]. Available: https://www.youtube.com/watch?v=xdi_9XD55KU

[17] M. Mahoney, “Jst rcy connector.” [Online]. Available: https://commons.wikimedia.org/wiki/File:JST_RCY.JPG

[18] “Nvidia tensorrt.” [Online]. Available: https://developer.nvidia.com/tensorrt

6 Appendices

A Github links to source code

Wiki for all of our Meetings
https : //github.com/NaviRice/NaviRice/wiki

Repository for the Renderer
https : //github.com/NaviRice/renderer

Repository for Head Tracking
https : //github.com/NaviRice/HeadTracking

Repository for the mobile application
https : //github.com/NaviRice/android − app

Repository for the KinectServer
https : //github.com/NaviRice/KinectServer

B Protofiles

B.1 Phone application -> Rendering server

```
//step.proto
syntax = "proto2";

package navirice.proto;

message Step {
   required double latitude = 1;
   required double longitude = 2;
   required string description = 3;
   required string icon = 4;
}

//accelerationForce.proto
syntax = "proto2";

package navirice.proto;

message AccelerationForce {
    required float x = 1;
    required float y = 2;
    required float z = 3;
}

//requestHeader.proto
syntax = "proto2";

package navirice.proto;

message RequestHeader {
    enum Type {
        CURRENT_STEP = 0;
        CURRENT_LOCATION = 1;
        CURRENT_ACCELERATION_FORCE = 2;
        CURRENT_ROTATION_RATE = 3;
        ON_BUTTON_CLICK = 4;
    }
   required Type type = 1;
   required uint32 length = 2;
}

//response.proto
syntax = "proto2";

package navirice.proto;

message Response {
    enum Status {
        SUCCESS = 0;
        BAD_REQUEST = 1;
        NOT_IMPLEMENTED = 2;
    }

    required Status status = 1;
}

//rotationRate.proto
syntax = "proto2";

package navirice.proto;

message RotationRate {
    required float x = 1;
    required float y = 2;
    required float z = 3;
}
```

B.2 KinectServer -> HeadTracking

```
//navirice_image.proto
syntax = "proto3";

package navirice;

message ProtoImageCount {
        uint64 count = 1;
        uint64 byte_count = 2;
}

message ProtoAcknowledge {
        enum ACK {
                NONE = 0;
                CONTINUE = 1;
                STOP = 2;
        }
        ACK state = 1;
        uint64 count = 2;
}

message ProtoImage {
        uint32 width = 1;
        uint32 height = 2;
        uint32 channels = 3;
        enum DataType {
                FLOAT = 0;
                UBYTE = 1;
        }
        DataType data_type = 4;
        uint32 data_size = 5;
        bytes data = 6;
}

message ProtoImageSet {
        uint64 count = 1;
        ProtoImage RGB = 2;
        ProtoImage Depth = 3;
        ProtoImage IR = 4;
}

message ProtoCaptureSetting {
        bool RGB = 1;
        bool Depth = 2;
        bool IR = 3;
        uint64 count = 4;
}

message ProtoRequest {
        enum ReqType {
                IMAGE = 0;
                CAPTURE_SETTING = 1;
        }
        ReqType state = 1;
        uint64 count = 2;
        ProtoCaptureSetting capture_setting_value = 3;
}
```