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 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

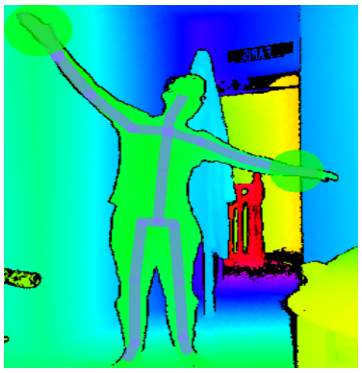
falling. Initially we felt this to be a good idea and thought it made the most sense, we would have the character slowly fall which required regular flaps from the user. Following our creation and implementation of the gesture we quickly concluded we had made a mistake during our brainstorm phase.

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

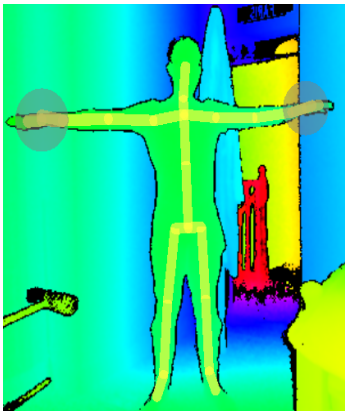
- Turn Right / Fly Up
- Turn Left / Fly Down
- Hover

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 so 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 four-hundred to six-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

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 specialized at.

Myo Armband including Gestures

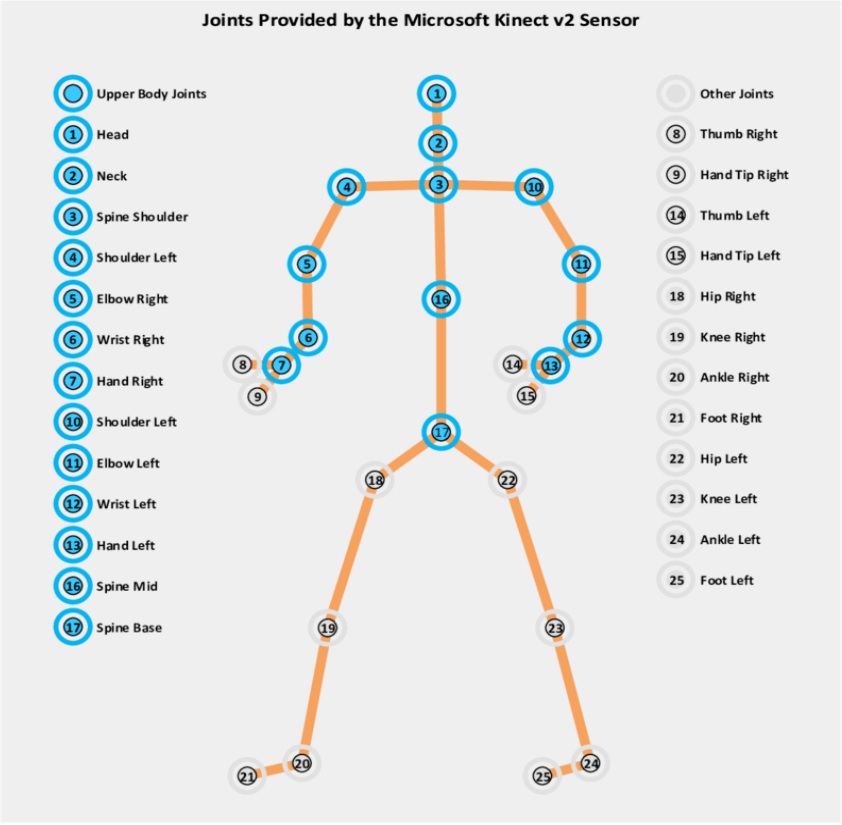
Microsoft Kinect

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

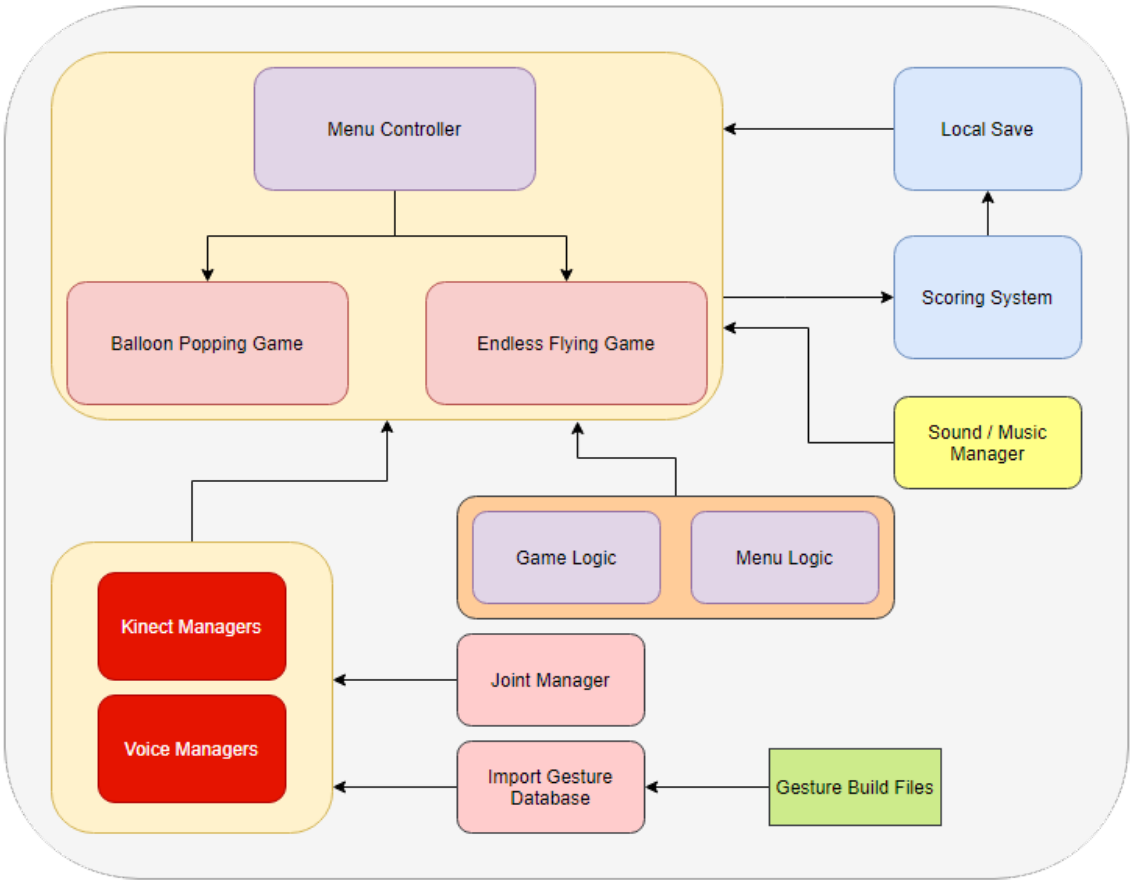
Our desire was to create an application that could capture multiple points on the user's body and translate those points into game objects that perform some function so naturally our go-to piece of hardware had to be the Kinect for reasons outlined in our analysis.

Initially we had discussed creating a drum-set simulator application with the Leap motion controller or the Myo armband since the application wouldn't require a huge field of view and was focused on hand or arm movement, however when it came to menu navigation, gesture creation and voice recognition the Kinect did look a lot more attractive. The main deciding factors for us were

- Kinect's visual gesture builder
- The multi-array microphone
- A wide range of possible gestures that wouldn't only incorporate hands or arms

In the end we didn't end up developing the drum-set simulator application, however the point still stood, and we would soon find our decision to be one we didn't look back on. The microphone allowed for highly accurate voice commands, the visual gesture builder opened a lot of doors for us in terms of possibilities we could have taken with the application and the scope of gestures weren't limited.

Solution Architecture



Solution Architecture containing the main components

Core Technologies

- Unity – Version 2019.3.0b5
- Kinect 2.0 SDK

Core Libraries and Development Tools

- Kinect Studio
- Visual Gesture Builder 2.0
- Unity Speech Recognition API
- Kinect for Windows v2 API

- Visual Gesture Builder for Unity API
- Kinect 2.0 Unity Pro Package

Architecture Design

There are a lot of moving parts with this project such as, Menu Managers, Gesture Database Managers, Kinect Managers, Voice Managers and that's not including the core Application Logic Managers. We attempted to loosely couple as much as we could to promote reusability and since this was a group project to promote readability. All classes are heavily commented to help remove any ambiguity one of us may have when working with the other member wrote.

Gesture Managers

The Gesture Managers consist of both the Voice Managers and the Kinect Managers. These classes are implemented into every scene in the application since the application is completely gesture driven. On runtime the gesture build files are read by the Gesture Database which then feed those loaded in gestures into the Kinect Manager classes. With help from the Joint Manager, the Kinect Manager can check every frame for an instance of any of those gestures and fire off a corresponding function if successful.

The Voice Manager classes are a bit different, there isn't any outside files required so they integrate a lot easier and don't require as much setup. They simply listen on every scene for a pre-defined key word and execute the method mapped to that key word.

We felt this to be the best path to take, initially we had the Gesture Managers contained within the Game Manager and Menu Manager since they only really needed to interact with each other. However once our project grew in size and scope everything quickly became very convoluted, so we took action to separate the two components.

Game & Menu Logic

Essentially contains the core classes for the whole application. These group of classes control the execution of the program from start to finish, from what happens when a button is pressed to what happens when a new game is loaded. The functions within this group of classes are those that are mostly executed by Gestures or Voice Commands.

Score Manager

The scoring system within this application allows for persistent high scores using Player Preferences which can map scores to a key. When the key is called a local search is performed that retrieves the score associated with the key. Scores are set and viewable at the end of each game and may be checked via the menu.

Sound Manager

Initially we had very little regard for implementing sounds however after testing we did feel they served as great feedback for the user especially when the user was away from the desktop. Sound is incorporated into all scenes in the project, from game music to make everything even a bit more ambient, to reactive sounds to confirm commands or gestures executed by the user.

An example of where we used sound to give real time feedback to the user is during our endless flying game, we found it very hard to track the cooldown for shooting while away from the screen so added sound to reflect the shot being fired, the object being destroyed, the health being restored and also the shoot charge being fully recharged.

Core Classes and Scripts

- Balloon Game Scripts
  - Scripts for our Balloon Popping Game
- Bird Game Scripts
  - Scripts for our Endless Flying Game
- Kinect Scripts
  - Body Source Manager and Body Source View being the main relevant ones
- Import Gesture Database
  - Essential Script for loading in prerecorded gestures
- Voice Recognition Script
  - Essential Script for declaring and mapping voice commands
- Sound Manager Script
  - Script to manage sound throughout the application

Testing

Our tests ranged from looking at UI gesture interaction and game functions to fine tuning the overall performance of the application. Testing provided us with valuable insight and allowed us to continuously integrate new features and components we would otherwise overlook which I'll discuss further.

Testing Approach

We naturally adopted a Continuous Integration style of testing. While we can't automate our tests, automation isn't strictly a requirement of CI testing. Each code integration would be manually tested for its intended functionality and verified by the non-submitting project member, we found this to be surprisingly yielding in terms of results since having someone else look at and test your implementation would generally lead to hidden pitfalls and bugs that the submitting party wouldn't initially identify.

**Due to hardware sharing complications there was parts of the project that couldn't be tested by the corresponding member*

Test Case ID	Test Scenario	Test Steps	Test Data	Expected Result	Actual Result	Pass
TC01	Voice Commands	1. Start Application 2. Speak through microphone	Voice input via Kinect Microphone	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	Video input via Kinect Camera	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 Popping Game 3. Move hands into balloons	Video input via Kinect Camera	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	Video input via Kinect Camera	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

Post-Development Testing

To ensure our application was ready for deployment we thoroughly tested and compared actual outcomes to our desired outcomes.

Unfortunately, due to the current international and national situation we couldn’t be picky about our selection of testers which meant our team of testers consisted of both group members as well as family members.

User Acceptance Testing Results

We identified several features and additions that could be changed, implemented or removed as a result of our tests. These included

- Lack of constant gesture feedback during menus
    - **Result**: We implemented a skeleton joint-body that represents the user.
- Lack of feedback in our Flying Game while flying regarding gestures
    - **Result**: We added constant feedback on screen that output the current gesture being performed.
- Sound responsiveness was poor and didn’t compliment gestures
    - **Result**: Sounds were added to shooting, collision and balloon popping.
- Swipe responsiveness was poor when performed by a non-developer
    - **Result**: The Swipe gesture was trained on more clips and re-imported into the project.
- The Balloon Popping game would start without the user being ready
    - **Result**: The game only began when hand joints were read by the frame reader.

Other minor changes including music or sound volume altercations and game mechanics being too punishing, these were dealt with accordingly. We’re pleased with the testing results considering our small testing pool.

Conclusion

The project was very intimidating initially and slow to pick up pace, with that said once we began our research phase and began to integrate components and technologies then saw how they clicked everything became very practical. The biggest hump to get over was creating our own gestures and incorporating those into Unity. Fortunately we found some resources that enabled us to seamlessly continue to develop and integrate new libraries and technologies.

Prior to the project neither of us would have ever considered using the Kinect to develop software, now knowing the availability of compatible applications, libraries and tools like the visual gesture builder it's highly possible that future personal projects of ours could see the Kinect make an appearance.

What We've Learned

- Gestures can be highly effective, but they must be practical
    - We learned this firsthand when it came to our initial idea to implement a flap gesture to control player altitude. While it seemed like a good inclusion it really wasn't practical to have a user constantly flap their hands for minutes on end. We settled for a middle of the road solution that still included continuous hand positional gestures that were a lot less cumbersome and more practical.

- Gesture accuracy and simplicity contribute greatly to the fluidity of the overall application
    - Initially before we had trained our gestures thoroughly, they were incredibly clunky and degraded the overall pacing of the project. A gesture driven application must include gestures and interactions that are simple to learn, recognize and perform.
    - Before we conducted our user acceptance testing all gestures had been created and performed by ourselves, we failed to consider how a user might interpret a 'swipe' or 'hover' gesture. This brought us back to the drawing board and meant retraining gestures to not be as specific but rather be a bit more forgiving and lenient.

- Constant feedback to the user improves the overall experience
    - A responsive environment is vital regarding user experience. Instructions must be placed at every point throughout the project that describe the possible gestures executable at that given point. We accomplished this by having multiple notifications active throughout the project and tutorial pages that instructed the user the different possible routes they could take through the application and how to navigate those routes.
    - In our games we also accomplished this using feedback sounds and on-screen statuses to provide as much feedback as possible.

Recommendations

- Integrate and experiment with more discrete gestures
    - In the early stages of development, we initially had planned to introduce more discrete gestures, one being ‘clap’, which would act as a pause. We also discussed menu traversal via hand movements. Deadline constraints and the time required to build and train those gestures meant we had to abandon those features.

- Include and experiment with more hardware
    - The Kinect and its toolkit are great, but as previously identified in the hardware section there are some tasks that the Kinect doesn’t specialize at. In theory we could also integrate more technologies like including a Myo armband to measure the intensity of a gesture or associate a specific muscle action to a joint movement and incorporate additional mechanics into our application.

- Begin testing earlier and more often
    - Frequent testing from the start would have absolutely saved us a lot of complications further down the road especially when it came to gestures. While we did identify most of the major issues eventually, if caught earlier we could have maintained good practice and saved ourselves a whole lot of time.

Reflections

As mentioned above this was one of our more intimidating projects regarding it being uncharted waters for us. Once the main hump was overcome it

become one of our more enjoyable projects and hope that’s evident from the amount of work committed to it.

We learned the power well thought out and practical gestures can bring to an application assuming they’re accurate and non-cumbersome to replicate and also how testing early and often can iron out potential issues that may arise further down the development line.

Considering the scope of the project we’re delighted with where we’ve finished off. We’ve achieved our goal of developing two different games that incorporate different gestures and mechanics and integrating those games into a gesture responsive UI. Moreover, we’ve accomplished this while adhering to our development goals outlined in our introduction.

We’re especially happy with the overall accuracy and practicality of gestures incorporated into the application and how those gestures contribute to the fluidity of the overall application while being simple to learn and recognize. As an honorable mention we’re also pleased with how the multiplayer option turned out in the Balloon Popping game considering it doesn’t have any gesture heavy mechanics other than joint translation.