Gesture.js: A Cloud-Deployable Framework for Building Embodied Experiences

by

Azaria Fowler

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2022 by the
Graduate Supervisory Committee:

ARIZONA STATE UNIVERSITY

May 2022

ABSTRACT

Emerging body movement detection and gesture recognition software have opened a gateway of possibilities to make technology more intuitive, engaging, and accessible for people. A vast area of natural user interfaces is leveraging body motion tracking and gesture recognition technologies and a human's readily expressive body to extend interactions with software beyond mouse clicks and scrolls. However, these interfaces have been limited by hardware and software expenses, high development time and costs, and learning curves. This paper explores different approaches to providing both software developers and designers with easier ways to incorporate computer vision-based body and gesture detection solutions into the development of embodied experiences without suppressing creativity. Gesture.js is a JavaScript framework as a service (FaaS) that is both a thin library on top of the Document Object Model (DOM) consisting of a collection of tools for developing embodied-enabled applications on the web and a landmark computation and processing application programming interface. It wraps MediaPipe, an open-source collection of machine-learning solutions that perform inference over arbitrary sensory data, and additional landmark processing frameworks such as KalidoKit, a 3D model rigging solution, and ports the necessary information through either an object-oriented or an API-oriented implementation. It also comes with its web-based graphical interface for easy connection between Gesture.js and other application clients with little to no JavaScript code. This thesis also details a collection of example applications that demonstrate the usability, capacity, and potential of this framework.

Keywords: gesture recognition, embodied applications, cloud computing

DEDICATION

I would like to dedicate this thesis to my beloved family members and close friends. I am truly thankful for having a constant source of encouragement and affirmation from them during my graduate school journey. I am deeply grateful for the support and motivation of my parents, Yu-mei and Todd Sinclair, who have encouraged me to apply to graduate school even when I felt I would have never succeeded in doing so. Without them, I would have never been able to become the person I am today.

ACKNOWLEDGMENTS

Throughout my graduate school journey, I have received a great deal of support and inspiration from several people at Arizona State University. I would like to thank my thesis chair, Dr. Tejaswi Gowda, whose expertise was invaluable to not only my research but software development career. Your insightful feedback has pushed me to become more resourceful when solving my problems and has brought my work to a higher level. I would also like to acknowledge my committee members, Dr. Stacey Kuznetsov, and Dr. Yoshihiro Kobayashi, for taking the time to guide me throughout my research. Your expertise has been both inspiring and insightful to my thesis, and I am grateful for all the tools you have provided me to choose the right direction in my research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The ability to move your body is an essential element in interactive settings. It is natural to result in gestures to communicate with others, use your hands to interact with objects, or even use your fingers to count. This is because the way we interact with the world, stored as a set of abilities and skills, is stored not only in our mind but in our entire body (J, 2019). Similar to muscle memory, body intelligence is our learned interactions with the world and can be used to strengthen our cognitive abilities. The idea of leveraging this relationship between one's body movement and their overall experience with the world is the study of embodied cognition. More specifically, the exploration of one's psychological factors concerning bodily sequences during an interaction with technology is the study of embodied interaction.

When a user is allowed to move their body naturally and unrestrictedly, it becomes significantly easier for them to adapt to a new piece of technology or software program (Schrammel, Paletta, & Tscheligi, 2010). This datum has inspired the development of natural user interfaces in human-computer interaction. These interfaces utilize natural human actions such as voice commands, body movement, or gesture and pose recognition as an input to control a software application. More importantly, what makes an interface "natural" is the user's behavior and feeling during the experience by better assisting their human abilities, skills, and processes (Wigdor & Wixon, 2011). This thesis will focus on a specific area of natural user interfaces-- those that leverage pose, face, hand tracking, and gesture recognition. This collection of applications will be referred to as embodied applications.

While embodied applications are worthwhile, there lacks emphasis on the role of body movement in the field of human-computer interaction (Schrammel, Paletta, & Tscheligi, 2010). This is due to technological barriers, high hardware and software costs, and a lack of standardization in the development of embodied applications. There are two approaches to capturing and interpreting body movement. The first approach is known as the hardware-based approach, where the user is required to wear some form of device that collects data via sensors and sends it to the computer. Unfortunately, the current devices developed tend to be bulky and expensive in to be accurate, which then hinders the naturalness and practicality of one’s interactions with a software application (Yeo, Lee, & Lim, 2013). The second approach is known as the computer vision-based approach, where cameras and computer vision techniques are used to collect and interpret body movement. This approach is a lot more natural for the user but lacks accuracy if the analyzing software is weak or the camera is of low quality (Yeo, Lee, & Lim, 2013).

Fortunately, with the improvement of computer hardware (i.e., higher quality webcams, faster computation speeds) and the introduction of highly accurate machine-learning solutions for a live video stream, the role of movement in human-computer interaction is starting to make a comeback. MediaPipe, an open-source collection of machine-learning solutions that perform inference over arbitrary sensory data, such as live video (Lugaresi, et al., 2019), provides developers the opportunity to integrate body detection and tracking without the need to train their models, purchase, and integrate hardware, or pay for a subscription or license. The introduction of this framework has sparked many ideas in the software development and research community, such as sign language interpreters, augmented reality applications, and expression or gesture recognition machine learning models. The possibilities are endless.

MediaPipe is powerful and provides a gateway of opportunities for embodied applications. However, it only provides the inference data, more specifically a collection of landmark coordinate points for each body part in view. The ability to interpret this data semantically such as if a gesture is being made or make use of it by mapping a landmark location to a point on a webpage or is not provided and up to the user’s implementation choice, process, and skills. This can be discouraging for those who are unfamiliar with the development of embodied applications and especially limiting to those who lack access to gesture recognition machine learning models. Moreover, while machine learning inference and processing is fast for common devices, it is built-in to the device which requires processing power. This thesis will explore different approaches to providing both developers and designers with a more streamlined, accessible development experience of embodied applications.

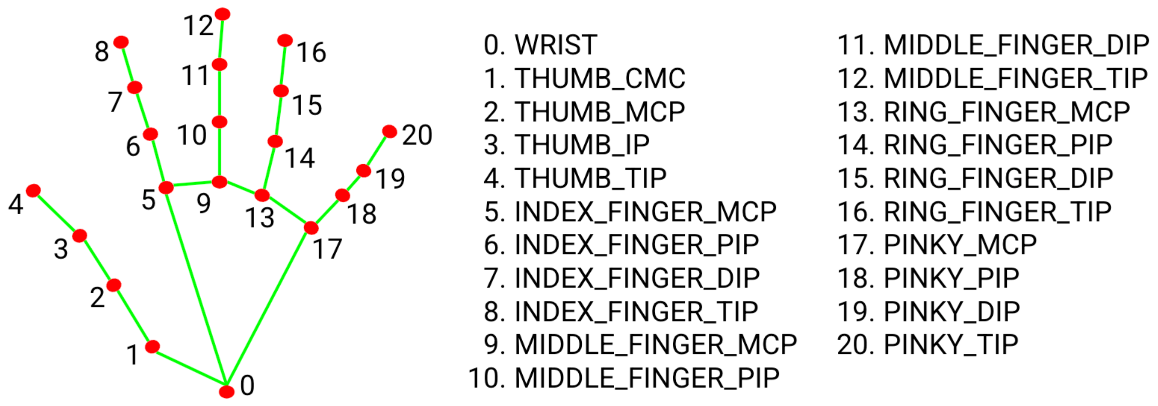
CHAPTER 2

BACKGROUND LITERATURE

MediaPipe hosts a collection of machine-learning solutions that fundamentally analyze a live video stream and compute a set of landmark points. A landmark point is a three-dimensional coordinate composed of x, y, and z. x and y are normalized to [0.0, 1.0] by the image width and height respectively, while z represents the landmark depth (MediaPipe, 2020). In Figure 1, a depiction of the landmark model for the Hands solution is presented.

Figure 1

MediaPipe Hands Landmark Model



While this information is mobilizing, the interpretation of this data requires additional technological support. For example, the integration of a mapping algorithm of these landmark points to a specific location on a website or a machine-learning-based gesture recognition layer. These technologies are highly dependent on the developer of an embodied application as there is no standard programming language or framework that bundles all these technologies to develop embodied applications. In addition to this, MediaPipe's solutions are built-in, meaning that the final application not only has to semantically process the landmarks output but also run the process that produces the landmarks. While MediaPipe is fast and lightweight, there are opportunities to package both the production and processing of landmark data as a service.

Building Embodied Experiences

When the field of computing software was born, it became obvious that a more standardized approach to the development of software was needed. This is because as the software was seen to solve problems in the world, many methodologies on how to develop this software were also proposed. This resulted in a “methodology jungle,” which is the difficulty of selecting the appropriate methodology for a given software development project. To combat this, frameworks were introduced to package certain methodologies in a way that abstracts upon the design or architecture of a specific system (Mnkandla, 2009). A software framework is a code skeleton that abstracts upon a specific implementation of a solution.

In this paper, the solution is creating embodied applications, more precisely, applications that interpret body detection and tracking data semantically. There is no framework or specific implementation that abstracts upon or standardizes this for software developers. NoTouch.js is a front-end development framework for interacting with websites with hand gestures. It is great at semantically wrapping gestures into an action, such as mapping a pointer finger’s curl to a click. However, it is limited as it only provides a click interaction and does not allow flexibility for the user to choose what gesture results in a click (Akcura, 2018). Gest.js is like NoTouch.js but gives developers more flexibility as to what a gesture could mean. It is computer vision-based and focuses on listening and using pre-defined gestures to define an interaction (Micheal, 2013). However, unlike NoTouch.js, the coordinates of where one’s hand is to the application interface are not given, and so gestures are difficult to map interactions to a specific element on a website.

The closest framework that attempts to interpret both gestures and body landmark point coordinates into interactions with a website is Handsfree.js. Unlike the previously introduced frameworks, Handsfree.js is powered by MediaPipe’s machine learning models. This allows the developer to directly access the raw landmark data but also have access to a collection of plugins that interpret the data (Ramos, 2021). These plugins are tremendously useful for generic interactions like click and scroll, but once again, they constrain the interaction by locking in the gesture. Also, Handsfree.js' interactions are restricted to only clicks and scrolls, which at first seem sufficient as these interactions are almost equivalent to what a mouse can do, but compress the capability and potential of embodied applications. Adding a new dimension to webpages transcends beyond simply the addition of input devices; They should capitalize increased expressivity our bodies can provide as an input to software applications.

Pose and Movement Detection as a Service

Simply interpreting landmark data into a collection of useful interactions for developers is not enough. Another key element that is missing from these frameworks is that the computation and generation of landmark data require it to be computed on the application itself. This is problematic for users of a specific application, especially if they are running it on a slow computer, and developers need to make heavy use of processors for other reasons such as rendering a high-fidelity virtual world. For example, TouchDesigner is a software development platform for building real-time interactive multimedia content and makes heavy use of a computer's CPU for video and audio reading and decoding and GPU for rendering media (Derivative, 2021). Therefore, the production and processing of landmark data are not feasible for experiences created in TouchDesigner. A framework as a service (FaaS) offers solutions by production environment type (i.e. a web application) by providing the foundation to rapidly develop an application. It can be customized according to business needs but does not require the full implementation of a system (McKenzie, 2014). There is a need for this in embodied application development for software developers and designers.

This paper proposes a framework as a service, Gesture.js, that aims to simplify the development of embodied applications without hindering the creativity of developers and designers. It wraps the computation of MediaPipe's landmark data as well as the processing of this data into a set of customizable interactions and gestures as a service so that applications can focus on creating immersive experiences efficiently and smoothly.

CHAPTER 3

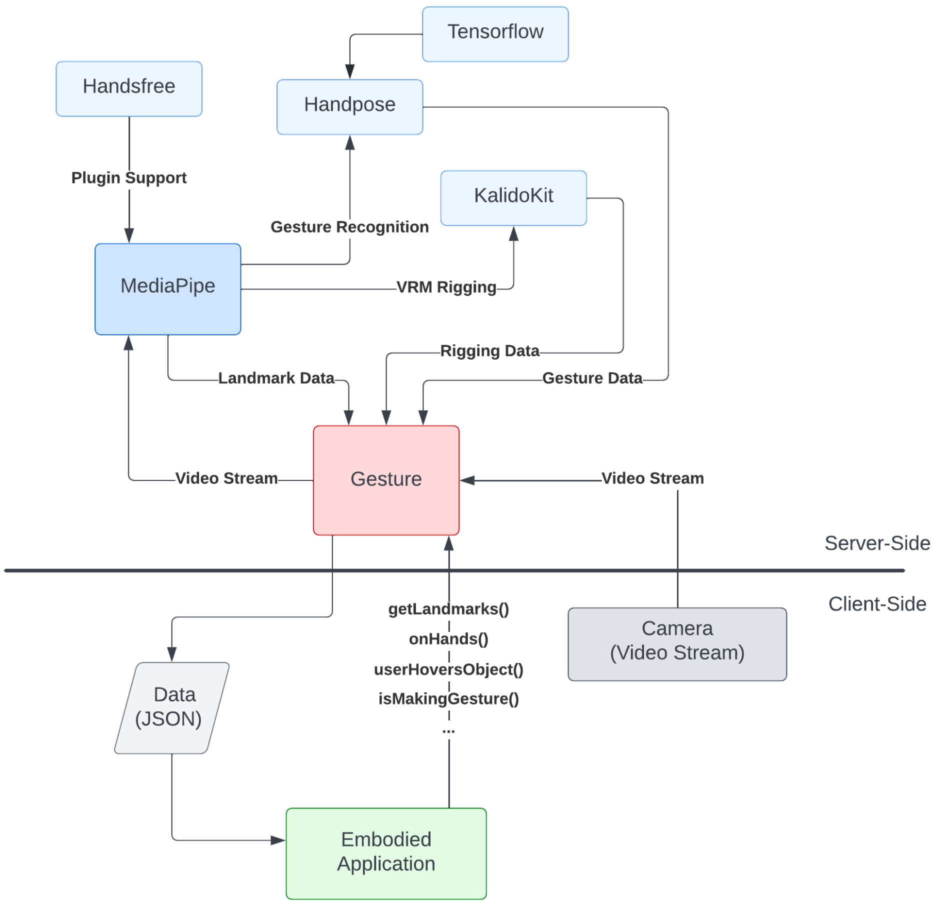
FRAMEWORK ARCHITECTURE AND DESIGN

System Architecture

Gesture.js takes in data from MediaPipe and additional body detection or gesture recognition frameworks and a video stream all in conjunction with each other to interpret and package them as customizable semantic functions for an embodied application to use.

Figure 2

Gesture.js Data Flow and Entry Points



As indicated in Figure 2, there are three main entry points of data. The first entry point is for the external frameworks and libraries that compute the necessary data for evaluation. While MediaPipe computes the raw landmark data, Fingerpose is a gesture classifier that will interpret these landmarks as an arrangement of finger positions that define a specific gesture. For example, the victory sign gesture is distilled to an uncurled index and middle finger pointing upwards and fully curled ring, pinky, and thumb fingers. The combination of finger curls and directions make up a gesture description, which is compared against the live input of landmark data. KalidoKit is a blendshape and kinematics solver specifically for MediaPipe's landmark data designed for rigging virtual reality models (VRM) and Live2D avatars. The second entry point is for the input of a camera's video stream. Typically, this camera is the webcam on a user's laptop or desktop computer, it can also be configured to be an external camera. This video stream is routed to MediaPipe either directly or with WebRTC. Lastly, the third entry point is the embodied application itself. Since Gesture.js is an interface for MediaPipe and interpretations of its landmark data, a collection of features are provided for an embodied application to call accordingly. These features can be distilled into four categories: Data Processing, Gesture Classification, Interaction Semantization, and Virtual Reality Model Rigging.

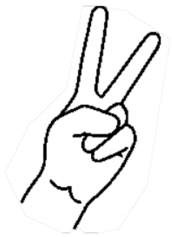
Data Processing. To ensure the highest amount of customization, the ability to compute landmark data within the application (not recommended) and access raw landmark data is provided. If the developer would like to use MediaPipe individually or in conjunction with other frameworks, the object-oriented implementation allows for the developer to do so. However, if the developer would rather use the CPU to compute graphics or other heavy processes instead of the external frameworks, then the API-oriented implementation allows for easy configuration to do so. The developer can simply install Gesture.js' npm package and run gesture to reveal a graphical user interface for configuration. This feature is a work in progress and a screenshot of this interface will be provided when completed.

Gesture Classification. Powered by Fingerpose, gesture recognition is as easy as calling the isMakingGesture() method in either the object-oriented or API-oriented implementations. A collection of predefined, static gestures is provided, so the developer will only need to refer to its name without any pre-training or classification beforehand. As exhibited in Table 1, a list of common gestures is delineated as a collection of finger curls and directions.

Table 1

Excerpt of Gestures Library

Gesture Name	Parameter Name	Estimate Type	Reference Image
Pinch	“pinch”	interactiveGestures	
Fist	“fist”	interactiveGestures	
Point	“point”	interactiveGestures	
One	“one”	numberGestures	
Two	“two”	numberGestures	

Three	“three”	numberGestures	
Four	“four”	numberGestures	
Five	“five”	numberGestures	
Thumbs Up	“thumbsUp”	miscellaneousGestures	
Thumbs Down	“thumbsDown”	miscellaneousGestures	
Victory Sign	“victorySign”	miscellaneousGestures	

Interaction Classification. Like the gesture classification feature, there is a collection of predefined interactions that allow the developer to build their experiences. This feature is available for both the object-oriented approach and the API-oriented approach. Each interaction can be customized with a series of callback functions and gesture variables. Detailed in Table 2, each interaction has its designated method name with specific callback requirements and optional gesture variables.

Table 2

Excerpt of Interactions Library

Interaction Name	Parameters	Gesture Variable(s)
onFace	callback : function	none
onPose	callback : function	none
onRightHand	callback : function	none
onLeftHand	callback : function	none
userHoversObject	landmarkPoints : array of int, object : HTML element, successCallback : function, failedCallback : function	none
userClicksObject	object : HTML element, successCallback : function	clickGesture (Default: pinch)
userDragsAndDrops	pickUpCallback : function, dragCallback : function, dropCallback : function	prePickUpGesture (Default: open hand) pickUpGesture (Default: fist)
isMakingGesture	gestureName : String	none

Virtual Reality Model Rigging. With the support of KalidoKit, the ability to rig a 3D model based on real-time body and face landmark points is available in the object-oriented implementation. Since KalidoKit is based on the animated 3D computer graphics application programming interface, Three.js, only WebGL-capable applications can utilize its rigging solutions. Moreover, KalidoKit only supports humanistic models as it only provides solvers for pose, face, and hands. This feature is designed for developing Three.js scenes with virtual reality human-like models with real-time body and face tracking.

CHAPTER 4

EXAMPLE APPLICATIONS AND ANALYSIS

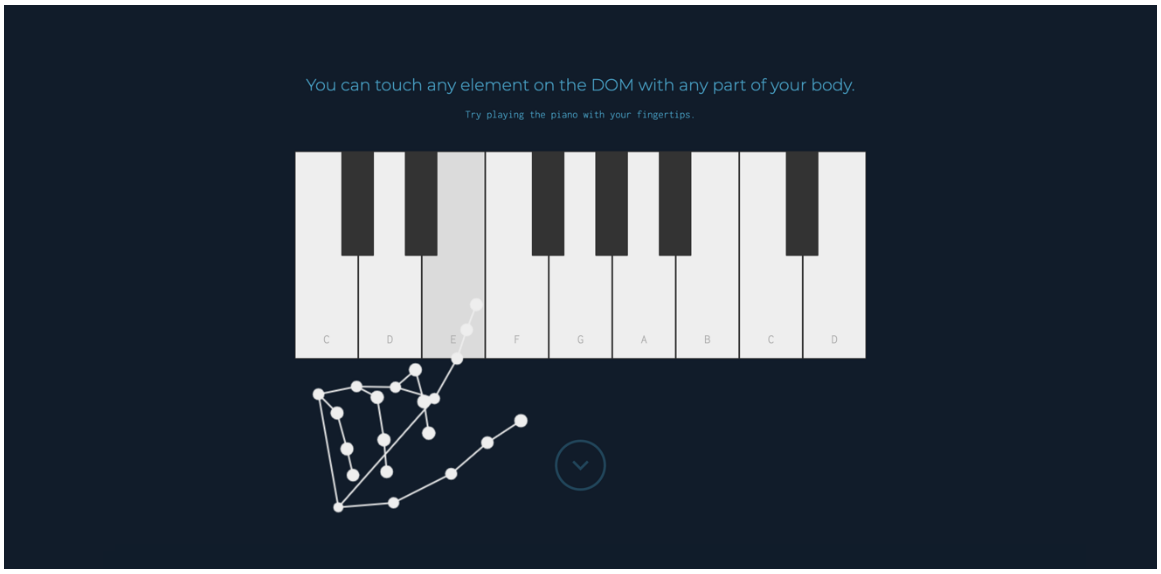
To demonstrate the ease of setting up Gesture.js and using its features, a variety of example applications were developed. These examples can be accessed at the following link: https://masters-thesis-project.herokuapp.com/example. This paper will present them and discuss their potential impacts.

Interacting with the DOM: Improving Accessibility on the Web

Creating website applications that can be interacted with using gestures and movement is the foundation of Gesture.js. This example application demonstrates how to use Gesture.js' gesture classification library and interaction classification methods to create simple, yet powerful experiences. Being able to touch an element on a website is the first step. By instantiating the Gesture class, the developer can pass in a callback function when a user "touches" an element in the Document Object Model (DOM). In this example, as shown in Figure 3, a piano is composed of div elements, and each element is passed into the userHoversObject() method. More complex methods such as userDragsAndDropsObject(), demonstrated in Figure 4, require more callbacks but are just as easy to implement onto a div element.

Figure 3

Demonstrating Hover or Touch with a Piano

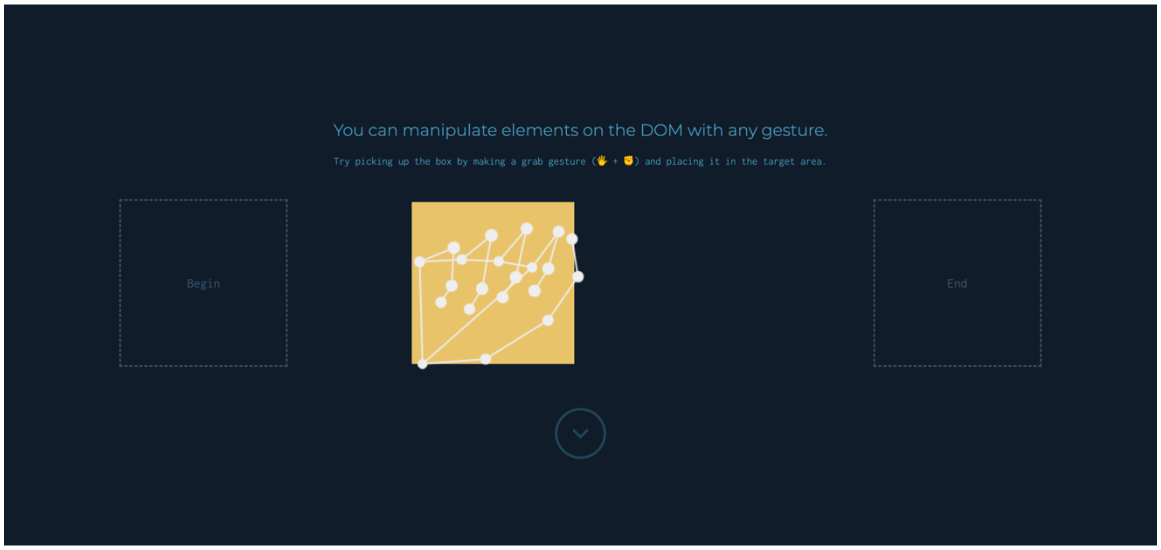


You can touch any element on the DOM with any part of your body.

Try playing the piano with your fingertips.

Figure 4

Demonstrating Combination of Hover and Gesture with Drag and Drop



You can manipulate elements on the DOM with any gesture.

Try picking up the box by making a grab gesture (🖐 + ✊) and placing it in the target area.

Three Game Environments: Analyzing the Performance of Gesture.js

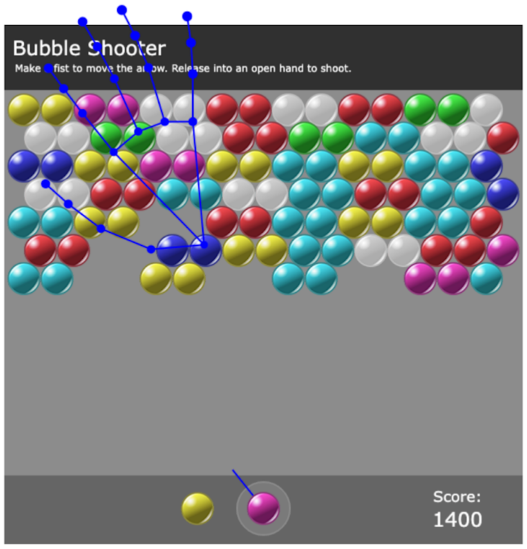
While the performance of Gesture.js on each game environment is visually apparent, a brief data collection process on each embodied experience was performed. The following performance metrics recorded are times of loading events, range of frame per second (FPS), CPU usage, and GPU usage. Lag, which is the response time from the computer, is computed based on the Animation Frame Fired activity. This activity is performed every time the application calls requestAnimationFrame() to redraw the game environment, and so lag is understood based on the total time an Animation Frame Fired activity takes to complete. Since the duration of each Animation Frame Fired activity varies, the total time of all activities, as well as the range, is provided. The performance metrics and durations of Animation Frame Fired activities were collected using Google's Developer Tools' Performance tab and computed accordingly. The bandwidth, or rate of data transfer from Gesture.js to an application, was also computed based on the network metrics found in the Developer Tools' network tab. These metrics are the number of requests, amount of data transferred, and total time to complete, and the total time spend loading. If the application is using Gesture.js as a service, then additional details about the time taken to complete a request related to landmark computation or interpretation are provided.

Bubble Shooter Performance Analysis

The first game environment, Bubble Shooter, demonstrates how to map landmark points to an HTML document and use gestures as a form of input. In this example, a simple bubble shooter game consisting of HTML elements for the bubbles and gun is controlled by a user's hand location and gestures. When a user closes their first, the angle of the gun can be aimed with the user's hand position, and when the user opens their hand, the gun shoots a bubble into the rows of bubbles. This HTML and Javascript code was taken from an open-source example written by rembound.com and a Gesture.js layer was applied that effectively replaced the mouse cursor position with hand landmark tracking (Rembound, 2015).

Figure 5

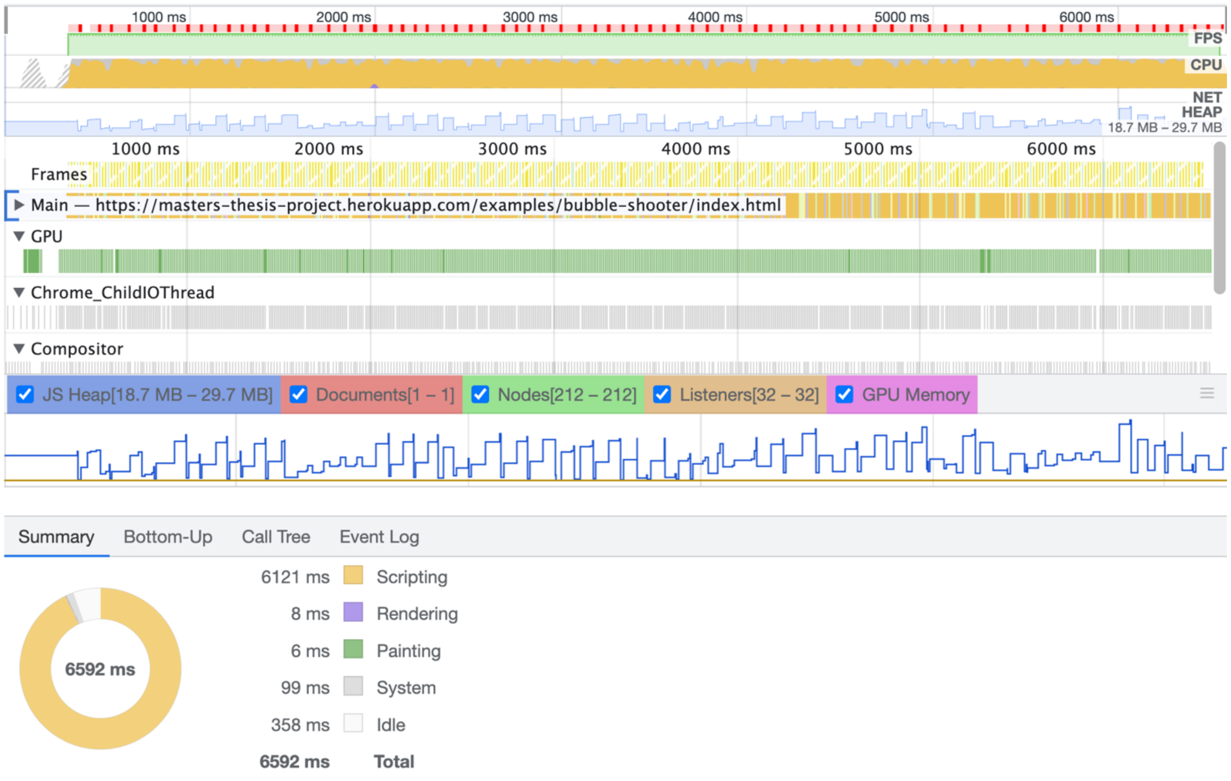
Bubble Shooter Graphical User Interface


Bubble Shooter
Make a fist to move the arrow. Release into an open hand to shoot.
Score:
1400

As shown in Figure 6, even though the game environment consists of HTML elements that are relatively low impact, most of the CPU is being used by MediaPipe for landmark interpretation and Tensorflow due to Fingerpose.js' gesture recognition. In the Bubble Shooter application, in the activities which are Request Animation Fired, a large majority of the activity duration is the computation of landmarks, more specifically the running MediaPipe's Holistic model. Detailed in Figure 7, about 91.1% of the time is spent running MediaiPipe's landmark computation, 5.9% is spent using Fingerpose.js' gesture recognition, and 0.2% is spent running Gesture.js' landmark interpretation. Lastly, the network performance, or the bandwidth, is shown in Figure 8. Since the Bubble Shooter game environment is running Gesture.js itself, it does not request or receive any landmark data over a network. Most of the load time occurs at the beginning, where it loads the appropriate models, libraries, and files.

Figure 6

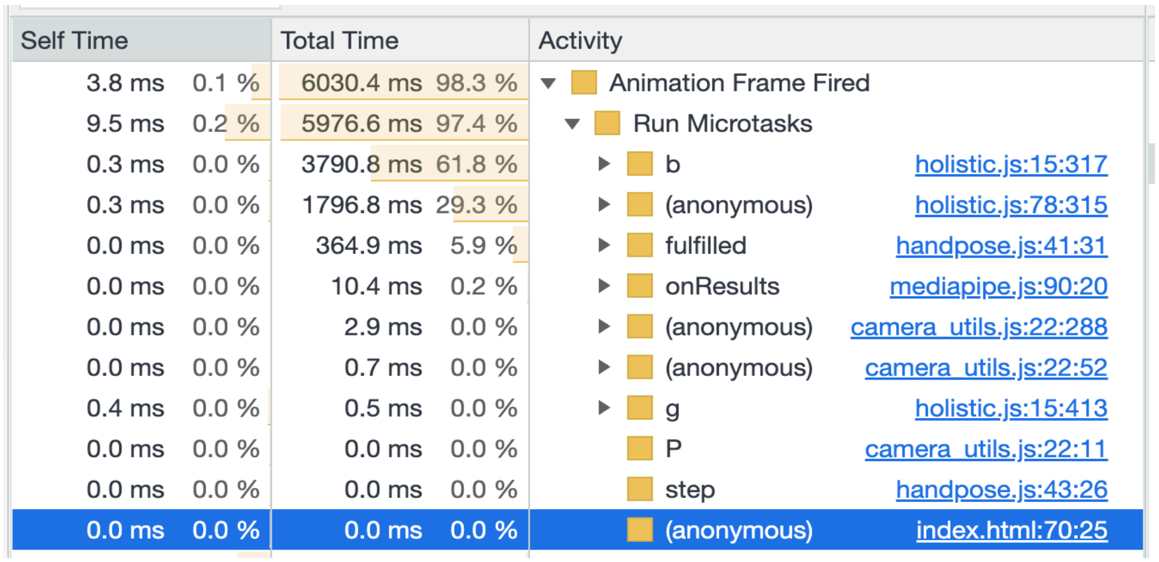
Developer Tools' Performance Profile for Bubble Shooter



6121 ms	🟨	Scripting
8 ms	🟪	Rendering
6 ms	🟩	Painting
99 ms	⬜	System
358 ms	⬜	Idle
6592 ms		**Total**

Figure 7

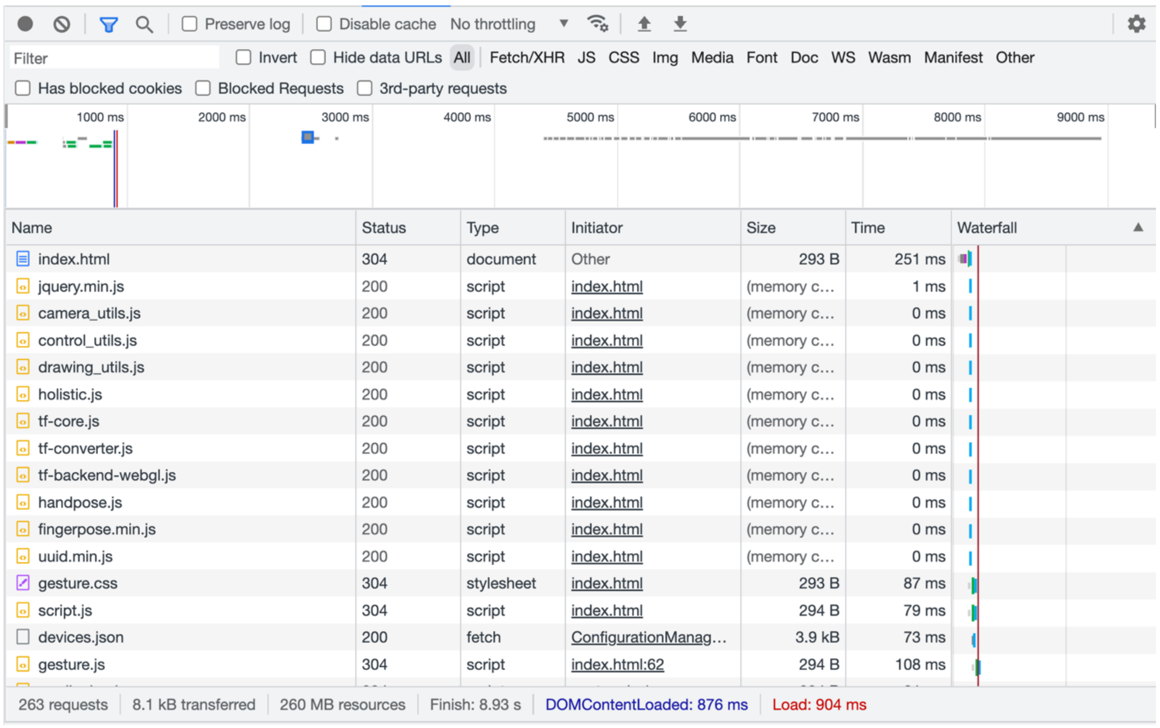
Bubble Shooter's Animation Frame Fired Composition



Self Time		Total Time		Activity	
3.8 ms	0.1 %	6030.4 ms	98.3 %	▼ Animation Frame Fired	
9.5 ms	0.2 %	5976.6 ms	97.4 %	▼ Run Microtasks	
0.3 ms	0.0 %	3790.8 ms	61.8 %	▶ b	holistic.js:15:317
0.3 ms	0.0 %	1796.8 ms	29.3 %	▶ (anonymous)	holistic.js:78:315
0.0 ms	0.0 %	364.9 ms	5.9 %	▶ fulfilled	handpose.js:41:31
0.0 ms	0.0 %	10.4 ms	0.2 %	▶ onResults	mediapipe.js:90:20
0.0 ms	0.0 %	2.9 ms	0.0 %	▶ (anonymous)	camera_utils.js:22:288
0.0 ms	0.0 %	0.7 ms	0.0 %	▶ (anonymous)	camera_utils.js:22:52
0.4 ms	0.0 %	0.5 ms	0.0 %	▶ g	holistic.js:15:413
0.0 ms	0.0 %	0.0 ms	0.0 %	P	camera_utils.js:22:11
0.0 ms	0.0 %	0.0 ms	0.0 %	step	handpose.js:43:26
0.0 ms	0.0 %	0.0 ms	0.0 %	(anonymous)	index.html:70:25

Figure 8

Developer Tools' Network Metrics for Bubble Shooter



Name	Status	Type	Initiator	Size	Time	Waterfall
index.html	304	document	Other	293 B	251 ms	
jquery.min.js	200	script	index.html	(memory c…	1 ms	
camera_utils.js	200	script	index.html	(memory c…	0 ms	
control_utils.js	200	script	index.html	(memory c…	0 ms	
drawing_utils.js	200	script	index.html	(memory c…	0 ms	
holistic.js	200	script	index.html	(memory c…	0 ms	
tf-core.js	200	script	index.html	(memory c…	0 ms	
tf-converter.js	200	script	index.html	(memory c…	0 ms	
tf-backend-webgl.js	200	script	index.html	(memory c…	0 ms	
handpose.js	200	script	index.html	(memory c…	0 ms	
fingerpose.min.js	200	script	index.html	(memory c…	0 ms	
uuid.min.js	200	script	index.html	(memory c…	0 ms	
gesture.css	304	stylesheet	index.html	293 B	87 ms	
script.js	304	script	index.html	294 B	79 ms	
devices.json	200	fetch	ConfigurationManag…	3.9 kB	73 ms	
gesture.js	304	script	index.html:62	294 B	108 ms	

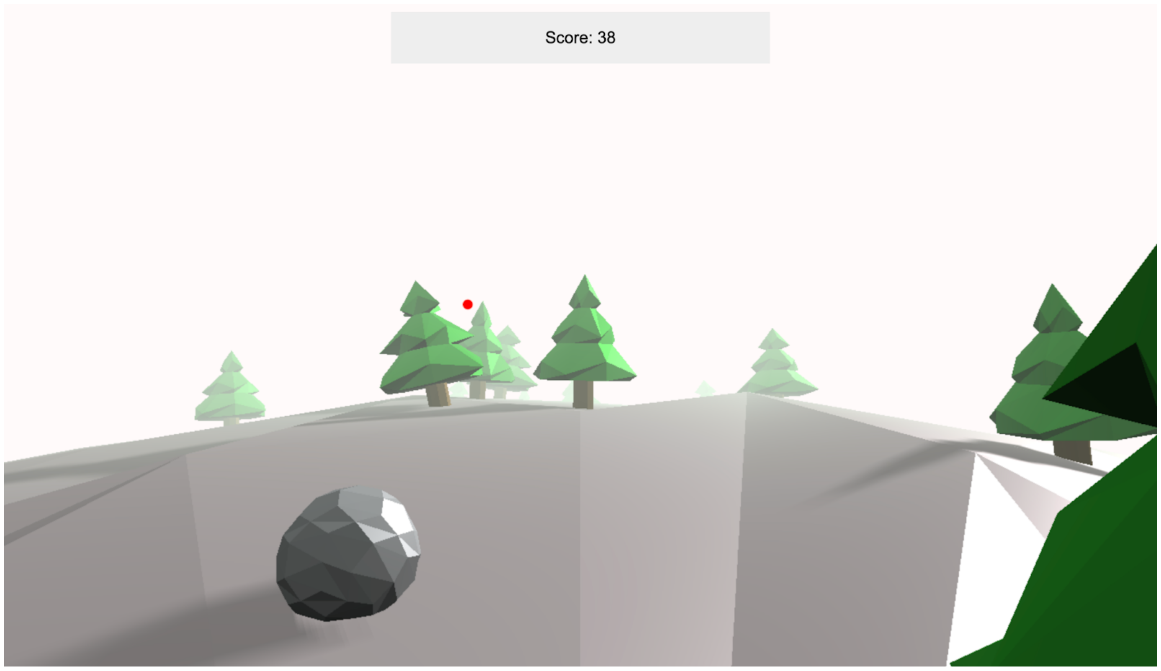
263 requests | 8.1 kB transferred | 260 MB resources | Finish: 8.93 s | DOMContentLoaded: 876 ms | Load: 904 ms

Infinite Runner Performance Analysis

The next game environment, exhibited by the Infinite Runner example application, demonstrates how to map landmark points to a 3D space. Three.js, a computer graphics application programming interface, is used to render a WebGL-based 3D scene that animates a hero object. In this example, a simple infinite runner game consisting of a stone ball rolling through an obstacle course of random dynamically-rendered trees are adapted for a user to control via their nose. The Three.js code was taken from an open-source example written by Juwal Bose and a Gesture.js layer was applied that effectively replaced the arrow key interactions with nose landmark tracking (Bose, 2017).

Figure 9

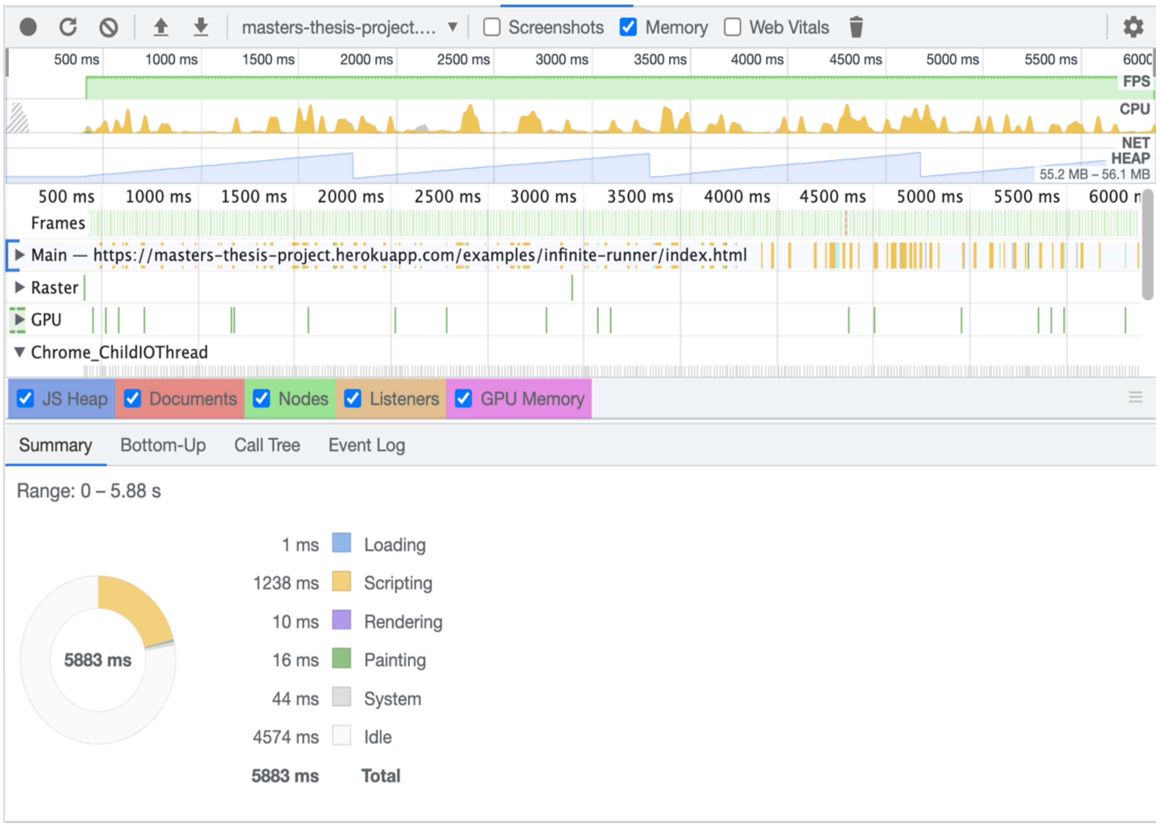
Infinite Runner Graphical User Interface



In the performance analysis of this game environment, it is best to first understand to CPU, GPU, and frame rate of the application without the Gesture.js layer. Therefore, in Figure 10, the performance profile of just the Three.js code is provided to show the relatively higher need for the CPU and GPU to render a 3D scene compared to an HTML-based scene. About 28% of the CPU is needed for scripting, which mostly consists of the request of animation frames to rerender the game environment. It was also found that due to faster Animation Frame Fired activities, the frame rate was about 59.7 FPS. Once the Gesture.js layer is added, the CPU and GPU must now focus on handling the computation and interpretation of landmarks—both handled in the requests of animation frames—as well as the Three.js environment. Depicted in Figure 11, as soon as the application changes to an embodied application, the CPU usage jumps from 28% to 96%. This change is largely due to the computation of landmarks on MediaPipe's end, which is shown in Figure 12 to take about 80% of an Animation Frame Fired activity. This led to an average frame rate of 25 FPS, which is visually apparent to the user. Similar to Bubble Shooter, the application is experiencing this lag because it runs Gesture.js itself instead of using it as a service. This is also the cause of why the network performance is similar to Bubble Shooter.

Figure 10

Developer Tools' Performance Profile for Infinite Runner (without Gesture.js)

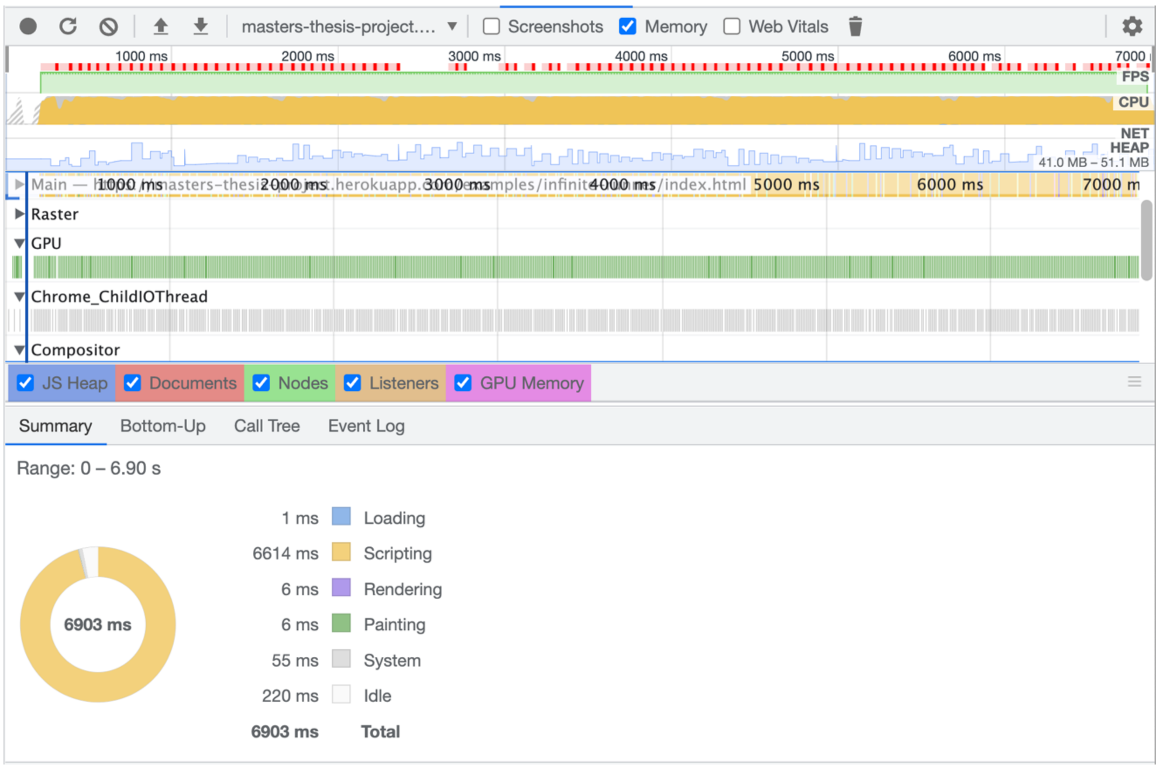


Range: 0 – 5.88 s

1 ms	🟦	Loading
1238 ms	🟨	Scripting
10 ms	🟪	Rendering
16 ms	🟩	Painting
44 ms	⬜	System
4574 ms	⬜	Idle
5883 ms		**Total**

Figure 11

Developer Tools' Performance Profile for Infinite Runner (with Gesture.js)

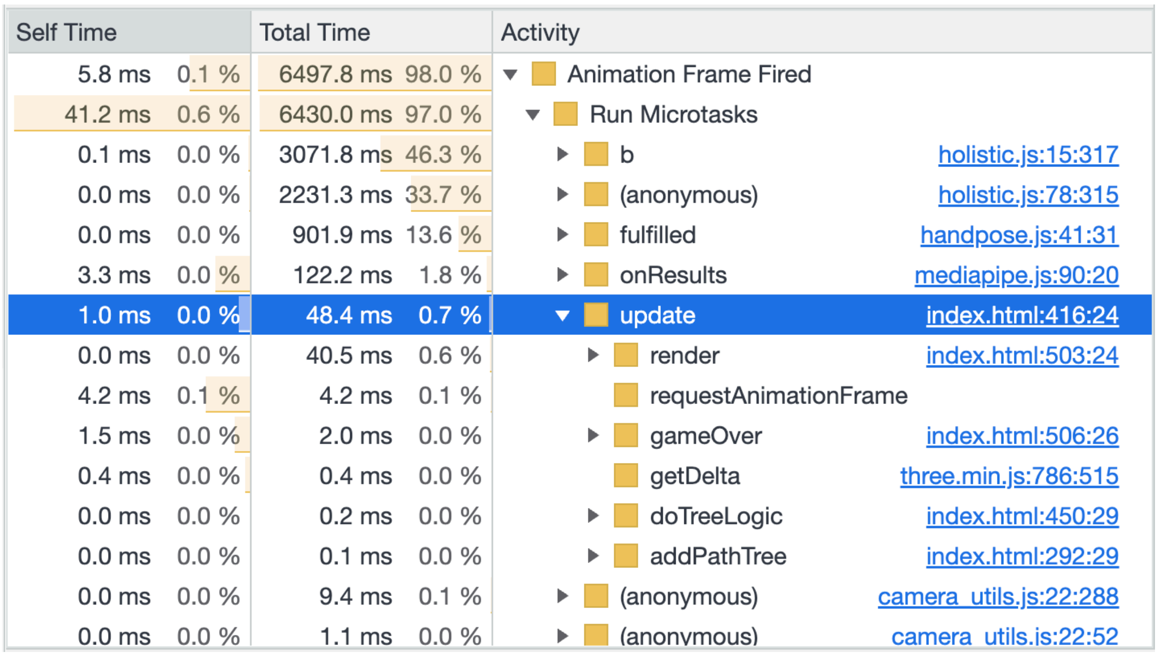


Range: 0 – 6.90 s

1 ms	🟦	Loading
6614 ms	🟨	Scripting
6 ms	🟪	Rendering
6 ms	🟩	Painting
55 ms	⬜	System
220 ms	⬜	Idle
6903 ms		**Total**

Figure 12

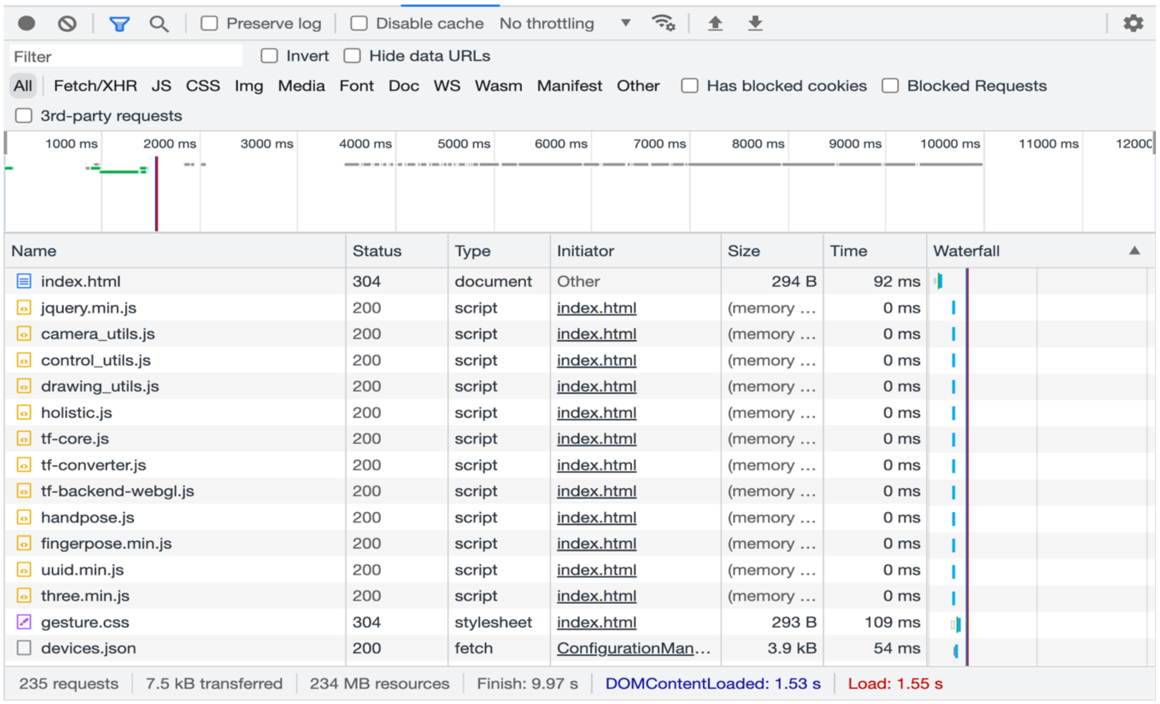
Infinite Runner's Animation Frame Fired Composition



Self Time		Total Time		Activity	
5.8 ms	0.1 %	6497.8 ms	98.0 %	Animation Frame Fired	
41.2 ms	0.6 %	6430.0 ms	97.0 %	Run Microtasks	
0.1 ms	0.0 %	3071.8 ms	46.3 %	b	holistic.js:15:317
0.0 ms	0.0 %	2231.3 ms	33.7 %	(anonymous)	holistic.js:78:315
0.0 ms	0.0 %	901.9 ms	13.6 %	fulfilled	handpose.js:41:31
3.3 ms	0.0 %	122.2 ms	1.8 %	onResults	mediapipe.js:90:20
1.0 ms	0.0 %	48.4 ms	0.7 %	update	index.html:416:24
0.0 ms	0.0 %	40.5 ms	0.6 %	render	index.html:503:24
4.2 ms	0.1 %	4.2 ms	0.1 %	requestAnimationFrame	
1.5 ms	0.0 %	2.0 ms	0.0 %	gameOver	index.html:506:26
0.4 ms	0.0 %	0.4 ms	0.0 %	getDelta	three.min.js:786:515
0.0 ms	0.0 %	0.2 ms	0.0 %	doTreeLogic	index.html:450:29
0.0 ms	0.0 %	0.1 ms	0.0 %	addPathTree	index.html:292:29
0.0 ms	0.0 %	9.4 ms	0.1 %	(anonymous)	camera_utils.js:22:288
0.0 ms	0.0 %	1.1 ms	0.0 %	(anonymous)	camera_utils.js:22:52

Figure 13

Developer Tools' Network Metrics for Infinite Runner



Name	Status	Type	Initiator	Size	Time	Waterfall
index.html	304	document	Other	294 B	92 ms	
jquery.min.js	200	script	index.html	(memory …	0 ms	
camera_utils.js	200	script	index.html	(memory …	0 ms	
control_utils.js	200	script	index.html	(memory …	0 ms	
drawing_utils.js	200	script	index.html	(memory …	0 ms	
holistic.js	200	script	index.html	(memory …	0 ms	
tf-core.js	200	script	index.html	(memory …	0 ms	
tf-converter.js	200	script	index.html	(memory …	0 ms	
tf-backend-webgl.js	200	script	index.html	(memory …	0 ms	
handpose.js	200	script	index.html	(memory …	0 ms	
fingerpose.min.js	200	script	index.html	(memory …	0 ms	
uuid.min.js	200	script	index.html	(memory …	0 ms	
three.min.js	200	script	index.html	(memory …	0 ms	
gesture.css	304	stylesheet	index.html	293 B	109 ms	
devices.json	200	fetch	ConfigurationMan…	3.9 kB	54 ms	

235 requests | 7.5 kB transferred | 234 MB resources | Finish: 9.97 s | DOMContentLoaded: 1.53 s | Load: 1.55 s

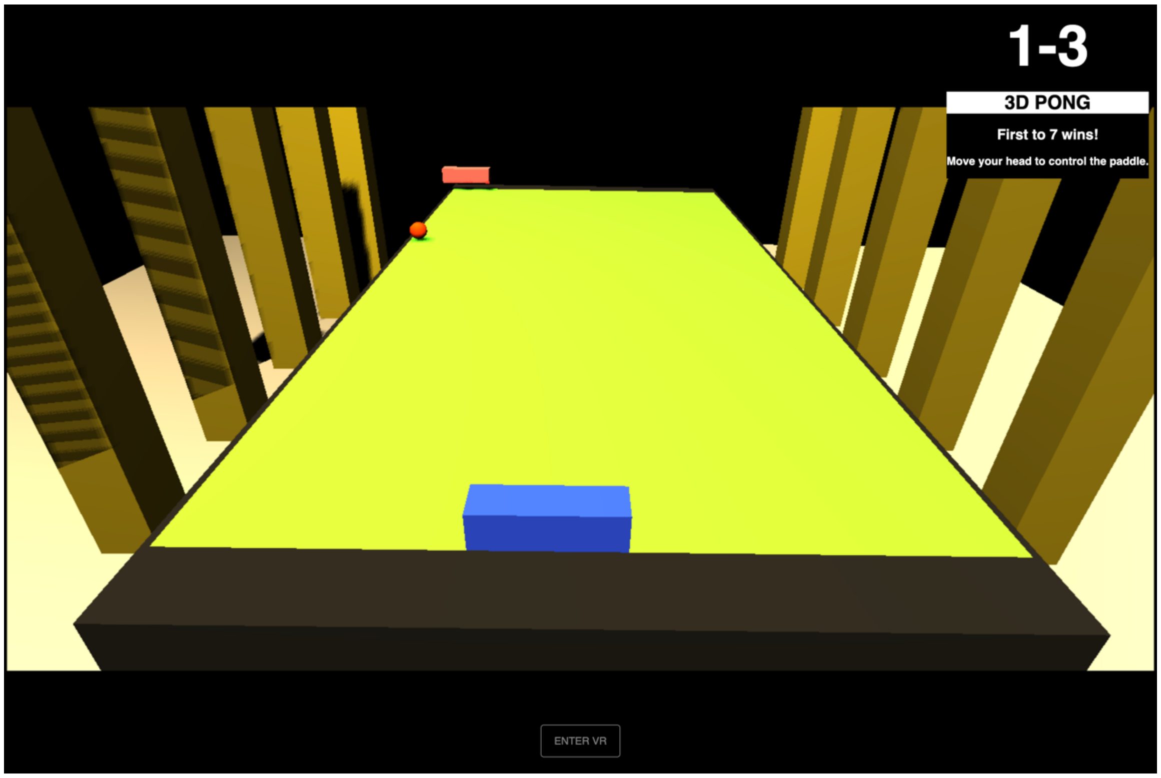
Embodied Pong Performance Analysis

To reap the benefits of Gesture.js, it must be used as a service. Moreover, the computation and interpretation of the landmark data need to be computed by something other than the embodied application itself. In this next game environment, Embodied Pong demonstrates how to call Gesture.js as a service, particularly with WebSockets, and map results to a point in 3D space. Similar to Infinite Runner, the game environment consists of a 3D scene made by Three.js and takes a single landmark point, a user's nose, to move an object in this scene. In this example, the object is a paddle and the scene is a pong board with another paddle that is controlled by a simple AI. This Three.js code was taken from an open-source example written by Nikhil Suresh and a Gesture.js layer was applied that effectively replaced the A and D keys with nose landmark tracking (Suresh, 2013).

This example is slightly different from the last two previous game environments, as two instances need to send and receive landmark data. An instance can be a client or a server, but there needs to be at least one instance in charge of computing the landmark data. In the Bubble Shooter and Infinite Runner examples, the embodied application served as an instance, but since there was only one instance, it was the one that was in charge of computing the landmarks. In this example, you have presented the instance in charge of computation, and it will automatically generate a link to the appropriate instance that will serve to be the embodied application. The embodied application, the 3D pong game, is essentially the client for the first page and will request landmark data from it to correctly move the paddle according to the user's node position. For both instances to communicate, both links must be open. This can be done by opening both links in two different windows (not tabs) on the same computer or a different one. For example, the generated link can be opened in a mobile browser and it will still work. If the user opens the generated link on a WebXR-compatible phone, the user can view the embodied application in VR mode, attach it to a headset such as Google Cardboard, and enter the 3D space from the perspective of the paddle. This is key to Gesture.js, as now web applications gain a new dimension of interaction that allows higher immersion and interactivity.

Figure 14

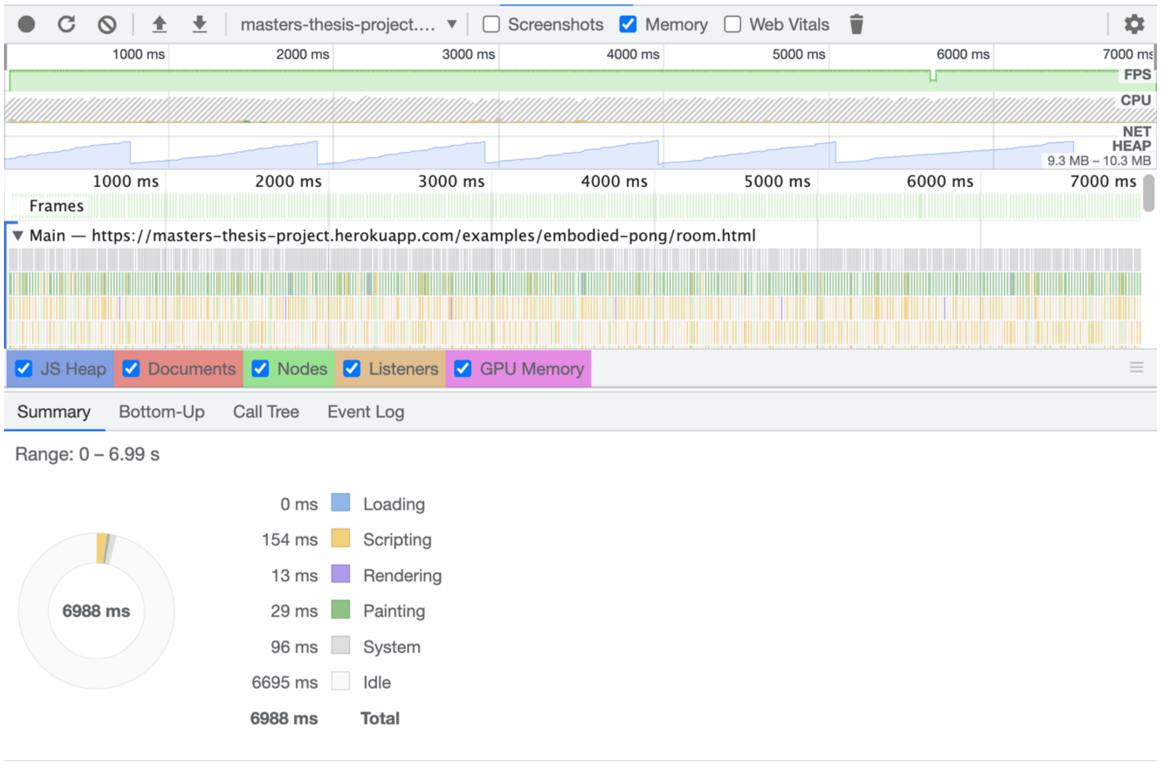
Embodied Pong Graphical User Interface



As shown in Figure 15, both the CPU and GPU usage for the embodied application is characteristically lower than those of the previous two examples. This is because all of the work to compute the landmarks is done by another instance, the “server” page. In Figure 16, this is exhibited as the performance profile is notably similar to the performance profiles of Bubble Shooter and Infinite Runner. Moreover, the Animation Frame Fired activity only contains the work of Three.js’ rendering of the 3D environment and objects. Instead of getting the landmarks directly from MediaPipe within this Animation Frame Fired activity, it utilizes WebSockets to call the “server” instance for landmark data. This is exhibited in Figure 18, where the network performance shows the request and response timings of one request of landmark data. On an internet connection with a download speed of 13.70 Mbps and upload speed of 6.86 Mbps, the amount of time a request and response would take is about 273.86 ms. This method of retrieving landmark data from another instance is considerably better than the amount of time it was taking to compute them itself within the Animation Frame Fired activities in the previous two examples.

Figure 15

Developer Tools' Performance Profile for Embodied Pong (Client)



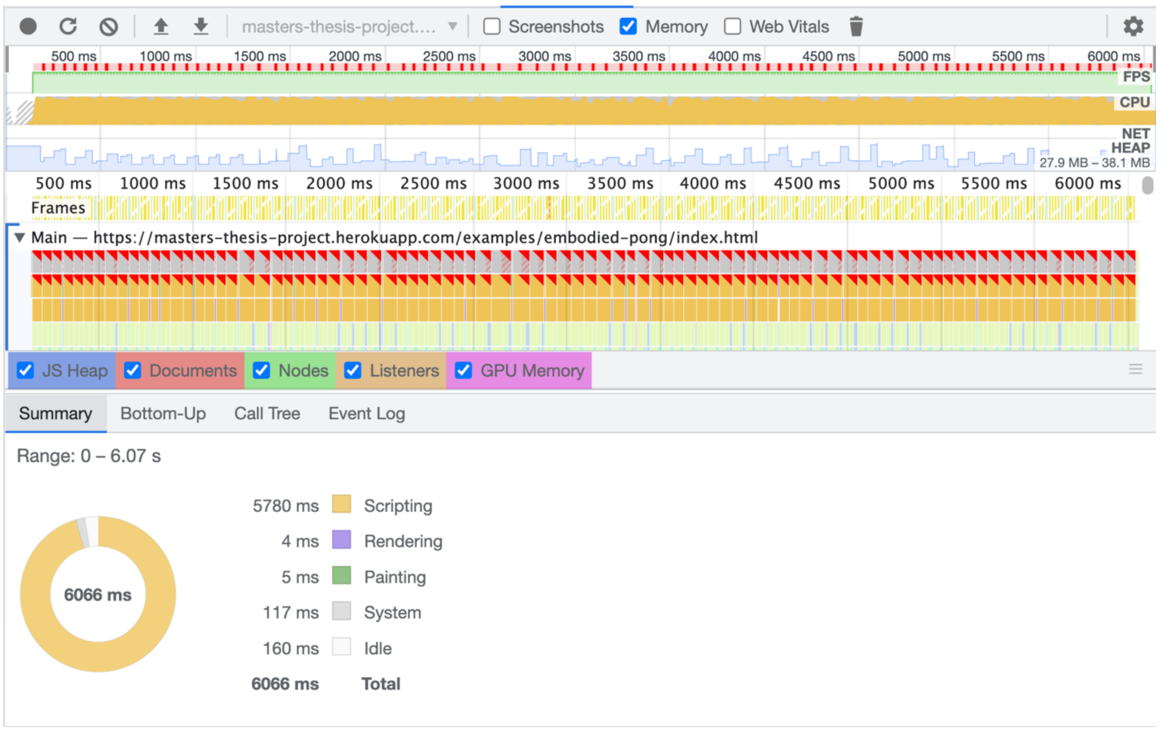
Main — https://masters-thesis-project.herokuapp.com/examples/embodied-pong/room.html

Range: 0 – 6.99 s

0 ms	🟦	Loading
154 ms	🟨	Scripting
13 ms	🟪	Rendering
29 ms	🟩	Painting
96 ms	⬜	System
6695 ms	⬜	Idle
6988 ms		**Total**

Figure 16

Developer Tools' Performance Profile for Embodied Pong (Server)



Main — https://masters-thesis-project.herokuapp.com/examples/embodied-pong/index.html

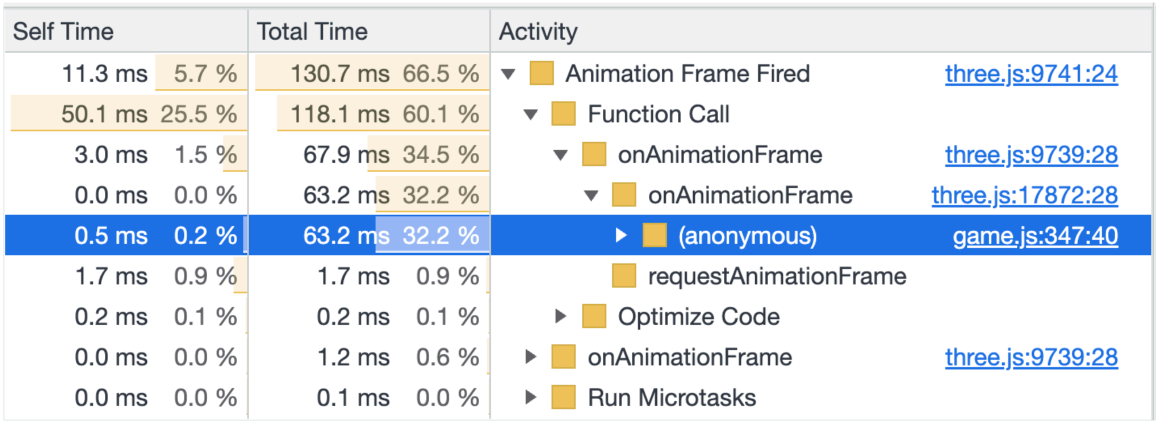
Range: 0 – 6.07 s

6066 ms

5780 ms	🟨	Scripting
4 ms	🟪	Rendering
5 ms	🟩	Painting
117 ms	⬜	System
160 ms	⬜	Idle
6066 ms		**Total**

Figure 17

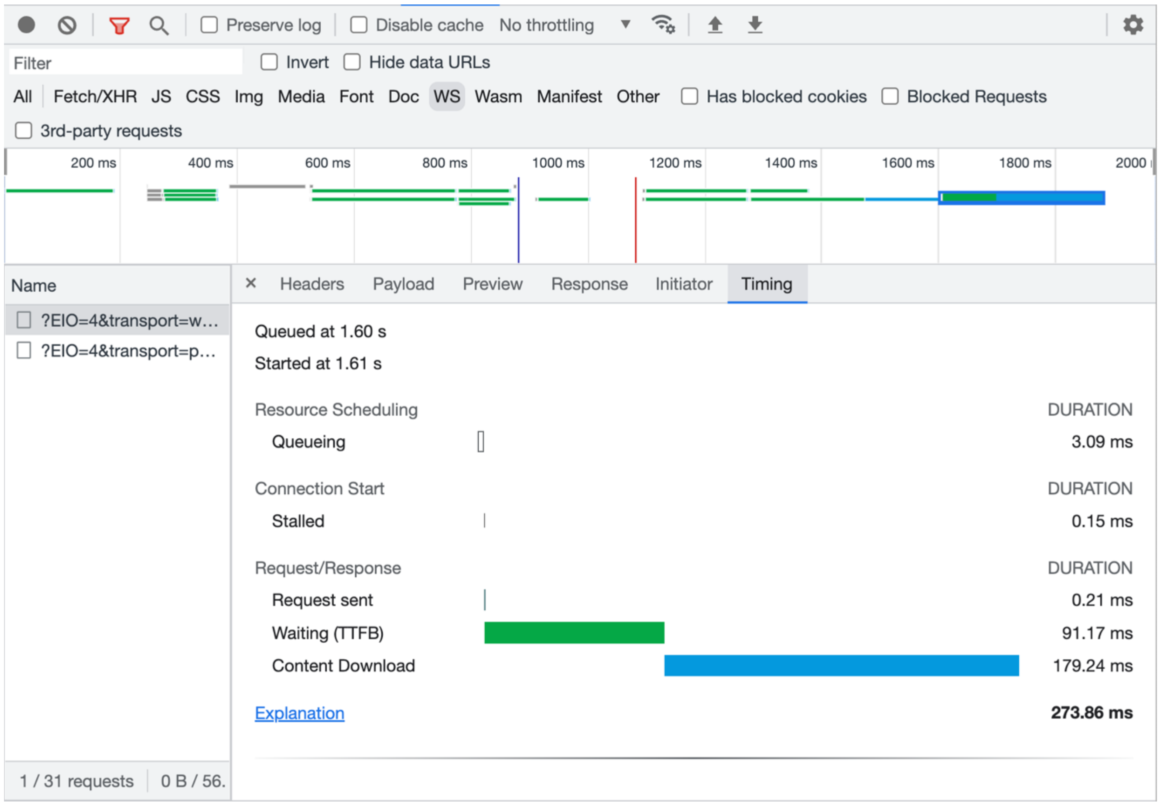
Embodied Pong's Animation Frame Fired Composition



Self Time		Total Time		Activity	
11.3 ms	5.7 %	130.7 ms	66.5 %	Animation Frame Fired	three.js:9741:24
50.1 ms	25.5 %	118.1 ms	60.1 %	Function Call	
3.0 ms	1.5 %	67.9 ms	34.5 %	onAnimationFrame	three.js:9739:28
0.0 ms	0.0 %	63.2 ms	32.2 %	onAnimationFrame	three.js:17872:28
0.5 ms	0.2 %	63.2 ms	32.2 %	(anonymous)	game.js:347:40
1.7 ms	0.9 %	1.7 ms	0.9 %	requestAnimationFrame	
0.2 ms	0.1 %	0.2 ms	0.1 %	Optimize Code	
0.0 ms	0.0 %	1.2 ms	0.6 %	onAnimationFrame	three.js:9739:28
0.0 ms	0.0 %	0.1 ms	0.0 %	Run Microtasks	

Figure 18

Developer Tools' Network Metrics for Embodied Pong (with Request Details)



?EIO=4&transport=w...
?EIO=4&transport=p...

Queued at 1.60 s
Started at 1.61 s

Resource Scheduling	DURATION
Queueing	3.09 ms
Connection Start	DURATION
Stalled	0.15 ms
Request/Response	DURATION
Request sent	0.21 ms
Waiting (TTFB)	91.17 ms
Content Download	179.24 ms
Explanation	273.86 ms

1 / 31 requests 0 B / 56.

Ideally, this other instance would not need to serve an HTML page and should just run on an actual server, local or on the cloud. Gesture.js supports this as it is also a published node package manager (npm) module that can be installed and used as a Node.js module for server-side computing. The module is currently public under the name @ahfowler/gesture and can be found here: https://www.npmjs.com/package/@ahfowler/gesture.

Performance, Lag, and Network Results

Building embodied applications are indeed CPU-intensive. Even with the addition of MediaPipe's pre-trained models, the consistent computation of these landmarks requires more than 80% of the CPU which is problematic for applications requiring addition computation itself for other processes, such as rendering a complex game environment. As shown in Table 3 and Table 4, the first two applications, Bubble Shooter and Infinite Runner, both attempted to run

Gesture.js on the same instance as the embodied application. This drastically affected the rate of frames per second, dropping it from 60 FPS to around 20 FPS. It also required more than 80% of the GPU and more than 90% of the CPU to simply maintain the constant computation of landmark data.

Table 3

Performance Metrics of Game Environments

Game Environment Name	Loading Events	Range of Frames per Second (FPS)	CPU Usage	GPU Usage
Bubble Shooter	**DOMContentLoaded:** 1969.8 ms **onLoadEvent:** 1992.2 ms **FirstPaint:** 4107.8 ms **FirstContentfulPaint**: 4107.8 ms	20.1 – 22.2	**Scripting**: 92% **Rendering**: 0.12% **Painting**: 0.09% **System**: 1.50% **Idle**: 5.43%	**GPU**: 81% **Idle**: 18%
Infinite Runner	**DOMContentLoaded:** 987.6 ms **onLoadEvent:** 1028.6 ms **FirstPaint:** 974.8 ms **FirstContentfulPaint**: 974.8 ms	24 – 28.6	**Scripting**: 95% **Rendering**: 0.08% **Painting**: 0.08% **System**: 0.79% **Idle**: 3.19%	**GPU**: 83.38% **Idle**: 16.62%
Embodied Pong	**DOMContentLoaded:** 817.5 ms **onLoadEvent:** 1026.5 ms **FirstPaint:** 491.6 ms **FirstContentfulPaint**: 491.6 ms	59.4 – 58.8	**Scripting**: 2.2% **Rendering**: 0.19% **Painting**: 0.41% **System**: 1.37% **Idle**: 95.81%	**GPU**: 86.19% **Idle**: 13.81%

While CPU usage is highly determined for the computer running the embodied application, it is also important to note the amount of time it takes to complete an animation frame. The lag of a website is highly dependent on the animation frames it attempts to render at an as fast, consistent rate. In the first two examples, the Animation Frame Fired duration was considerably larger than the last example, mainly because in this activity, it is computing and interpreting landmarks as well as rendering the game environment with or without these landmarks. As depicted in Table 4, around 80% of the animation activity is spent computing landmarks, while less a 1% of it is being spent rendering the game environment. This results in the lag user experiences as they move with the application.

Table 4

Lag Computation of Game Environments

Game Environment Name	Range of Animation Frame Fired Duration	Total Animation Frame Fired Duration	Landmark Computation Usage	Landmark Interpretation Usage	Game Environment Usage
Bubble Shooter	0.3 ms – 159.8 ms	6030.4 ms	91.1%	6.1% (0.2% Gesture.js)	0.00%
Infinite Runner	0.4 ms – 101.4 ms	6497.8 ms	80%	15.4% (1.8% Gesture.js)	0.7%
Embodied Pong	0.3 ms – 159.8 ms	6030.4 ms	91.1%	6.1% (0.2% Gesture.js)	0.00%

As discussed, the solution to eliminating lag is to decrease the time spent creating an animation frame. This can be done by taking the landmark computation off of the embodied application’s hands and put into another instance such as another application or a server. Demonstrated in Figure 18, the third example is distributed system that requests data from another instance, a “server” page, and uses this data to render the Three.js environment. The duration of a request and response consists of its wait time after the request and the response content download time. It is also good to note that the number of requests significantly decreases with the third application because the experience no longer needs to request MediaPipe's models for landmark computation or external libraries for landmark interpretation. Instead, it only focuses on the emitting event through WebSockets to an instance that will make those requests themselves.

Table 5

Network Metrics of Game Environments

Game Environment Name	Number of Requests	Amount of Data Transferred	Amount of Resources Used	Request Response Wait Time	Request Response Content Download Time
Bubble Shooter	263 Requests	8.1 kB	8.93 s	N/A	N/A
Infinite Runner	235 Requests	7.5 kB	9.97 s	N/A	N/A
Embodied Pong	31 Requests	56.8 kB	4.3 MB	91.17 ms	179.24 ms

Embodied Virtual Reality: Advancing WebXR and Three.js

As Gesture.js is powered by MediaPipe, all machine-learning models developed by MediaPipe are accessible. The Holistic Model is particularly powerful because it combines the Hands, Pose, and Face Mesh models. This tracking can be used for full-body movement tracking in virtual reality environments. This example application demonstrates this in conjunction with KalidoKit's rigging solutions to simulate a hardware-less, purely computer-vision-based virtual reality experience.

Figure 19

Rigged Characters in a Virtual Room Space

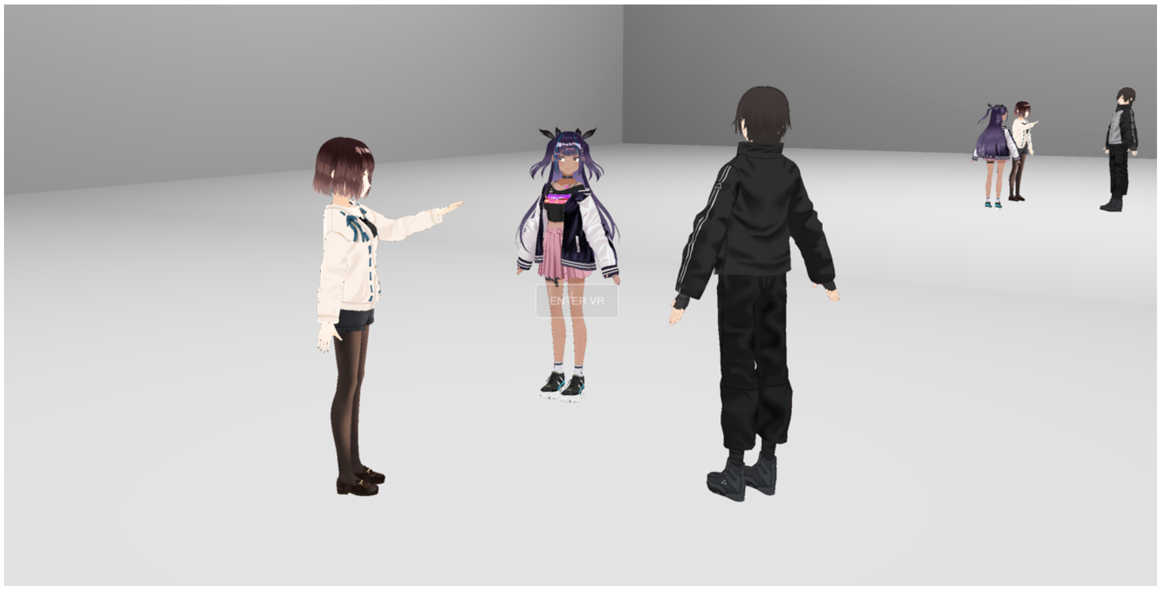
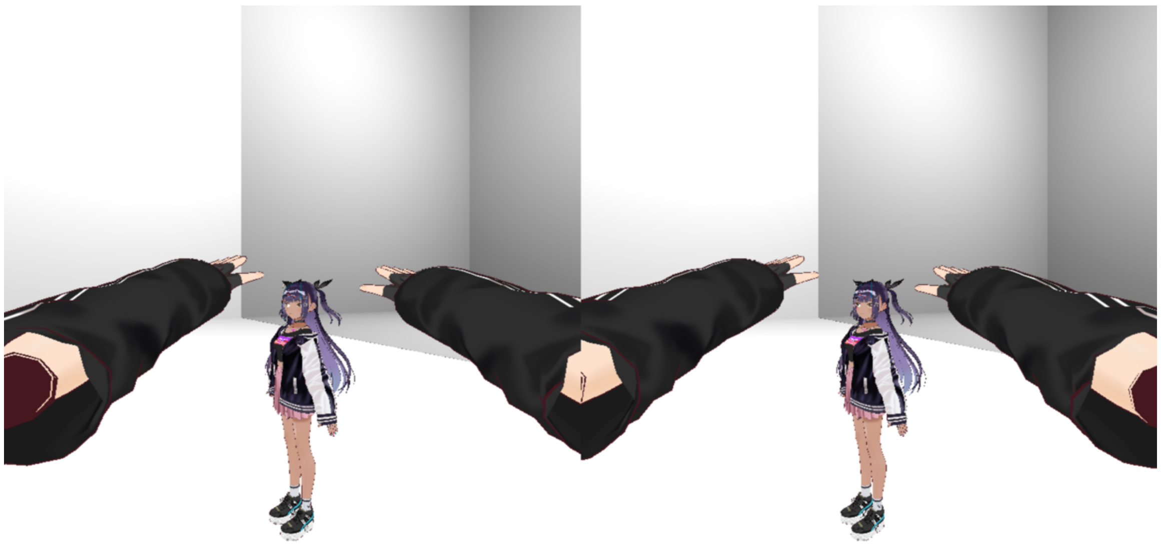


Figure 20

First-Person View in WebXR Mode



CHAPTER 5

CONCLUSION

Embodied applications are a game-changer as they provide a new way for people to interact with applications that transcend beyond mouse clicks and scrolls. Using body movements or gestures can allow users to use highly expressive interactions with software and therefore create interactive, immersive experiences. With the introduction of MediaPipe, an open-source collection of machine-learning solutions for body point detection, a gateway of embodied application development has resurfaced in the field of software engineering and human-computer interaction. However, there is a need for the standardization and abstraction of software development methodologies as well as a service that separates the computation and inference of landmark data into semantic gesture and interaction configuration. This paper has explored the ways to help developers achieve as well as introduced a framework, Gesture.js, that aims to fulfill these needs for software developers and designers so that they can focus on creating immersive experiences efficiently and smoothly.

Current Limitations

There are several limitations of Gesture.js that were discovered throughout the development process. Unfortunately, most of these problems stem from the limitations of MediaPipe. The good news is that while there is a cascading effect of limitations, this means that with the inevitable advancement of computer vision-based body part detection models and improvement of hardware like webcams and computing power of personal devices, Gesture.js and its relevant frameworks will consequently become better.

For example, KalidoKit uses landmark data from MediaPipe to calculate 3D model face, body, and hand positions. However, the rigging is severely limited by the low accuracy and tracking rate of MediaPipe's pose model. This leads to some technical difficulties with Gesture.js' embodied virtual reality example application, especially when a user backs up to move their legs.

Another aspect of Gesture.js is the lack of machine learning, such as a neural network-based, gesture recognition framework. Fingerpose is simply a similarity calculator, and so gesture recognition depends highly on the clarity of its gesture descriptions and the distinctiveness of the gestures for each other. For example, in the interactive example application, the click gesture (a pinch between an index finger and thumb) scores quite similarly to an open hand gesture because only two out of five fingers are not fitting the description of a pinch. Therefore, utilizing gestures in embodied applications is quite difficult, especially when certain gestures look even remotely like one another.

Future Research and User Studies

Aside from the anticipation of better computer vision-based body part detection models and improvement of computer hardware, the next steps of Gesture.js are to replace Fingerpose with a neural network-based gesture recognition model, provide the user the ability to define their gestures, and incorporate gestures that contain movement patterns or require more than one hand. Since the purpose of this thesis was to develop a framework that enables developers to create embodied applications easier, the ability to develop a gesture recognition machine-learning model is outside of this scope. However, in the future, Gesture.js hopes to integrate a pre-trained gesture recognition model that utilizes neural networks

Similarly, since the focus of Gesture.js is to improve the development experience of embodied applications, user experience studies are needed to effectively evaluate the qualitative experience of development. Moreover, it would also be beneficial to learn more about how one's body can effectively interact with an application. Since a user experience case study is outside of this scope, yet highly advantageous, a conceptual plan will be proposed for future research.

Using Gesture.js: A User Case Study Proposal

To evaluate the usability of Gesture.js with the assistance of creating embodied applications, a study detailing the learnability, ease of technological integration, creative flexibility, and reliability is suggested. As Gesture.js is intended to support application developers, the participants should intend to create a web-based embodied experience, thus requiring at least a general idea of their application’s architecture. This is so they can effectively decide whether to use Gesture.js as a service or directly. An introductory level of JavaScript is also required as Gesture.js is a JavaScript-based framework. Ideally, the participants should be a mix of software developers, designers, and researchers with various software development experiences.

The study should be conducted where a participant is instructed to develop a simple embodied application with pre-existing code, such as an HTML document with a button. Given Gesture.js’ documentation and open-source code, the participant is expended to apply Gesture.js as a layer and convert, say a button, to react to a user’s gesture or body position. The complexity of this activity can increase with the discretion of the conducting researcher. Regardless of the outcome of the participant, the following results, shown in Table 6, should be recorded.

Table 6

User Case Study Metrics and Descriptions

Metric Name	Purpose	Result Format Suggestions	Example
Learnability	This is recorded to calculate how easy it is to learn how to use Gesture.js. This is the evaluation of API documentation, semantics of functionality, intuitiveness of architecture, and judgment of the framework's attempt at abstraction.	Time spent consulting documentation; Participant comments; Amount of "errors" made during development	13 minutes spent consulting documentation
Ease of Integration	This is used to determine how easy it is to convert a pre-existing application into an embodied application. This is the evaluation of the portability and modularity of Gesture.js.	Participant comments; How many lines of code of the preexisting application were changed to support Gesture.js	The user fixed 2 lines of preexisting applications to incorporate gesture-based click
Creative Flexibility	This metric determines how easy it is to implement an envisioned embodied interaction with Gesture.js. This evaluates the capability and flexibility of the framework.	Similarity between participant's initial intention and final product; Participant comments; Amount of "errors" made during development	The user intended to create a click gesture with a pinch but resulted in using a fist
Reliability	This is recorded to calculate the number of technical errors that occurs during development. This evaluates Gesture.js' logical implementation.	Amount of technical difficulties occurred that were not caused by the participant; Log of technical errors	onHands() function does not fire when onFace() also runs
Comments	This metric is to capture any other qualitative experiences a participant would like to share.	Log of participant comments, questions, or suggestions	"I don't like how it does not capture two gestures at the same time."
Development Success	This is the final determination of whether the user successfully converted a preexisting application to an embodied one.	Binary result (Pass/Fail); Analysis or evaluation if the final application meets embodied standard	Partial Pass; User incorporated gesture but not the position of the hand

Next Steps for Gesture.js

As previously mentioned, Gesture.js is a cloud-deployable framework that can be installed as a node package manager (npm) module for Node.js applications. The module is currently public under the name @ahfowler/gesture. This module will be maintained and updated.

Gesture.js provides many benefits to both the software development and art and design community and industry. As its purpose is to help developers and designers create applications that are more engaging, immersive, and interactive without having to sacrifice their CPU or time to pick an application-specific algorithm and develop an interpretation/mapping technique, there are many beneficial impacts it can have on the general community. One impact is that it can make previously existing applications more accessible to those who struggle with motor abilities of traditional input methods, such as touch, mouse, or keyboard. Integrating movement and gestures into applications can make technology more inclusive and natural, and there is no need to build a new application infrastructure for it as Gesture.js serves as a layer on top of pre-existing HTML documents. Another benefit is that Gesture.js can increase the overall interactivity of software in general. By adding a new dimension of interaction that transcends typical input methods such as clicks and scrolls, web applications and gaming environments can increase engagement between users and applications, leading to more meaningful experiences.

A technological contribution of Gesture.js is the ability to provide its support in the form of service by leveraging server-side or cloud computing. As discussed in the analysis of the three gaming environments, when Gesture.js is used as a service, embodied applications can run more complex scripts and render higher quality experiences without having to worry about the computation and interpretation of body point landmarks. This leads to the last impact, which is to empower creativity among everyone regardless of technological background. By removing the technological limitations by simplifying the development and separating computing, it is the goal that Gesture.js will effectively make the development of embodied applications more accessible so anyone and everyone can leverage the power of using one’s body.

REFERENCES

Akcura, K. (2018). *NoTouch.js A JavaScript Library for Touch-Free Web Browsing.* Concordia University, Montreal.

Basques, K. (2017, April 6). *Analyze runtime performance*. Retrieved from Chrome Developers: https://developer.chrome.com/docs/devtools/evaluate-performance/

Bose, J. (2017, Sep 4). *Creating a Simple 3D Endless Runner Game Using Three.js*. Retrieved from envatotuts+: https://gamedevelopment.tutsplus.com/tutorials/creating-a-simple-3d-endless-runner-game-using-three-js--cms-29157

Derivative. (2021, April 21). *Frequently Asked Questions*. Retrieved from TouchDesigner by Derivative: https://docs.derivative.ca/Frequently_Asked_Questions

J, P. (2019, May 13). *Embodied interaction: body, movement and experience*. Retrieved March 2022, from UX Design: https://uxdesign.cc/embodied-interaction-body-movement-and-experience-ba1e5ea9d616

Lugaresi, C., Tang, J., Nash, H., McClanahan, C., Uboweja, E., Hays, M., . . . Grundmann, M. (2019). MediaPipe: A Framework for Building Perception Pipelines. 1-9.

McKenzie, C. (2014, August). *FaaS (Framework as a Service)*. (TechTarget, Producer) Retrieved from TechTarget: https://www.techtarget.com/searchcloudcomputing/definition/FaaS-Framework-as-a-Service.

MediaPipe. (2020). *MediaPipe Hands*. Retrieved from MediaPipe: https://google.github.io/mediapipe/solutions/hands.html

Micheal, H. (2013). *Gest.js*. Retrieved from GitHub: https://hadi.io/gest.js/

Mnkandla, E. (2009). About software engineering frameworks and methodologies. *AFRICON*, 1-5.

Ramos, O. (2021). *Handsfree.js*. Retrieved from Handsfree.js: https://handsfree.js.org/

Rembound. (2015, August 22). *Bubble Shooter HTML5*. Retrieved from Rembound: https://rembound.com/articles/bubble-shooter-game-tutorial-with-html5-and-javascript

Schrammel, J., Paletta, L., & Tscheligi, M. (2010). Exploring the Possibilities of Body Motion Data for Human Computer Interaction Research. *HCI in Work and Learning, Life and Leisure - 6th Symposium of the Workgroup Human-Computer Interaction and Usability Engineering*, (pp. 305-317). Klagenfurt.

Suresh, N. (2013, July 24). *Creating a 3D Game With Three.js and WebGL*. Retrieved from Build New Games: http://buildnewgames.com/webgl-threejs/

Wigdor, D., & Wixon, D. (2011, December). The Natural User Interface. In *Brave NUI World.*

Yeo, H.-S., Lee, B. G., & Lim, H. (2013, May). Hand tracking and gesture recognition system for human-computer interaction using low-cost hardware. *Multimedia Tools and Applications, 74*, 2687-2715.