Gesture Based UI Development Project

A Unity Application developed using the Kinect V2 that incorporates the unique use of various Gestures and Speech Patterns into two separate games linked together by a gesture-controlled UI

Faris Nassif & Alex Cherry | BSc (Hons) in Software Development

# <https://github.com/farisNassif/FourthYear_GestureBasedUIDevelopment>

# Purpose of Application

The purpose of this application is to explore and experiment with the capabilities of the Kinect v2 and the development components that come along with it such as the Visual Gesture Builder. At the end of development our goal is to have an application that

* Naturally incorporates practical gestures into various components that contribute to the fluidity of the overall application
* Is fully context aware
* Provides a high rate of accuracy in relation to gesture recognition
* Uses those highly accurate gestures as a base for seamless navigation and to provide an additional layer of engagement and fun that wouldn’t otherwise be found in a traditional game-based application
* Provides continuous feedback to the user
* Contains input definitions that are simple to perform and non-cumbersome
* Has Interactions that are simple, easy to learn, recognize and master

The gesture-based games we chose to implement were:

* **A Multiplayer Balloon Popping Game**

- *Controlled solely by the user’s hands*

* **An Endless Flyer**

- *Controlled by the tilt of the user that also incorporates voice commands*

Both games would be linked together by an easy to navigate UI that may be traversed with voice commands or gestures.

*Gifs of these games running are available in the README or alternatively a video is available showcasing the full project in the* [*Documentation*](http://www.github.com/farisNassif/FourthYear_GestureBasedUIDevelopment/tree/master/Documentation/Video) *folder in the Repository.*

# Gesture Identification and Rationale

Prior to our finalization of gesture implementation, we looked at three main types of gestures that we felt would enhance and be practical in our application

* Discrete Gestures
* Continuous Gestures
* Voice Recognition

# Discrete Gestures

A discrete gesture occurs once in a multi-input sequence and results in a single action sent. We ultimately decided it would make sense to incorporate a discrete gesture into the menu navigation and have it adhere to our previously outlined development goals.

We looked at the traditional Xbox One dashboard navigation for the Kinect that incorporates gestures like having the user open their hands parallel to the sides of the screen then bring them into a fist to navigate Home, or raising their hand to the sensor and ‘pushing’ it forward to access the System Menu. Initially we felt these gestures to be a bit convoluted and wanted to make our navigation as simple as possible while still effective.

We agreed to implement a ‘Swiping’ action that acted as a sort of Back function. We also heavily considered implementing a feature that allowed the user to traverse the menu by having an object represent the position of their right or left hand and when that object came into contact with an interactable game object for more than two seconds it would ‘Click’ that button. We decided to not implement that feature for reasons that we’ll outline further in the document.

In the end we constructed a UI that was fully traversable via voice commands and with the ‘Swipe’ gesture acting as a return function.

# Continuous Gestures

Continuous gestures differ from discrete gestures in that a continuous gesture passes through multiple phases. A continuous gesture begins, then over the course of several events may change throughout its cycle, then ends (or is cancelled).

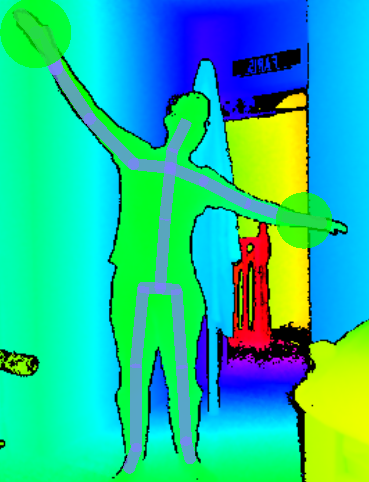
Initially for our endless flying game we had decided to introduce a discrete ‘Flap’ gesture that the player had to maintain to stop the character from falling. Initially we felt this to be a good idea and thought it made the most sense, we would have the character slowly fall which required regular flaps from the user. Following our creation and implementation of the gesture we quickly concluded we had made a mistake during our brainstorm phase.

While the gesture worked and was considerably accurate the gesture was incredibly uncomfortable and cumbersome to maintain for an extended period, something we failed to identify early on. We quickly shifted our scope and idea for the game and began to consider alternative gestures.

We ultimately decided to implement a continuous gesture that controlled the ascension, descension and hover status of the character that the user had to maintain. We trained and built three separate gestures that would cover the three potential states of the character

* Turn Right / Fly Up
* Turn Left / Fly Down
* Hover

**Turn Right / Fly Up**



**Turn Left / Fly Down**



**Hover / Maintain Altitude**



We trained each of the three continuous gesture builds with on average around ten training clips each to ensure a high rate of accuracy and seamless transition from one gesture to another s0 the game would be as fluid as possible and to avoid a choppy transition to hovering from ascending or descending.

The reason we went with such animated gestures rather than say a discrete hand gesture to control each state is because we felt it brought a sense of immersion to the game and fun, something that wouldn’t be found with a traditional state controller.

# Voice Recognition

Being a Kinect oriented application, the user would generally be standing away from their desktop meaning voice-controlled features would play a big role in all aspects of the project. We looked at different methods of implementing voice recognition, we had experience with creating grammar files with UWP applications, so this was something we considered before we opted to use Unity as our main development tool.

We found consequently that Unity has it’s own speech recognition libraries that turned out to be a lot more practical to implement and removed a bit of the convolution that comes with creating the individual grammar file and everything else that comes with interacting with it.

We could essentially declare a dictionary of words and map individual functions to each word in the dictionary, when the listener in a specific scene detected any of the declared words it fired off the corresponding function in the mapping.

The accuracy of the voice recognition was comfortably high with the only two downsides being the short to medium range from which you could say the key word and also the delay of around five-hundred to seven-hundred milliseconds. We implemented voice controls into all menu options as well as end of game functions, meaning the user never had to return to their device to confirm any action within the application itself.

Initially for our flying game we wanted to have a discrete gesture control when the player could shoot at incoming objects to destroy them, with us shifting our flying gesture to a continuous one this would become very impractical. To overcome this, we added a fire command that reacted when the player shouted ‘fire’, allowing them to maintain the continuous gesture of flying. As mentioned above, voice commands do have a slight delay, to compensate for this we greatly increased the speed of the projectile fired when ‘fire’ is called.

# Creating the Gestures

We looked mainly at two pieces of software to record and train our gestures, the first one being GesturePak. GesturePak simplified is an application that records your gestures via the Kinect and generates xml files to load those gestures from in the future. We honestly didn’t find much data or examples of implementing GesturePak or xml files for that matter into Unity applications. We felt it was a slightly outdated piece of software when compared with something like the Visual Gesture Builder.

The Visual Gesture Builder is a very intimidating piece of software initially. Though it comes with the Kinect SDK there really isn’t a lot of resources on how to work with it, however we were fortunate enough to find a handful of tutorials that helped to set it up.

The main idea behind the Visual Gesture Builder was to define a new gesture, record multiple clips of that gesture being performed and feed those recorded .xef files into that build. Once fed in, each frame of that clip had to be analyzed and declared as true or false, true being the point the gesture began to be performed and false being everything after that gesture stopped being performed.

To ensure maximum accuracy with similar gestures, like Turn Right / Hover / Turn Left, clips of incorrect gestures had to be loaded into builds and set to false to specifically tell that build to ignore those movements. For example, before we started doing this, gestures like Turn Left / Turn Right would often be mistaken for one another, we had to retrain our Turn Left gesture with clips of Turn Right and set all frames where turning right would be true to false to ensure they would be mutually exclusive and highly accurate.

Once around six to ten clips had been identified and integrated into the build, the gesture could then be generated as a gesture build database file (.gbd). Using the Visual Gesture Database API in Unity this type of file could be loaded into our application on runtime and stored as a Gesture variable within our project. Once encapsulated within a variable we could then check the frame reader for any instances of that gesture, once detected it was only a matter of executing a code block when it was confident the corresponding gesture had just been detected.

Overall, we felt comfortable on being able to build any gesture we wanted for our application with the only setback being the amount of time it takes to accurately train, generate and integrate the gesture into our project. It was for this reason we didn’t get around to implementing some practical gestures we felt would be great additions to our project, like for example individual gestures for menu traversal or something like a clapping gesture to pause a game.

# Hardware and Libraries

While discussing the different routes we could take with the project we did spent a lot of time on the topic of hardware. We wanted to integrate hardware that would compliment the vision we had and adhere to the goals of the project we previously outlined. We carefully considered the different hardware available to us, namely

* Leap motion controllers
* Myo Armbands
* Microsoft Kinect v2

Talk about the Myo/Kinect/Leap motion thing here, this part is written in past tense, so we haven’t done the project yet, we ‘wrote’ all the research stuff before

# Leap Motion Controller

The Leap motion controller was designed to mainly follow finger and hand movements. It consists of a small USB device placed near the desktop which works by illuminating the space near the camera with an infrared light that helps it locate the user’s hands and fingers, allowing it to analyze their location and orientation.

The Leap motion controller’s small area of scope means it’s highly accurate and precise at tracking, excelling at integration into a buzz-wire or operation simulation application, something that the Kinect for example wouldn’t be specialized at.



Leap Motion Controller

# Myo Armband

The Myo Armband allows the wearer to wirelessly interact with applications via hand gestures. The armband measures electrical activity from muscles in your arm to detect gestures made by your hand and the intensity of those gestures. Using electromyographic sensors it can also measure the motion, orientation and rotation of your forearm.

The armband is excellent at isolated gestures and measuring the intensity of those gestures, for example measuring the intensity of a flick, power of a punch or the strength of your grip, tasks the Kinect or Leap motion controller wouldn’t be able to perform.

# 

Myo Armband including Gestures

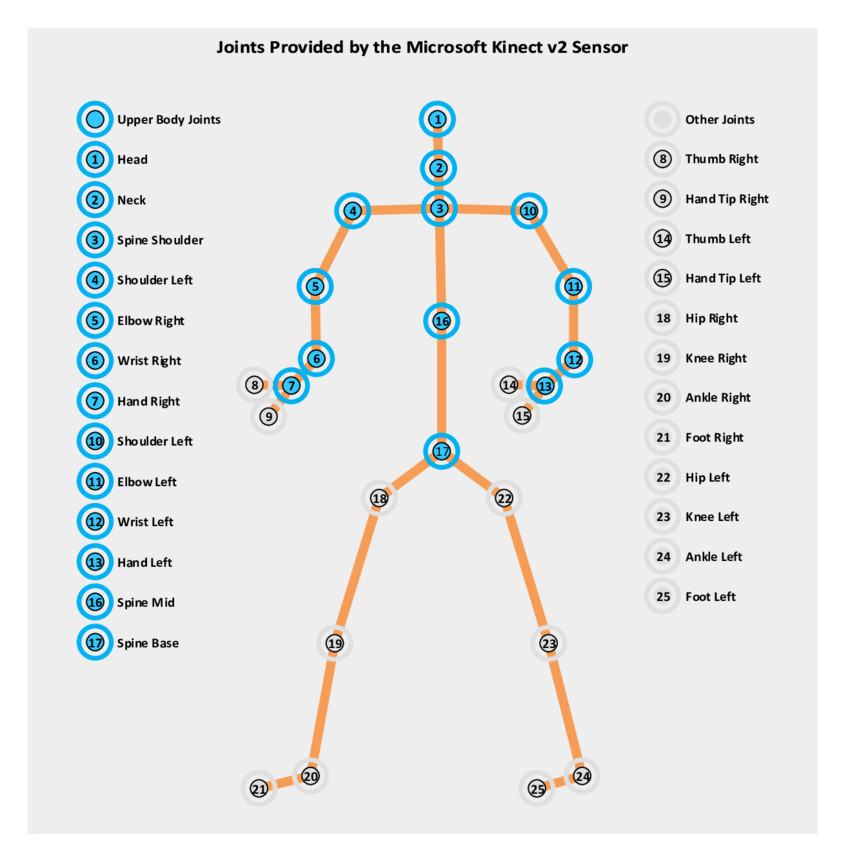
# Microsoft Kinect

The Kinect v2 was Microsoft’s add-on for the Xbox gaming console, it contains three major pieces that work to generate its output stream

* Depth sensor
* RGB colour VGA camera
* Multi-array microphone

The camera has a pixel resolution of 640 x 480 and a frame rate of 30 fps which helps with facial and body recognition. The Kinect is also able to calculate the distance of each point of the user’s body by transmitting infrared light and measuring its travel time after it bounces off objects. The microphone consists of an array of four microphones that may isolate the player’s voice from other background noise.

The Kinect’s sensor can isolate twenty-five different joints, dwarfing other hardware options in terms of body scale that may only isolate a handful.



Joints tracked by the Microsoft Kinect

The Kinect thrives in applications that incorporate a lot of body movement, for example a Flappy Bird type game that requires the following of different targeted joints within the shoulders and arms or even applications that aim to fully mirror the actions and movements taken by a user’s body.

# Hardware Selection

# Application Interface

# Menu/UI System

The Menu and User Interface incorporate several features for ease of use for the User. It includes:

* A visual representation of the Kinects body image
* Voice Recognition for navigation
* Swiping gestures for navigation
* Score page for High Scores and clearing Scores
* Buttons for navigation if the User is not using gestures

# Research

Our main source of research blah blah blah

You looked at the stuff in the Library, that’s some good research boyo. A lot of the research was just done while trying to find people that actually used the Kinect for this stuff. We used Unity because of something. The voice recog crap was really easy to implement with Unity, and the Kinect was done in a lab? I can’t remember.

# Resources

Some images maybe of some similar games don’t know, and maybe some gesture diagrams or something??

YouTube videos would be good here too, Brackeys and Alexander Kalashnikov. Birb game was thought of cause we like Airplanes.

# Application Design

# Design Goals?

Voice navigation, swipe, user experience, interactions, everything is constant time back to the user, types of gestures, continuous/discrete etc.

Our goals were to implement as many gestures as possible into our game so that it had to be controlled as little as possible with your hands. Pretty much the whole game can be done without actually touching the computer, pretty cool.

# Divided Workload :

Faris Nassif :

* Kinect Gesture Database
* Motion Control for Games
* Menu Navigation using Swipe Gesture
* Scoring Systems
* Game Mechanics

Alex Cherry :

* Voice Recognition
* Menus and Navigation
* Sprite Animations
* Game Mechanics

The workload was divided accordingly as to who had the hardware for the application. Multiple opinions were required to assess the Game Mechanics for the two games designed. A database creation was required for the Kinect V2 which was designed by Faris, requiring several inputs of repeated gestures to read properly. Voice recognition systems were designed by Alex to take only certain keywords in certain areas or scenes in Unity for accuracy and consistency.

# Architecture

Diagram describing software architecture or something

# Libraries/Game Engine/Talk more about architecture diagram

# Testing

Used my brother for testing because sure he may as well have

# Strategy/Methodology

Integration testing, Regression testing, system testing, always testing along while getting functions working (regression testing). Test other functions once other function has been added (integration testing) and I guess this is system testing too.

# Who tested it?

Alex’s brother and Faris’ dogs.

# Acceptance/Beta tests

# Methodology

# Results

We added or changed this thing because we tested it and yea

# Conclusion

The project is pretty awesome, turns out flappy bird is a fun game when anime music is put into it. Balloon game gives me nightmares of PC crashes but other than that its pretty noice.

# What we learned

We learned that the Kinect is pretty cool, and that you can make pretty much any Gesture you want by using the Kinect V2s built in Gesture Database system with Unity. It requires immense repeated testing but is very accurate once fully functional.

# What we would do differently

Not get Coronavirus lul

# Test Cases

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Test Case ID** | **Test Scenario** | **Test Steps** | **Test Data** | **Expected Result** | **Actual Result** | **Pass** |
| TC01 | Voice Commands | 1. Start Application 2. Speak through microphone | N/A | Voice is recognized and commands work accordingly | Voice is recognized and commands work accordingly | True |
| TC02 | Gesture Activation for Balloon Game | 1. Start Application 2. Load Balloon Game 3. Wait for Joint Recognition | N/A | Joints appear and move around the screen | Joints appear and move around the screen | True |
| TC03 | Hand Gesture Collision Balloon Game | 1. Start Application 2. Load Balloon Game 3. Wait for Joint Recognition 4. Move hands into balloons | N/A | Balloons are popped upon collision when moving hands | Balloons are popped upon collision when moving hands | True |
| TC04 | Gesture Activation for Bird Game | 1. Start Application 2. Load Bird Game 3. Raise arms and tilt them left or right | N/A | The Bird ascends or descends correctly | The Bird ascends or descends correctly | True |
| TC05 | Game Over Sequence Balloon Game | 1. Start Application 2. Load Balloon Game 3. Complete Game | N/A | Game ends correctly and score is shown | Game ends correctly and score is shown | True |
| TC06 | Game Over Sequence Bird Game | 1. Start Application 2. Load Bird Game 3. Complete Game | N/A | Game ends correctly and score is shown | Game ends correctly and score is shown | True |
| TC07 | High Score Register Balloon Game | 1. Start Application 2. Load Balloon Game 3. Complete Game 4. Go to Score Menu | N/A | Score is saved correctly | Score is saved correctly | True |
| TC08 | High Score Register Bird Game | 1. Start Application 2. Load Bird Game 3. Complete Game 4. Go to Score Menu | N/A | Score is saved correctly | Score is saved correctly | True |