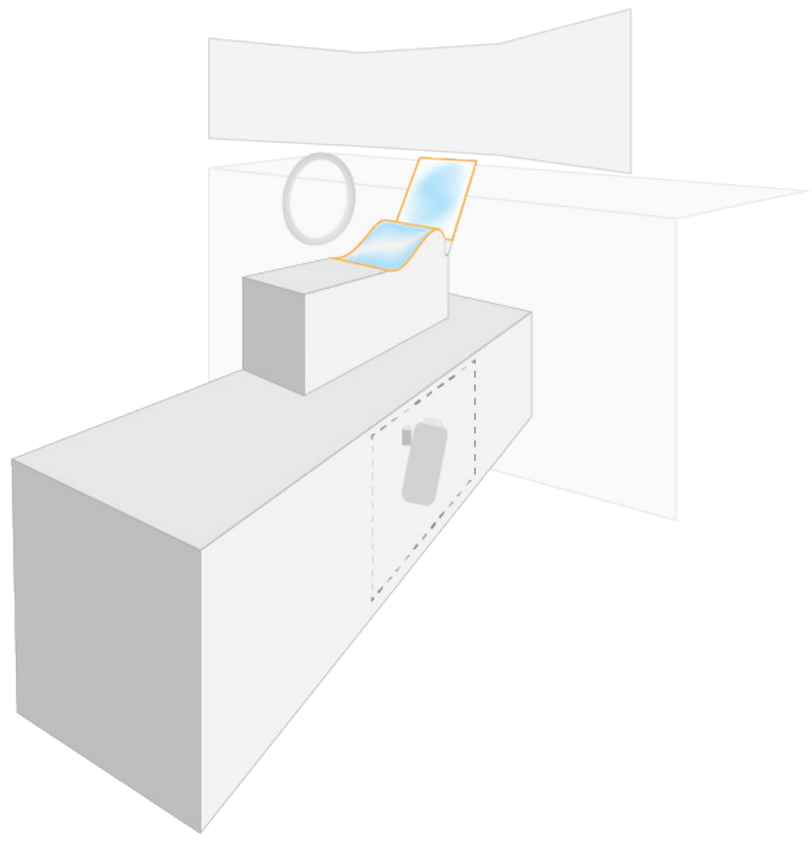
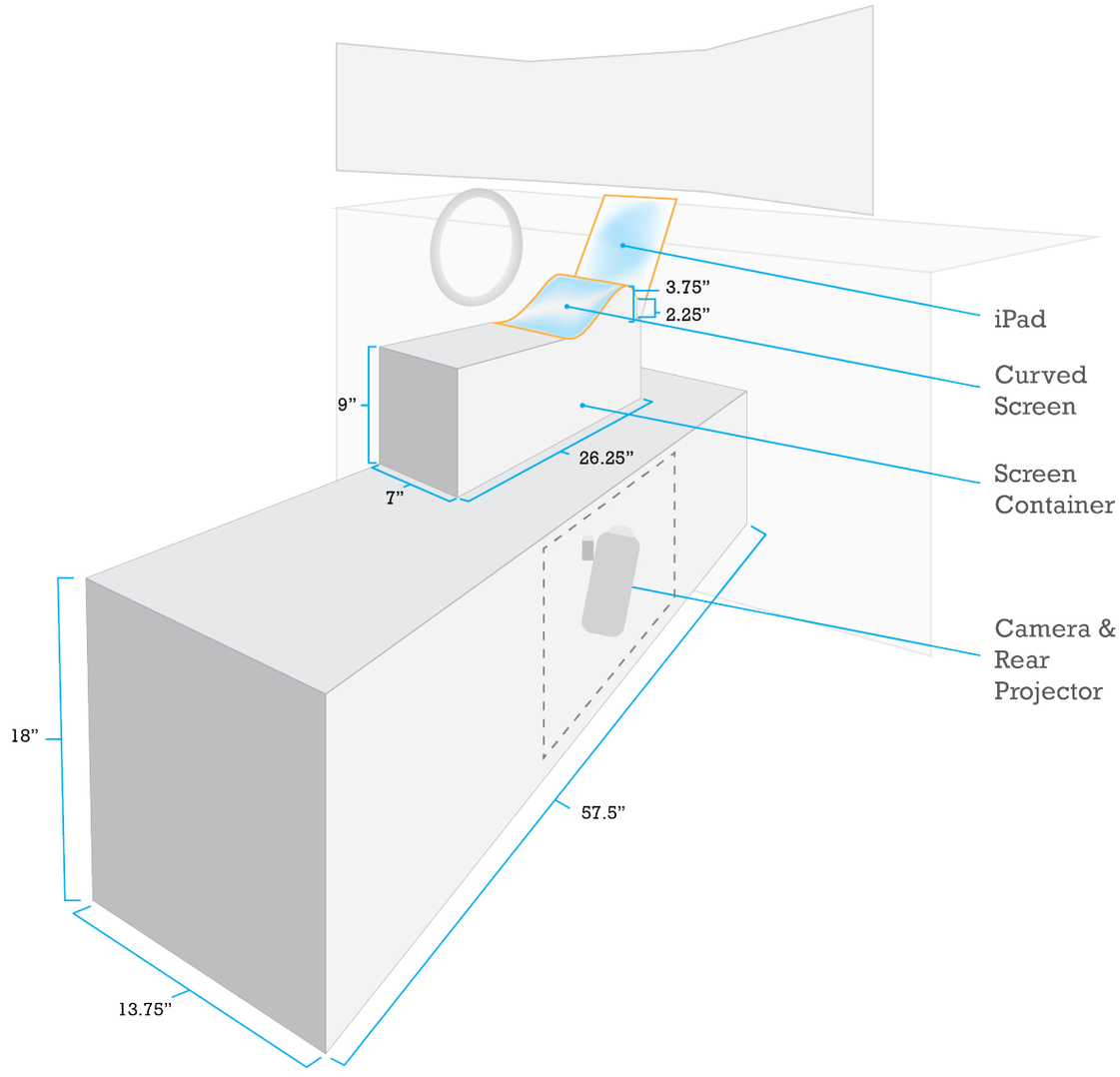
Human/AI Collaborative Control technical diagram



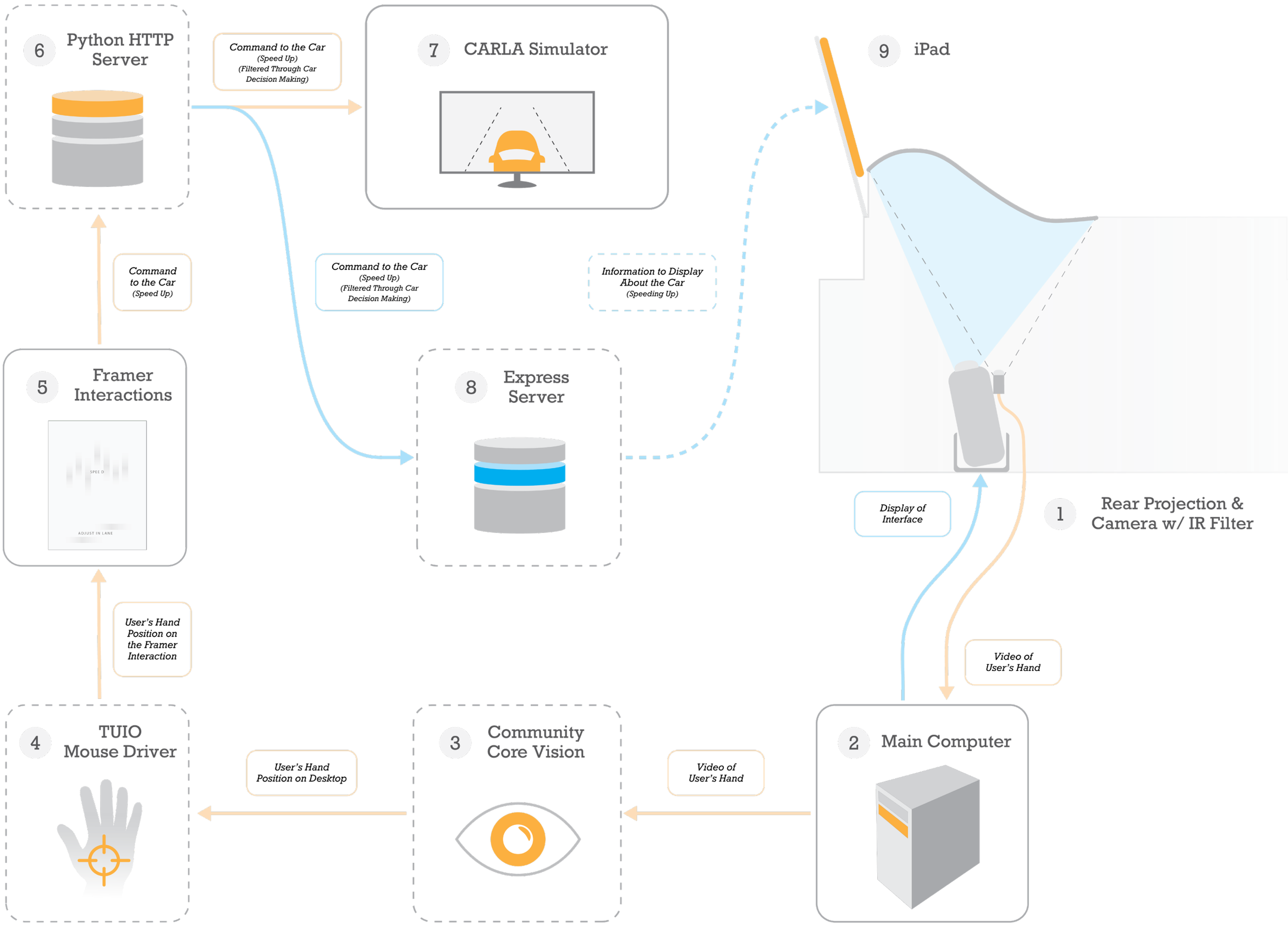
About This Document

This document outlines the technical process used to create an interactive, curved plexiglass control to be used in conjunction with controlling an autonomous vehicle. Our system creates a complete interaction experience, with user input resulting in real-time car movement (via the CARLA driving simulator (link)) and visual feedback on an iPad confirming the user's action and the state of the car.

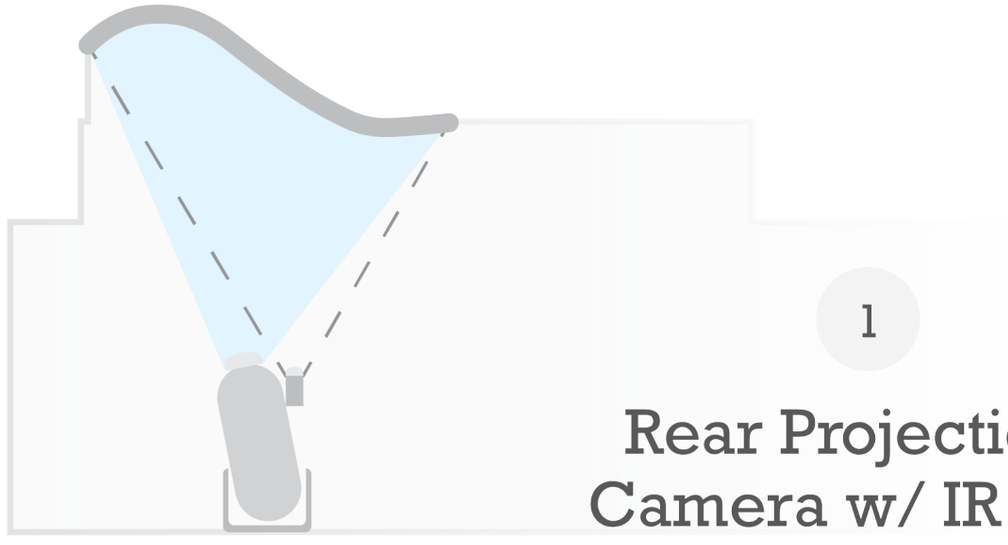
On the next page is a diagram outlining the individual components used to create our interactive environment, which is to be used in conjunction with the more detailed description of each part later in this guide. Here is a guide for measurements in re-creating our setup.



The Diagram


6 Python HTTP Server
Command to the Car (Speed Up) (Filtered Through Car Decision Making)
7 CARLA Simulator
9 iPad
Command to the Car (Speed Up) (Filtered Through Car Decision Making)
Information to Display About the Car (Speeding Up)
5 Framer Interactions
8 Express Server
Display of Interface
1 Rear Projection & Camera w/ IR Filter
User's Hand Position on the Framer Interaction
Video of User's Hand
4 TUIO Mouse Driver
User's Hand Position on Desktop
3 Community Core Vision
Video of User's Hand
2 Main Computer

1 Rear Projection and Web Camera



Our interactivity begins with our faux curved touchscreen. Instead of one of these expensive displays, our model begins with a curved piece of plexiglass. This piece of plexiglass is backed with a rear projection film, allowing us to project a desktop from our computer onto it (giving the appearance it is in fact a screen).

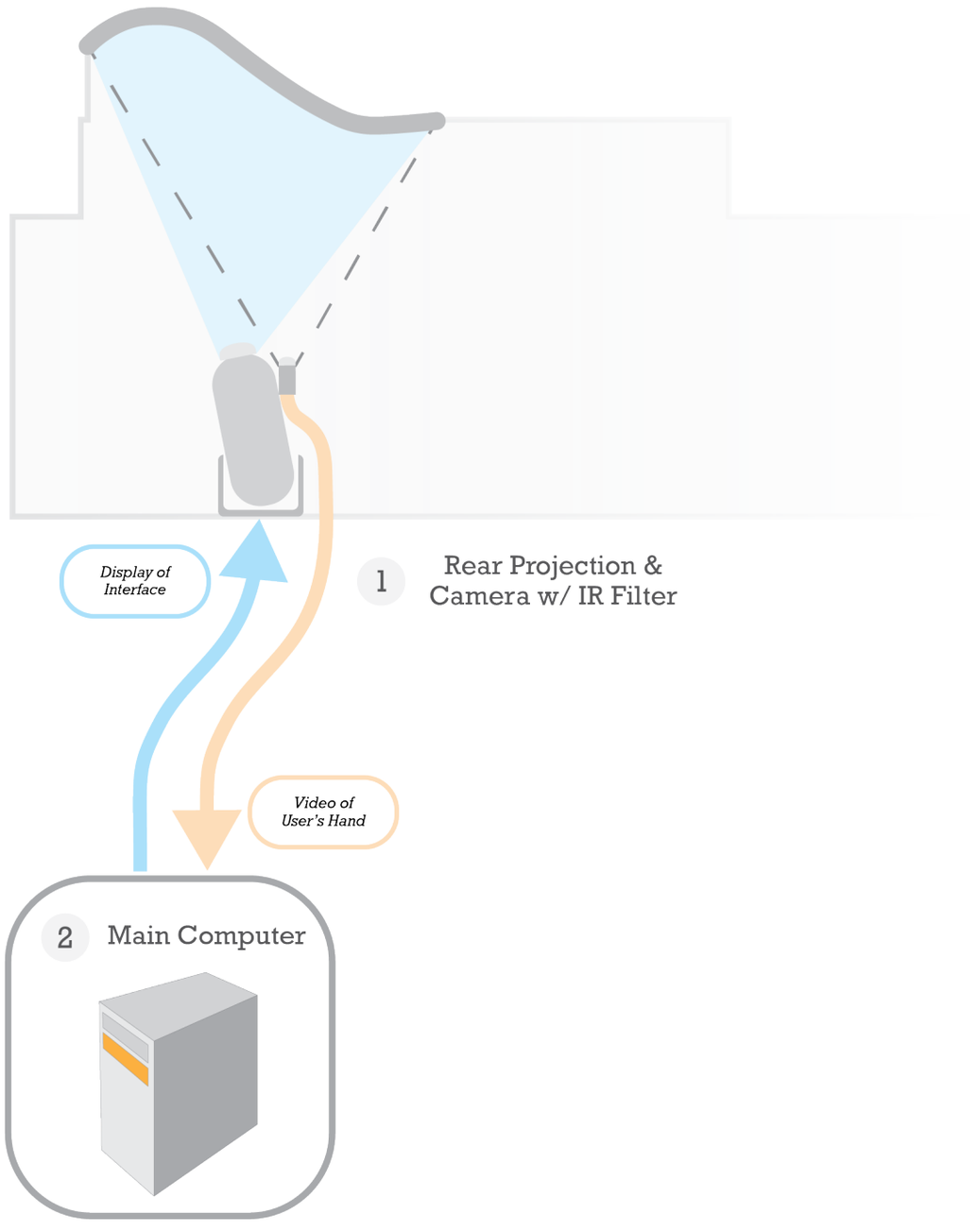
From here we have our rear projector. This serves to project the interactive interface onto the curved plexiglass surface. For our setup, we needed to be mindful of the space constraints of a car space, so our projector is angled upwards, being held up by a supporting box made of foam core. This projector was attached to our main computer via HDMI to display an extended desktop.

In order to bring make our interface projection interactive, we decided to invoke computer vision to track the position of the user's hand and relay that to our framer interface (spanning steps 1-5 of this technical setup). In order to capture the position of the user's hand, we first needed to take in a video feed of the user's hand. This webcam allowed us to relay raw images of the user's hand to the main computer, and then our computer vision software. The webcam was attached to our main computer via usb.

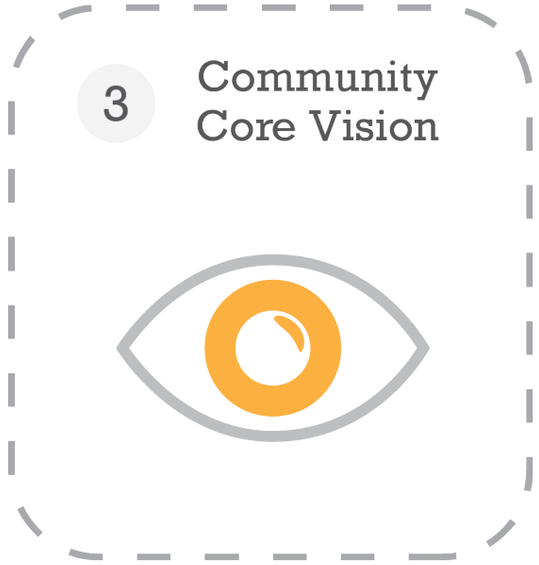
2 Main Computer

Our main lab computer served as the hub for displaying our car simulator, CARLA, as well as processing the commands of the user through our interface before updating the CARLA simulation in real time and sending information to another server to display information on our secondary screen iPad.

The computer is a Windows 10 MSI MS 7A63, sporting an Nvidia GTX 1080 graphics card. It was connected to a local wireless router in our testing room.

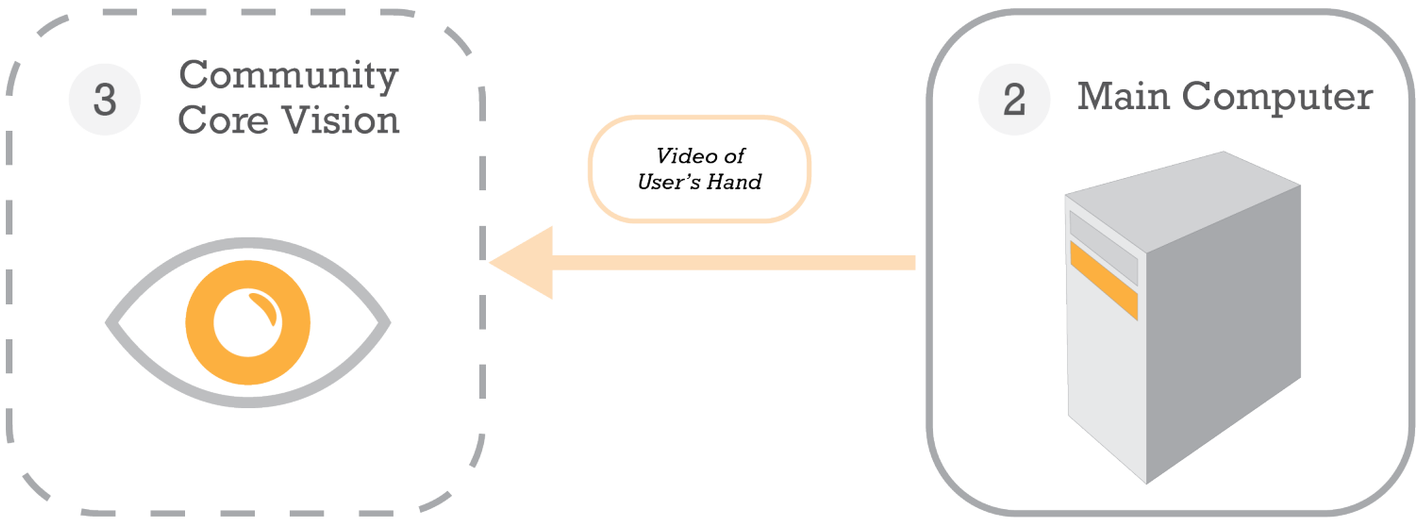


3 Community Core Vision (CCV)

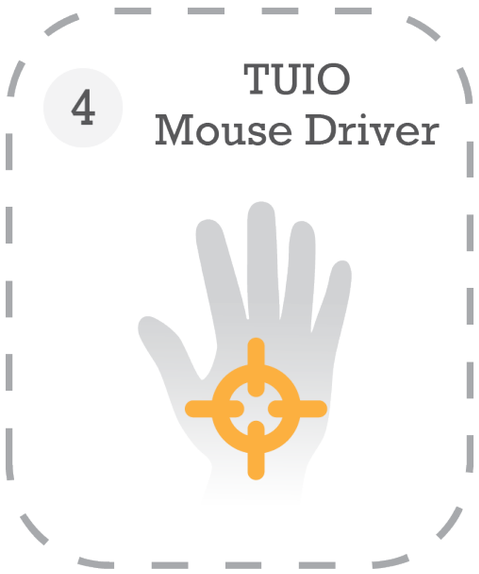


Community Core Vision is a freely-available software used for tracking the position of objects in space. It is optimized for use in tough tables, or other projection-based interfaces. It can be found at http://ccv.nuigroup.com/. For our setup, we used version 1.5 on Windows.

Community Core Vision was used as a means of capturing the position of the hand in space, and relaying such coordinates to a separate driver for processing. Community Core Vision recorded the position of the hand as an absolute positioning relative to the desktop canvas (1920x1080) and passed that along to the TUIO mouse driver.



4 TUIO Mouse Driver



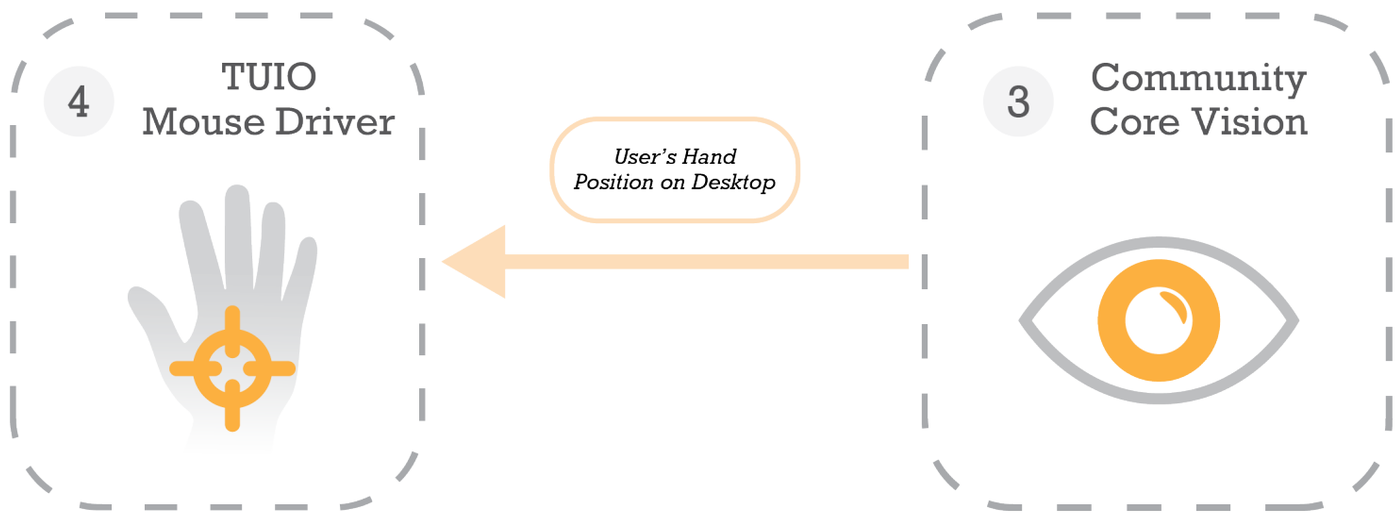
Once we were able to obtain the position of the hand in two-dimensional space from Community Core Vision, we sent this positioning to a customized driver to control the main computer and move the mouse on the computer to where the user's hand was. Community Core Vision features the ability to send commands via the TUIO protocol to listening programs. The TUIO protocol is a standard used in rear-projection interfaces outlining the means by which hand position is relayed to programs. In our case, we used a JAVA-implemented mouse driver to eventually control the operating system mouse cursor based on the user's hand position.

Once the hand position was passed to our JAVA program via the listening TUIO protocol, we needed to scale it to fit onto the canvas of the interface the user had access to. Not all 1080 pixels of height were visible by the camera, and so we had to scale the y position of the hand in order to allow the user to have access to the full vertical height of the interface. A similar process needed to be performed for the width.

Once we had our scaled height and width coordinates of the user's hand, we then used the same JAVA program to move the mouse to such position and click. As the user would then slide their hand around the interface (so long as they kept their hand on the interface), the mouse would move as a dragged click. This helped to prevent false positives during testing and allowed for easier resetting if need be.

More about the TUIO Protocol can be found at https://www.tuio.org/, with the driver our customized solution was based off of at https://github.com/mkalten/TUIO11_Mouse. The code can be found in the CMU Harman Handoff GitHub Repository at https://github.com/jestapinski/CMU_Harman_Handoff/tree/master/TUIO_mouse_driver_code

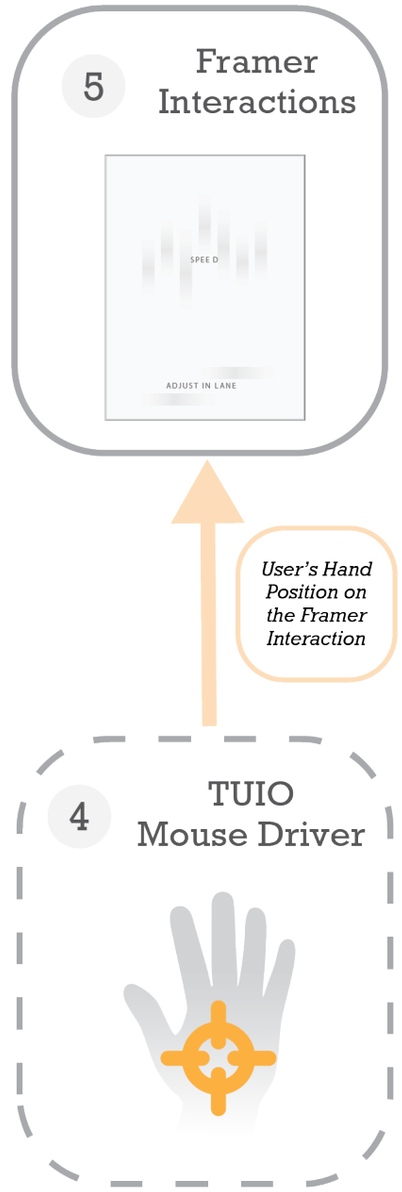
Running the TUIO Mouse Driver from a bash terminal can be done with the command `java -jar TUIOMouse.jar`.



5 Framer Control Interactions

Knowing where the user was touching on the interface, we now passed these events over to our user interface, engineered in the Framer JS environment. Framer JS is an interactive design tool which allows developers to add interactive elements to design prototypes through Coffeescript. More about Framer JS can be found at https://framer.com/.

These events, because they were recorded as mouse click and drag events from the TUIO mouse driver in step 4, were registered in Framer as “Touch Start” and “Touch Move” events. Our main task was figuring out how to register the intent of the user and translate that intent into commands to send to our driving simulation and secondary screen iPad for feedback. Our approach involved using separate zones for speed and adjust in lane, with different height and width thresholds to avoid false positives. The user would have to move at least a certain distance horizontally in the lane adjust zone to be able to perform a lane adjust, and likewise for speed control. This was helpful in preventing false positives and also helped in allowing us to better capture the user’s intent and displaying immediate feedback.

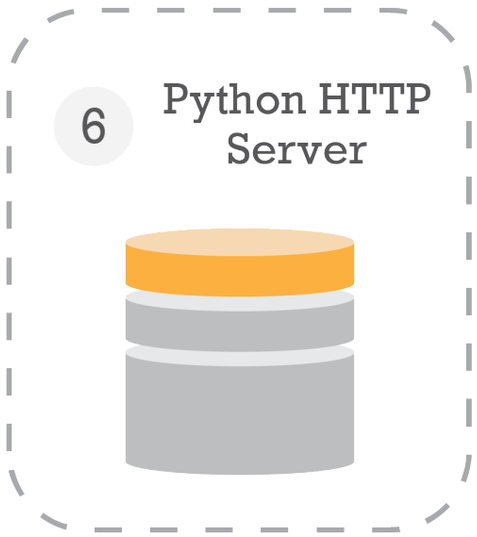


Once we were able to figure out exactly how to recognize the user’s intent, we created a custom module for Framer to use to communicate with our custom CARLA Python HTTP Server, called “CARLA_API”. The main idea is when we figured out what the user wanted to do to the autonomous vehicle, such as speed up, we sent a POST request to our Python HTTP server with a message informing the server we now wanted to speed up the car, and the Python HTTP server would take care of the rest.

The Framer code can be found in the CMU Harman Handoff GitHub Repository at https://github.com/jestapinski/CMU_Harman_Handoff/blob/master/framer_control_and_secondary_screen.txt

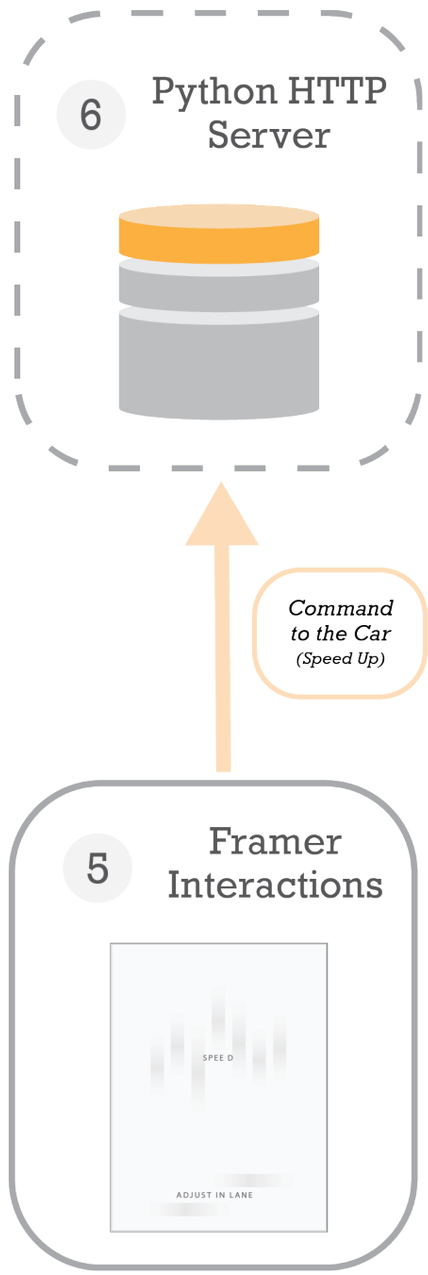
The Framer code can be run by using the IP:PORT link from running in browser preview in Framer Studio, and ensuring the main computer and where the Framer code is running from are on the same wifi network.

6 Python HTTP Server



The crux of communicating the user's intent to the live car simulation and providing feedback via the secondary screen iPad lies in the Python HTTP Server. The server listens for incoming requests to control the car via POST requests coming in from the Framer user interface prototype. Once a command is received, the Python server checks to see if the car in the CARLA simulation is able to perform the action requested by the user (the car should not speed up, for example, if it is at a red stoplight) before sending the action to the car to perform in real-time.

The Python HTTP Server maintains a queue of actions that the user has requested the car do, and performs them as it is safe to do so. However, we first check to see if the car is in a safe position to perform the move, allowing for this collaborative control experience we are vouching for in autonomous vehicles. The CARLA driving simulation features the ability to control the car remotely via a client/server architecture. Our Python HTTP server uses the CARLA Client Python API to be able to send commands to the car, with changes visible to our test subjects on the screen.



At the same time, we also need to send the car command to our secondary screen iPad for additional feedback to the user as to what the car is doing (as a form of transparency). We use an Express server to do so, and so we send a GET request to this Express server in order to begin to display feedback directly on the iPad.

The code for the Python HTTP Server (with its contextual CARLA API files) can be found in the CMU Harman Handoff GitHub Repository at https://github.com/jestapinski/CMU_Harman_Handoff/blob/master/python_http_server.txt.

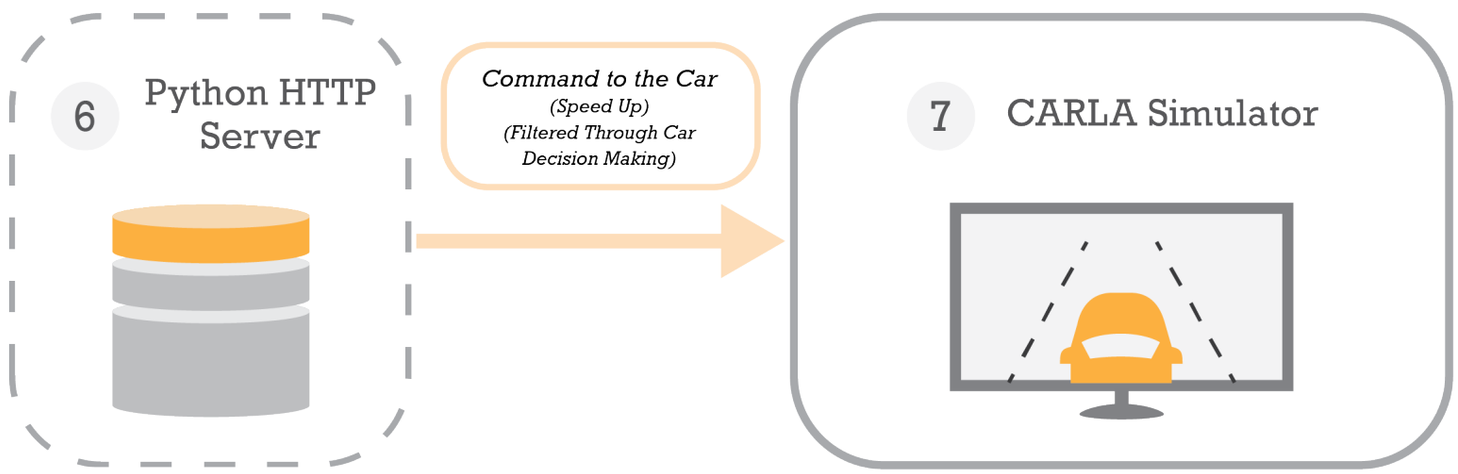
This script can be run from a terminal/command line with the command `python use_case_controller.py` (Python 3).

7 CARLA Driving Simulator

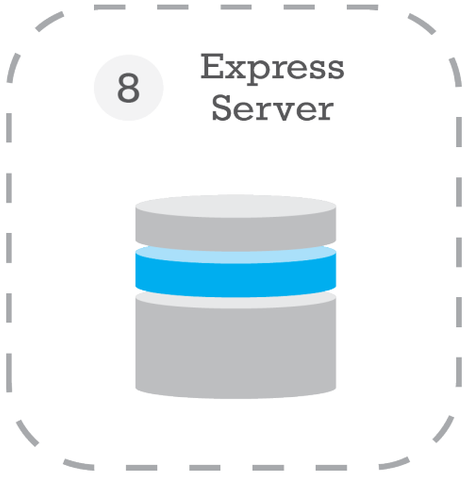
For placing our test subjects in a realistic driving experience (without the liability of real driving), we adapted version 0.8.4 of the CARLA car simulator. The simulator is an open-source project freely available for autonomous driving research, and can be found at https://github.com/carla-simulator/carla. As alluded to, CARLA runs on a client/server model, meaning our Python HTTP Server acts as a client which tells the server-side vehicle how to drive. The server, in return, provides details as to the objects surrounding the car, such as other vehicles, pedestrians, speed limit signs, traffic lights, and more.

The simulator is highly customizable, from weather to the number of cars and pedestrians on the road. The client/server architecture also allows for more fine tweaking of the autonomous driving experience through our Python HTTP server, however the CARLA car is able to drive completely on its own (respecting traffic lights, other vehicles, etc).

The CARLA Simulator can be started from the command line using the -carla-server flag. Thus the entire command is `./CarlaUE4 -carla-server`. This will hang until a server connects on the default port (2000). Note you can also customize the CARLA settings using a .ini file. An example of one such customization file can be found at https://github.com/carla-simulator/carla/blob/master/Docs/Example.CarlaSettings.ini.



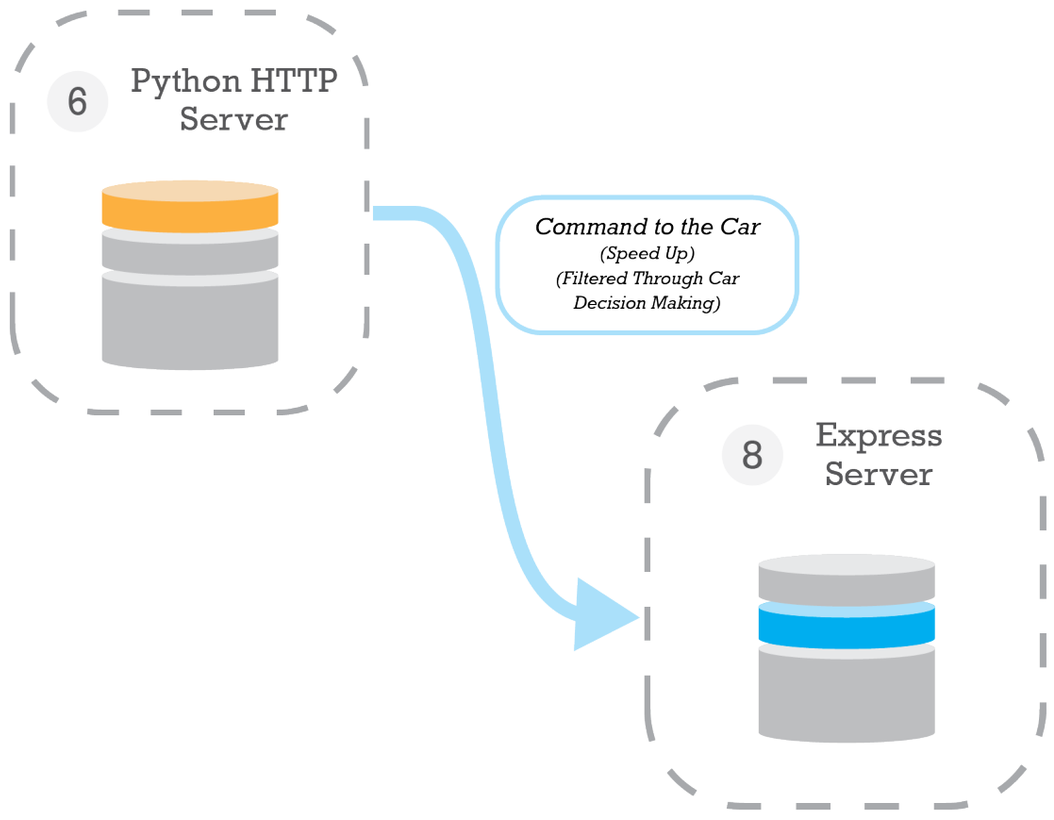
8 Express Server



Once we know how the behavior of the car is going to change with respect to the user's input, we needed a means to route this information to our secondary screen iPad. Our solution was an Express server which would route commands straight to the secondary screen iPad. The Express server would listen for incoming GET requests, and simply send those messages out to the secondary screen iPad listening via a websocket connection.

Code for the Express server can be found in the CMU Harman Handoff GitHub Repository at https://github.com/jestapinski/CMU_Harman_Handoff/blob/master/framer_control_and_secondary_screen.txt under Secondary Screen Server POC.framer as `server.js`.

The Express server can be run from a terminal/command line with the command `node server.js`.



9 iPad (Secondary Screen)

9 iPad

Finally, the secondary screen iPad would receive the socket message from the Express server detailing what is happening to the car (speeding up, slowing down, shifting in lane, etc). Our Secondary Screen iPad was running a different Framer prototype than the user interface being projected onto the curved plexiglass surface. This iPad Framer prototype would run our visual animations based on the websocket message being received.

Code for the Secondary Screen Animations in Framer can be found in the CMU Harman Handoff GitHub Repository at https://github.com/jestapinski/CMU_Harman_Handoff/blob/master/framer_control_and_secondary_screen.txt