KINECT™ BASED BIOLOGY EDUCATION SYSTEM

A Thesis

Presented to

The Graduate Faculty of The University of Akron

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Kritsakorn Chaumpanich

May, 2015
KINECT™ BASED BIOLOGY EDUCATION SYSTEM

Kritsakorn Chaumpanich

Thesis

Approved:  
Advisor  
Dr. Yingcai Xiao

Faculty Reader  
Dr. Philip A. Allen

Faculty Reader  
Dr. En Cheng

Accepted:  
Dean of the College  
Dr. Chand Midha

Interim Dean of the Graduate School  
Dr. Rex D. Ramsier

Date  
Dr. Timothy Norfolk
ABSTRACT

Biology is the science of studying life and living organism. Teaching Biology is difficult because of lack of interaction and student engagement by using textbooks and PowerPoint presentations. This project introduces a new technology, NUI (natural user interface) to Biology education. We designed and implemented a system with the Microsoft Kinect™ NUI, which allows students to actively explore virtual 3D educational contents with their hands instead of showing pictures with verbal descriptions. This approach promotes active, rather than passive, learning by employing embodied recognition. We created realistic 3D Biology models including human cells, nucleus, DNAs, etc. and animated them to demonstrate biological processes. To explore different NUI interaction styles, we carefully designed and implemented methods for both user-object interaction and mode switching. Hand-pushing and hand-holding methods were compared for user-object interaction, whereas hand-dropping and head-shaking were compared for mode switching.

Our study found that students who use the Kinect™ based system score better than the students who use the traditional teaching method. We also found that the hand-holding method is more suitable for NUI interaction compare to the hand-pushing method. In addition, the hand-dropping method is more suitable than
the head-shaking method for mode switching between user interface interaction and object manipulation. Deployment of the system in college and high school classrooms have shown the Kinect™ based system is a very useful tool for biological education.
ACKNOWLEDGEMENTS

I am very thankful to my parents, for encouraging and supporting me in pursuing my higher studies and also making this thesis possible.

I would like to acknowledge the professors who inspired me throughout my masters program. Dr. Yang H. Yun and Dr. Phillip A. Allen, thank you very much for inspiring me through this project and supporting and advising me in developing this software tool. I am very thankful to Dr. En Cheng for your advice throughout my entire master program. Dr. Yingcai Xiao, thanks a lot for being my professor and advisor, giving the opportunity at the same time motivating, encouraging and supporting me in the entire program.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF TABLES</th>
<th>ix</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>x</td>
</tr>
</tbody>
</table>

## CHAPTER

### I. INTRODUCTION  

1.1 Motivation  

1.2 Hypothesis  

1.3 Outline of Thesis  

### II. BACKGROUND  

2.1 Human Computer Interaction  

2.2 Overview of Hand Gesture Recognition  

2.3 Overview of Unity3D  

2.4 Overview of Microsoft Kinect™  

2.5 Overview of 3D Modeling Software  

### III. DESIGN  

3.1 Education Design  

   3.1.1 Biology Materials  

   3.1.2 Education Design
3.2 System Design .................................................. 14
  3.2.1 System Architecture ....................................... 15
  3.2.2 Natural User Interface .................................... 16
  3.2.3 Record Keeping ............................................. 18
3.3 Usability Design .................................................. 19
  3.3.1 User Interface Design ....................................... 20
  3.3.2 Interactivity ................................................ 20
  3.3.3 Hand Gestures .............................................. 22
  3.3.4 Mode Switching ............................................. 23
IV. IMPLEMENTATION ............................................... 25
  4.1 Education Component ......................................... 25
    4.1.1 Biology Contents ......................................... 25
      4.1.1.1 Assets ............................................... 30
      4.1.1.2 Animation ............................................ 40
  4.2 System Implementation ....................................... 41
    4.2.1 Unity3D with Kinect™ .................................... 41
    4.2.2 Animation with Unity3D .................................. 45
      4.2.2.1 Key Frame Animation .................................. 45
      4.2.2.2 Morphing ............................................. 46
  4.3 Usability Implementation .................................... 48
    4.3.1 User Interface Interaction .............................. 48
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Functions from the MonoBehavior class.</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Kinect features and benefits.</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>Experimental design</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>Example: Level of success by task</td>
<td>24</td>
</tr>
<tr>
<td>4.1</td>
<td>All animation scenes in the system</td>
<td>40</td>
</tr>
<tr>
<td>4.2</td>
<td>List of some useful components from Zigfu’s Unity package.</td>
<td>42</td>
</tr>
<tr>
<td>4.3</td>
<td>Lists of callback function for collision detection.</td>
<td>51</td>
</tr>
<tr>
<td>4.4</td>
<td>Detail about ray-casting function in Unity3D</td>
<td>59</td>
</tr>
<tr>
<td>5.1</td>
<td>Paired samples statistics</td>
<td>65</td>
</tr>
<tr>
<td>5.2</td>
<td>Paired samples test</td>
<td>65</td>
</tr>
<tr>
<td>5.3</td>
<td>CPU usage on Microsoft Windows 7</td>
<td>68</td>
</tr>
<tr>
<td>5.4</td>
<td>GPU usage on Microsoft Windows 7</td>
<td>68</td>
</tr>
<tr>
<td>5.5</td>
<td>CPU usage on Microsoft Mac OS X</td>
<td>70</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Components of working memory</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>Human computer interaction</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Supporting platform for Unity3D.</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>A component in Unity3D that derived from the MonoBehavior class.</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>Microsoft Kinect\textsuperscript{TM}</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>The layered system architecture</td>
<td>16</td>
</tr>
<tr>
<td>3.2</td>
<td>User tracking by Kinect\textsuperscript{TM}</td>
<td>18</td>
</tr>
<tr>
<td>3.3</td>
<td>Record keeping database for the system.</td>
<td>19</td>
</tr>
<tr>
<td>3.4</td>
<td>Controller bars on the left and right side of the scene.</td>
<td>21</td>
</tr>
<tr>
<td>3.5</td>
<td>UI interaction from Kinect Adventure</td>
<td>22</td>
</tr>
<tr>
<td>3.6</td>
<td>Two important user tasks in the system.</td>
<td>24</td>
</tr>
<tr>
<td>4.1</td>
<td>Polygon primitives in Maya.</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>Attribute editor in Maya.</td>
<td>28</td>
</tr>
<tr>
<td>4.3</td>
<td>Object Component Mode in Maya.</td>
<td>28</td>
</tr>
<tr>
<td>4.4</td>
<td>A living cell model in Maya.</td>
<td>29</td>
</tr>
<tr>
<td>4.5</td>
<td>Assets for the cell module.</td>
<td>30</td>
</tr>
<tr>
<td>4.6</td>
<td>Assets for the cell module.</td>
<td>31</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.7</td>
<td>Assets for the cell module.</td>
<td>32</td>
</tr>
<tr>
<td>4.8</td>
<td>Assets for the DNA module.</td>
<td>33</td>
</tr>
<tr>
<td>4.9</td>
<td>Assets for the DNA module.</td>
<td>34</td>
</tr>
<tr>
<td>4.10</td>
<td>Assets for the immune system module.</td>
<td>35</td>
</tr>
<tr>
<td>4.11</td>
<td>Assets for immune system module.</td>
<td>36</td>
</tr>
<tr>
<td>4.12</td>
<td>Assets for the immune system module.</td>
<td>37</td>
</tr>
<tr>
<td>4.13</td>
<td>Assets for the gene therapy module.</td>
<td>38</td>
</tr>
<tr>
<td>4.14</td>
<td>Assets for the gene therapy module.</td>
<td>39</td>
</tr>
<tr>
<td>4.15</td>
<td>A user interact with the system.</td>
<td>43</td>
</tr>
<tr>
<td>4.16</td>
<td>3D hand models.</td>
<td>43</td>
</tr>
<tr>
<td>4.17</td>
<td>ZigSkeleton public variables.</td>
<td>44</td>
</tr>
<tr>
<td>4.18</td>
<td>Timeline for key frame animation.</td>
<td>46</td>
</tr>
<tr>
<td>4.19</td>
<td>Morphing technique.</td>
<td>47</td>
</tr>
<tr>
<td>4.20</td>
<td>Example of morphing: lipid bilayer forms an endosome.</td>
<td>48</td>
</tr>
<tr>
<td>4.21</td>
<td>Hand interaction.</td>
<td>51</td>
</tr>
<tr>
<td>4.22</td>
<td>Example code for callback function.</td>
<td>52</td>
</tr>
<tr>
<td>4.23</td>
<td>Poses for objects scaling.</td>
<td>54</td>
</tr>
<tr>
<td>4.24</td>
<td>Poses for objects rotating.</td>
<td>55</td>
</tr>
<tr>
<td>4.25</td>
<td>A user is interacting with the system.</td>
<td>58</td>
</tr>
<tr>
<td>4.26</td>
<td>Example code for CORayCast.</td>
<td>61</td>
</tr>
<tr>
<td>4.27</td>
<td>Example program for capturing the head shaking.</td>
<td>62</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>5.1</td>
<td>An example of the Profiler window.</td>
<td>66</td>
</tr>
<tr>
<td>5.2</td>
<td>CPU usage for Microsoft Windows 7</td>
<td>69</td>
</tr>
<tr>
<td>5.3</td>
<td>CPU usage for Mac OS X</td>
<td>71</td>
</tr>
<tr>
<td>5.4</td>
<td>Levels of success for the hand pushing method.</td>
<td>73</td>
</tr>
<tr>
<td>5.5</td>
<td>Levels of success for the hand holding method.</td>
<td>73</td>
</tr>
<tr>
<td>5.6</td>
<td>Mean time on tasks</td>
<td>74</td>
</tr>
<tr>
<td>5.7</td>
<td>Levels of success for the head shaking method.</td>
<td>75</td>
</tr>
<tr>
<td>5.8</td>
<td>Levels of success for the hand dropping method.</td>
<td>76</td>
</tr>
<tr>
<td>5.9</td>
<td>Efficiency chart for the mode switching.</td>
<td>76</td>
</tr>
</tbody>
</table>
1.1 Motivation

Currently, there are many teaching materials being used beside books such as computers and internet. These materials help students to better understand educational contents. Computers provide all the information of a book and more. Research has shown that learning is more efficient when stimulating the visual component of the brain (the visuo-spatial sketchpad), rather than using language-based approaches [1]. Figure 1.1 is the working memory model of Baddeley and Hitch [2]. It consists of three components; central executive, short-term storage centers (the phonological loop and the visuo-spatial sketchpad), and the episodic buffer. The central executive control the flow of information to and from the phonological loop, the visuo-spatial sketchpad, and the episodic buffer. The phonological loop controls and stores verbal codes, whereas the visuo-spatial sketchpad controls visual codes. The episodic buffer links information across the domains with time sequencing. We can do this by using video games. We can use visual codes to help supplement verbal codes in working memory, because these codes can be used to interfere with working memory performance [2].
Furthermore, there is evidence that humans use “embodied cognition” to improve performance [3, 4, 5]. We do so by encoding the state of the body and using it as a “somatic marker” (or contextual cue), when storing working-memory information in the long-term memory [3, 4, 5, 6]. Using these assumptions, it is predicted that visual codes and motoric feedback from movements should improve the educational outcomes in learning science and engineering information.

The goal of this project is to refocus biological education by introducing a new technology that is integrated with interactive, educational content that engages, inspires, and stimulates the visual center of the brain. To accomplish these goals, a new curriculum has been designed and implemented. It engages students and interactively introduces visual information through the Microsoft Kinect™ device [7]. This device will be connected to a computer system and will allow students (as an example), to actively explore a cell’s anatomy as well as by its functions using an interactive spatial interface instead of simply showing pictures with verbal descriptions. This approach also promotes active, rather than passive, learning. It
does so by employing embodied cognition. This implies that human movements could impact a lasting effect on both the short-term episodic and long-term memories of students [3, 4, 5, 6]. In our approach, both the visual/spatial information and the motoric interactions can be stored with a memory trace of the learning event (e.g., structures and functions of an eukaryotic cell and its organelles).

1.2 Hypothesis

We hypothesize that by introducing NUI-based interactive learning of 3D graphical content, which is designed specifically to simulate the visuo-spatiaal sketchpad with gestures, it will positively impact the students’ learning and have a lasting effect on cognition and episodic memory [8, 9]. Since our educational content is also designed similar to a game interface, it will also enhance the emotional states of the students. This will potentially enhance episodic memory performance (e.g., remembering events) [9] as demonstrated by applications of Damasio’s somatic marker hypothesis [3, 4, 5]. Thus, encoding the visual information (in visuo-spatial sketchpad), along with integrated gestures, should result in more accurate retrieval of the contextual information than will lead to more efficient learning.

1.3 Outline of Thesis

This thesis is divided into six chapters. The first chapter is about motivation and hypothesis, whereas Chapter Two explains the background for each related
knowledge such as hand gesture recognition. Next, system design and implementation are in Chapter Three and Chapter Four. Chapter Five is about experimental results. Lastly, conclusion and future work are in Chapter Six.
2.1 Human Computer Interaction

Human Computer Interaction (HCI) deals with the interaction between humans and computer through a user interface[10]. There are several methods for humans to interact with the computer such as Command-Line Interface (CLI), Graphical User Interface (GUI), and Natural User Interface (NUI). The Command-Line Interface(Figure 2.1a) is a text-based command that requires users to know specific commands in order to interact with computer while the GUI(Figure 2.1b) allows users to interact with the computer using graphical icons and visual indicators, such as menus and buttons.

Figure 2.1: Human computer interaction.
GUI is less challenging for the users because they can interact with the computer without having to memorize the commands. Additionally, the Natural User Interface (Figure 2.1c), a new trend of HCI, allows the users to interact with the computer naturally. NUI uses voice and gestures as input with input devices [11], such as Kinect™, Leap motion, etc. In this project, we use Kinect™, which is a motion sensing input device developed by Microsoft. It enables users to interact with the computer using gestures and spoken commands.

2.2 Overview of Hand Gesture Recognition

Hand gestures can be used to produce interactions between humans and computers. Hand gesture recognition requires a capture device, such as a camera, an infrared camera, and an instrumented gloves. These items capture data as an image, position, and rotation from the users’ hands. Hand gesture recognition is typically classified into two categories including static gestures and dynamic gestures [12, 13]. Static gestures are used to describe hand poses while dynamic gestures are used to identify hand movements. There are several ways to operate hand gesture recognition. According to Byung-Woo Min et al. [14], static hand gestures and dynamic hand gestures are recognized by using structural analysis and using Hidden Markov Models, respectively. Ghosh and Ari [15] introduce static hand gesture recognition by using k-means with neutral network. They use 500 images as a training set and another 500 images as a test set for 25 gray scale images of static hand gestures. In this system, we plan to conduct both static and dynamic hand gesture recognition.
The users are able to apply these gestures in order to interact with the program, in this case, the users will be able to rotate and scale a virtual object on the scene.

2.3 Overview of Unity3D

It is worth noting that a game engine is one of the most important tools for game developers. This project focuses on using Unity3D, a cross platform game development engine, to develop the program. Unity3D is compatible with Windows, Mac, Linux, and other mobile platforms such as iOS and Android (Figure 2.2), which allows users to create scenes for their application by directly hitting a play button to see the actions of the program[16]. The users are also able to import 3D objects from other 3D modeling programs such as Autodesk Maya, 3DMax, Blender, etc., Moreover, the Unity3D includes a built-in physics engine providing components that handle the physics simulation for developers. Since the Unity3D provides an IDE called MonoDevelop (a cross-platform that is designed for C# or JavaScript), the users are able to write their own scripts on Unity3D using C# or JavaScript. In order to write a script for the Unity3D component, every class needs to derive from MonoBehavior class, which contains public function and public variables (Table 2.1.
Table 2.1: Functions from the MonoBehavior class.  

<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Awake</td>
<td>Awake is called when the script instance is being loaded.</td>
</tr>
<tr>
<td>FixedUpdate</td>
<td>This function is called every fixed framerate frame.</td>
</tr>
<tr>
<td>LateUpdate</td>
<td>LateUpdate is called every frame.</td>
</tr>
<tr>
<td>OnCollisionEnter</td>
<td>OnCollisionEnter is called when this collider/rigidbody has begun touching another rigidbody/collider.</td>
</tr>
<tr>
<td>OnCollisionExit</td>
<td>OnCollisionExit is called when this collider/rigidbody has stopped touching another rigidbody/collider.</td>
</tr>
<tr>
<td>OnMouseDown</td>
<td>OnMouseDown is called when the user has pressed the mouse button while over the GUIElement or Collider.</td>
</tr>
<tr>
<td>Reset</td>
<td>Reset to default values.</td>
</tr>
<tr>
<td>Start</td>
<td>Start is called on the frame when a script is enabled just before any of the Update methods is called the first time.</td>
</tr>
<tr>
<td>Update</td>
<td>Update is called every frame, if the MonoBehaviour is enabled.</td>
</tr>
</tbody>
</table>
Figure 2.3: A component in Unity3D that derived from the MonoBehavior class.

2.4 Overview of Microsoft Kinect™

Microsoft Kinect™ was first introduced by Microsoft Corporations in 2010. Kinect™ is a 3D motion sensing device used in Xbox 360, Xbox One video game console, and Window PCs[17, 7]. It consists of a microphone, 2 infrared cameras, an RGB camera, and a motorized tilt as shown in Figure 2.4. The device enables users to naturally interact with a computer (console) without a joint controller. Using an infrared projector, an RGB camera, and a microphone allows Kinect to connect with the users, as well as tracking all the players’ movement that will be sent to the core software for further processing. In addition, the infrared camera combines sensors that emit a huge amount of infrared signals that each has a unique ID and are 6 degrees apart from one another[18, 19, 20]. This method helps Kinect iden-

```csharp
using UnityEngine;
using System.Collections;

public class SampleClass : MonoBehaviour
{
    void Start()
    {
    }

    void Update()
    {
    }
}
```

1http://docs.unity3d.com/ScriptReference/MonoBehaviour.html
tify the distance between the players and the device. This method also enables the Kinect™ to identify the objects’ details such as hands, body, and even tiny objects by using Point Cloud. Because the price is more affordable than other devices, the Kinect™ is preferable for many users. Majority of the users well-aware of the newest version of Kinect, which was publicly released at the end of 2013 with key features’ improvements(Table 2.2).
<table>
<thead>
<tr>
<th>Feature</th>
<th>Benefits</th>
<th>Potential Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Improved skeletal, hand and joint orientation</td>
<td>With the ability to track as many as six people and 25 skeletal joints per person, including new joints for hand tips, thumbs, and shoulder center, and Improved understanding of the soft connective tissue and body positioning, you get more anatomically correct positions for crisp interactions and more accurate avateering and more lifelike avatars.</td>
<td>New and better scenarios in fitness, wellness, education and training, entertainment, gaming, movies, and communications.</td>
</tr>
<tr>
<td>Support for new development environments</td>
<td>New Unity support provides faster, cost-efficient, and high quality support for cross-platform development, enabling developers to build apps for the Windows Store by using tools you already know.</td>
<td>Build and publish apps to the Windows Store using tools you already know across multiple platforms.</td>
</tr>
<tr>
<td>Powerful tooling</td>
<td>With Kinect Studio’s enhanced recording and playback, developers can develop on the go, without the need to carry the Kinect sensor with them. And visual gesture builder lets developers build their own custom gestures that the system recognizes and uses to write code by using machine learning. These features increase productivity and keep costs down.</td>
<td>Increase your productivity and cost efficiency.</td>
</tr>
<tr>
<td>Advanced face tracking</td>
<td>With the resolution increased 20 times, applications can capture a face with a 2,000-point mesh that looks more true to life. This means that avatars will look more lifelike.</td>
<td>Build more lifelike avatars.</td>
</tr>
<tr>
<td>Simultaneous multi-app support</td>
<td>Improved multi-app support enables multiple applications to access a single sensor simultaneously.</td>
<td>For instance, by enabling a retail app and a business intelligence app access to the same sensor, you can get analytics in real time while customers are using a Kinect experience you have deployed in your store.</td>
</tr>
</tbody>
</table>

\(^2\text{http://www.microsoft.com/en-us/kinectforwindows/meetkinect/features.aspx} \)
2.5 Overview of 3D Modeling Software

A 3D modeling software such as Autodesk Maya, 3DMax, or Blender is required to create 3D models. The software is widely used in film and game industries. For example, Finding Nemo and Monster Inc. use these software to generate realistic 3D models and animation. In this project, we focus on using Maya to create 3D models because it is compatible with Unity 3D, and it supports 3D morphing (a special effect for animation that changes a 3D object into another) [21].
3.1 Education Design

The educational objective of this project is to improve the learning efficiency of students learning Biology. The students are placed into 2 different groups, and each group will learn educational materials in Biology courses using both the traditional teaching tool and our NUI-based system (Kinect). The assessment is measured by how well the students from each group perform on given exams.

3.1.1 Biology Materials

We plan to develop a set of Biology materials that can be divided into 4 modules. The first 3 modules focus on biological science while the last module focuses on biomedical engineering. Each module contains interactive visual objects. Those objects are created by using 3D modeling software. In the first module, users are able to explore components of living cell including nucleus, mitochondria, lysosome, endoplasmic reticulum, and plasma membrane. The second module contains both visual objects and 3D animation, which helps students learn and have better understandings about DNA. Lastly, the third and the fourth modules focus on visual objects associated with immune system and gene therapy, respectively.
3.1.2 Education Design

The software is designed to compare the learning outcomes between a group of students who use the traditional teaching tool and a group use the technological teaching tools (Kinect) (see Table 3.1). The first group uses the traditional teaching tools in the first and the second modules, then later uses the technological teaching tool (Kinect) in the third and the fourth modules. The second group begins with the traditional first, followed by technological. After each learning session, the students will be tested and their results will be averaged and compared using paired t-test.

Table 3.1: Experiment groups tested using the traditional verses the Kinect™ based teaching methods.

<table>
<thead>
<tr>
<th>Module 1 - Cell</th>
<th>Group 1</th>
<th>Group 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module 2 - DNA</td>
<td>Kinect™</td>
<td>Traditional</td>
</tr>
<tr>
<td>Module 3 - Immune System</td>
<td>Traditional</td>
<td>Kinect™</td>
</tr>
<tr>
<td>Module 4 - Gene Therapy</td>
<td>Traditional</td>
<td>Kinect™</td>
</tr>
</tbody>
</table>

3.2 System Design

For the system design, we have chosen specific hardware and software to use and implement. We divide consider our design into three parts, which are: system architecture, NUI, and record keeping. System architecture is about the big picture of our system, whereas NUI contains specific details about implementation. In addition, we designed the record keeping in order to track users when they access the system.
3.2.1 System Architecture

In order to share as much programming codes as possible among difference course modules, we proposed a layered architecture (as shown in Figure 3.1), for the Kinect™ educational system. In the hardware layer, the Kinect™ device will support gesture-based interactions. The animations will be implemented in real-time, a fast graphics card has been chosen for this project. Therefore, laptop with Intel Core™ i5-3230M processor and a 1 gigabyte graphics card will support the modelling as well as the rendering of animations. The Base layer provides the functions that are common to all applications regardless of the course content. These functions can be used for future development without altering the current course modules that are planned. The Biology layer provides the functions that are specific and common to this proposal, which include Cell, DNA, Immune System, and Gene Therapy modules. Unfortunate, some of the functions that are specific to each individual course module cannot to be shared. However, modular programming will be implemented to minimize these non-shared functions.
3.2.2 Natural User Interface

Natural User Interface (NUI) and human gesture recognitions have been developed over the last decade. They are used to develop human user interface, which allows the user to interact with the computer more naturally in a seemingly natural way[11]. Using NUI and human gestures, the users are able to directly interact with the computer without using a mouse or a keyboard. In this project, we focus on designing Kinect™ to capture the users’ movement. This is proven to be beneficial due to the lower cost and the better quality, when compared to other sensing devices on the market. Kinect™ allows the users to visually interact with educational contents.

In regard to the game engine, we use Unity3D because it is compatible with many kinds of platform, and it supports script languages such as C# and JavaScript. Fig-
Figure 3.1 illustrates the architecture of the Kinect™ based system. Users’ movements are captured by the Kinect and sent to the game engine through the Application Programming Interface (API). We have chosen Open Natural Interaction (OpenNI)[22] and Zigfu. OpenNI is a multi-language open source framework. It is widely used for creating software, which take advantage of natural interaction. Zigfu is a development kit for making motion-controlled software with Kinect™[23]. It supports Unity3D, HyperText Markup Language (HTML), and Flash Player.

In order for users to connect with the system. They must have Kinect™, which is a motion sensing device, used to track the users’ movement. It consists of an RGB camera, an infrared emitter and an infrared (IR) depth sensor. The RGB camera is used to capture color images. The infrared emitter and IR depth sensor are both used to capture in-depth images. The process starts from the emitter emits infrared light beams. The IR depth sensor receives IR beams, which reflect back to the sensor. Then, the reflected IR beams are converted to in-depth image as shown in Figure 3.2b. Moreover, Kinect™ can be used for skeletal tracking (Figure 3.2a). It can track up to twenty points of the human body (Figure 3.2c). This provides us with joint orientation information for skeletons. In our system, we choose to use Kinect™ with game engine (Unity3D). All virtual objects will be imported to Unity3D and displayed such objects on the screen. Thus, the users will be able to interact with any virtual object by performing hand gestures with Kinect™ and Unity3D. Additionally, the system will support only single user even though Kinect™ can capture multiple users at the same time. This is because of space limitations and
the systems difficulty in distinguishing hand gestures.

3.2.3 Record Keeping

We integrated SQLite with our system. SQLite is database management system. The SQLite database consists of five tables: Login, UserInfo, SessionLogin, ModuleLogin and Quiz (Figure 3.3). The login table stores a username and password for each user. The UserInfo table stores user information (for instance, gender and race). We decided to collect usage time for each user by storing usage time in two tables, which are the SessionLogin table, and the ModuleLogin table. Every time users login to use the program, it will generate a new sessionID. The sessionID can show us how long users spend utilizing the program during each login. Furthermore, we also collect usage time for each user within each module. This can help us to analyze the diversity of the as well as their primary reasons of usage.
3.3 Usability Design

User Interface (UI) design is a method that allows the users to interact with the computer, and it is a summary of the interaction design, visual design, and information architecture[10]. It connects the users to various software applications to perform special tasks. UI design covers a wide range of products including software applications, computers, cars, planes, and mobile communication devices. UI is able to provide users with task methods, processes the interactions, and displays the results through simple patterns. The complexity of UI varies depending on the users’ requirements such as language, pattern, and layout. It is very important to design a good UI in order to fulfill the users’ requirements. The users are required to have a good understanding of UI in order to increase the quality of the User Experience (UX). UX involves personal behavior, attitudes, and program experiences[24]. It may be
stated as users’ personal perception from the results of product usage, and UX and UI generally need to be in the same direction in terms of project efficacy. Generally in terms of projects efficacy UX and UI need to go along together.

3.3.1 User Interface Design

The system we create focuses on the design of the UI and how it will help users understand their task method. We designed two controller bars, which are located on the left and the right hand sides of the scene (Figure 3.4) to help explain the task. The left one consists of 3 buttons including reset, text, and main menu. The reset button is used to reset an object (on a scene) to the original orientation and scaling. The text button is used to enable any texts that belong to an object. Those texts describe the structures and functions for each individual. The main menu button is used to navigate a user back to the main menu scene. Additionally, the right controller bar consists of two groups of buttons. The fist group contains 3 buttons including x-axis, y-axis, and z-axis. These buttons allow users to rotate the visual object on the scene in any directions. The second group contains two buttons including zoom-in and zoom out buttons. These allow the users to scale the visual objects on the scene.

3.3.2 Interactivity

One of the objectives of this project is to design a UI that has good UX for the users. The interaction between the users and UI elements is an important factor for the system. We have carefully designed two methods of UI interaction. The first
one is hand pushing method. This allows users to interact with the UI elements by moving one of their hands to the same x and y coordinates with one of the UI elements. The users need to push their hands forward in order to complete the demand. For the second method, it is hand holding method. We found that many game software allows the users to interact with any of the UI elements by moving their visual hands on the software to the same x and y coordinates with the target UI element and holding those hands in the area of the target UI element for a specific period of time. Figure 3.5 illustrates an example of a game called Kinect Adventure, which requires the users to complete the task as described previously in the second method. The users are asked to use both of the methods, and we will take the time length they use to complete each specific task and the numbers of success rates for both methods into
an account (in order to determine which is the best method). In order to measure which method is better for the users, we will provide them access to both methods. During each session, we will record how much time each individual spends on each specific task, and how many success rate are archived for both methods. The times will be averaged and compared by using the efficiency metric process.

3.3.3 Hand Gestures

Not only use the variety kinds of button to interact with the virtual objects, but the users can also interact with the virtual objects by using their hands, which are connected to the visual hands on the screen, to interact with the visual objects. There are two operations (scaling and rotating), that users can possibly perform with the objects. The system need to be able to immediately recognize the users’ needs. Thus,
the hand gestures techniques have been added into the system. We have designed the system, so that it can recognize the 2 majority movements including scaling and rotating the visual objects, which are the most natural gestures that are used on a users’ daily basis.

3.3.4 Mode Switching

The system we use allows the users to freely use their hands to interact with the system without using a mouse or a keyboard. What we found when we first developed a prototype of the system with Kinect was that the users might have accidentally moved their visual hands over GUI elements. The GUI elements acted like buttons when they were trying to manipulate the visual objects. Then the button navigated the user to other scenes and even performed unauthorized actions. This could prevent the system from successfully operating. In order to collect this problem, we constructed hand gesture recognition techniques to help the users manage their commands (whether they want to choose visual object manipulation or to click on a GUI element). After observing the usage of the system prototype, we noticed that the user could be separated into two main tasks; including GUI and visual object interaction (Figure 3.6). Therefore, we implemented two new methods. These methods should reduce the accidental navigation conflict. The users are required to shake their head to change from the GUI interaction to the visual object interaction in the first method. In the second method, users only need to place both of their visual hands in the same x and y coordinates to manipulate the visual objects, and drop
Figure 3.6: Two important user tasks in the system.

their hands to switch to GUI manipulation. After we have completely implemented this, we will test both methods and then we will analyze the data by using a usability metric process called level of success (Table 3.2). The level of success will show us which method is best suited for the system.

Table 3.2: Example: Level of success by task for measuring both mode switching methods.

<table>
<thead>
<tr>
<th>Users</th>
<th>Task 1</th>
<th>Task 2</th>
<th>Task 3</th>
<th>Task 4</th>
<th>Task 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>No problem</td>
<td>No problem</td>
<td>No problem</td>
<td>Some Problem</td>
<td>No problem</td>
</tr>
<tr>
<td>2</td>
<td>Some problem</td>
<td>No problem</td>
<td>Some Problem</td>
<td>Failure</td>
<td>No problem</td>
</tr>
<tr>
<td>3</td>
<td>Failure</td>
<td>No problem</td>
<td>Failure</td>
<td>No problem</td>
<td>Some Problem</td>
</tr>
<tr>
<td>4</td>
<td>No problem</td>
<td>No problem</td>
<td>Failure</td>
<td>Failure</td>
<td>Failure</td>
</tr>
<tr>
<td>5</td>
<td>No problem</td>
<td>No problem</td>
<td>No problem</td>
<td>No problem</td>
<td>Some Problem</td>
</tr>
</tbody>
</table>
4.1 Education Component

As we previously discussed in Chapter three, our system consists of four modules (cell, DNA, immune system and gene therapy). Each module has its own set of interactive virtual objects. Most of the virtual objects are created by using 3D modelling softwares such as Autodesk Maya. Only the living cell model (in the cell module) was taken from TurboSquid (a digital media company that sells stock 3D models used in 3D graphics to a variety of industries). These virtual objects will be imported to Unity3D.

4.1.1 Biology Contents

Autodesk Maya (Maya) is a tool that we selected in efforts to create the interactive virtual objects. Maya is a 3D modelling software that is beneficial because it enables us to create impressive and realistic 3D models. Basically, to create a 3D virtual object, we must first create a Polygon Primitive, and then transform it to other shape by using variety tools from Maya. In Maya, there are many Polygon Primitives, which we can use such as spheres, cubes, cones and etc (Figure 4.1). Additionally, there is an attribute editor section (Figure 4.2) for all Polygon Primitives that are
created with in this scene. This attribute editor offers seemingly better control on how the primitive appears in the scene (such as position and size). In addition to attribute editor, Maya also provides Object Component Mode (which allows us to manipulate vertices, faces edge and etc). To access the Object Component Mode (Figure 4.3), we need to select any objects on the scene and hold the the right mouse button. This mode is very useful because it allows the user to transform a simple primitive object into another object, by adjusting the object vertices and faces. Figure 4.4 is an example on how to create a 3D model, which is consists of many polygons that transformed from the Polygon Primitives. When we finish creating the virtual objects, we will import to Unity3D, by using Filmbox file format (fbx).
Figure 4.1: Polygon primitives in Maya.
Figure 4.2: Attribute editor in Maya.

Figure 4.3: Object Component Mode in Maya.
Figure 4.4: A living cell model in Maya.
4.1.1.1 Assets

Four educational modules have been selected to implement into the system. The modules are based upon Biomedical Engineering which are Cell, DNA, Immune System and Gene Therapy. All assets were constructed by using 3D modelling software and imported to Unity3D.

For the first module (Cell), students will be able to explore components of a living cell such as mitochondria, lysosome, endoplasmic reticulum, plasma membrane, nucleus and etc. Figure 4.5, 4.6 and 4.7 represent all assets for the first module.

(a) Cell.  (b) Nucleus.

Figure 4.5: Assets for the cell module.
Figure 4.6: Assets for the cell module.
(a) Nuclear Pore.
(b) Cell Membrane.
(c) Level detail of Nuclear Pore.
(d) Sodium-Potassium Pump.
(e) Lipid

Figure 4.7: Assets for the cell module.
The second module is about DNA. Students will be able to learn about: the components of DNA, mechanism of DNA translation as well as DNA transcription. For the second module assets, some of them from the first module can be reused to demonstrate specific areas in relation to DNA, such as nucleus. DNA is packaged into chromosomes which can be found inside the cell nucleus. We reused some assets from the first module, and proceed by constructing eight 3D virtual objects, which are shown in Figures 4.8 and 4.9.

![Figure 4.8: Assets for the DNA module.](a) Chromosome and detail. (b) Chromosome.)
Figure 4.9: Assets for the DNA module.
The immune system is the third module. Thirteen virtual objects were created and used as assets for this module. All of the virtual objects for this module have a significant amount of detail that needed to consider making them as close to reality as possible. This factor created the most challenge among three other modules. For example, to create retrovirus model (Figure 4.10b), we had to pay close attention to every detail within it. These details ranged from texture mapping to the finer detail inside the virus. All assets for the third module are shown in Figures 4.10, 4.11 and 4.12.

(a) Adenovirus.  (b) Retrovirus.

Figure 4.10: Assets for the immune system module.
Figure 4.11: Assets for immune system module.
Figure 4.12: Assets for the immune system module.
The last module is gene therapy. Students will be able to design, optimize, and test their delivery vehicles on various conditions. They will have choices of retrovirus, adenovirus and non-viral vectors (DNA, DNA-LPEI, nanoparticle containing DNA, and nanoparticle containing DNA-LPEI). For the module assets, some of the assets from the first, second, and third module can be reused, and only seven 3D virtual objects were created, which are shown in Figures 4.13 and 4.14.

(a) Nanoparticle containing DNA.  
(b) Nanoparticle containing DNA-LPEI.

Figure 4.13: Assets for the gene therapy module.
Figure 4.14: Assets for the gene therapy module.
4.1.1.2 Animation

In addition, to the exploration of virtual objects with hands (for scaling and rotation), we implemented 3D animations for each module, as shown in Table 4.1. These animations will help students to better understand the contents over the use of virtual object manipulation (scaling and rotation). Key frame animation and morphing techniques were both used to construct the animations. These techniques will be discussed in the system implementation section.

Table 4.1: All animation scenes in the system.

<table>
<thead>
<tr>
<th>Module</th>
<th>Animation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell</td>
<td>Lipid in water</td>
</tr>
<tr>
<td></td>
<td>Sodium-Potassium pump</td>
</tr>
<tr>
<td>DNA</td>
<td>transfer RNA</td>
</tr>
<tr>
<td></td>
<td>DNA translation</td>
</tr>
<tr>
<td></td>
<td>DNA transcription</td>
</tr>
<tr>
<td>Immune system</td>
<td>Immune system</td>
</tr>
<tr>
<td></td>
<td>Macrophage</td>
</tr>
<tr>
<td></td>
<td>T-helper cell</td>
</tr>
<tr>
<td></td>
<td>B cell</td>
</tr>
<tr>
<td></td>
<td>Retrovirus</td>
</tr>
<tr>
<td></td>
<td>Adenovirus</td>
</tr>
<tr>
<td>Gene therapy</td>
<td>Introduction</td>
</tr>
<tr>
<td></td>
<td>Retrovirus</td>
</tr>
<tr>
<td></td>
<td>Adenovirus</td>
</tr>
<tr>
<td></td>
<td>DNA</td>
</tr>
<tr>
<td></td>
<td>DNA-LPEI</td>
</tr>
<tr>
<td></td>
<td>Nanoparticle containing DNA</td>
</tr>
<tr>
<td></td>
<td>Nanoparticle containing DNA-LPEI</td>
</tr>
</tbody>
</table>
4.2 System Implementation

In this section, it is consisted of Unity3D with Kinect™ and animation. The first one is about how to implement Unity3D with Kinect™. For the animation, it is separated into key frame animation technique and morphing animation technique.

4.2.1 Unity3D with Kinect™

We implemented Kinect on Unity3D by using the Zigfu unity bindings, (which comes with OpenNI). In order to use Kinect™ in our project, we first had to install Zigfu unity package in our project. Zigfu provides us with many useful components (as shown in Table 4.2). In our program, we first mapped all 20 joints of the user to the ZigSkeleton component. We found that this was unnecessary since users only need their hands to interact with objects within each scene. We mapped the ZigSkeleton with only hands of the user. Next, we debated about how many people can use the system at once. We decided to make the limit one user at a time. We found out, when there were multiple users at the same time, the system required much more space.

Next, we considered having only two parts of the human skeletons (left and right hand) to be used in our system since we can specify hand movements as gestures in order to control and interact with the software. Firstly, virtual hands were created by using Autodesk Maya (Figure 4.16). Then, we imported the model into Unity3D and connected it to the ZigSkeleton component from Table 4.2. The ZigSkeleton component provides public variables, which mostly relate to skeleton’s joints. These are
Table 4.2: List of some useful components from Zigfu’s Unity package.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zig</td>
<td>Define the Zig object and all binding we need for Zigfu. Detect input type for Kinect driver such as OpenNI, KinectSDK and OpenNI2</td>
</tr>
<tr>
<td>ZigDepthViewer</td>
<td>Render a depth map image from Kinect on a scene.</td>
</tr>
<tr>
<td>ZigImageViewer</td>
<td>Render a RGB image from Kinect on a scene.</td>
</tr>
<tr>
<td>ZigEngageAllUsers</td>
<td>Detect all users that stay in front of the Kinect camera.</td>
</tr>
<tr>
<td>ZigEngageSingleUser</td>
<td>Detect only one and first user that stay in front of the Kinect camera.</td>
</tr>
<tr>
<td>ZigUsersRader</td>
<td>Capture users’ position in the users world coordinate.</td>
</tr>
<tr>
<td>ZigSkeleton</td>
<td>Mapping user skeletons to any related objects on a scene in Unity3D.</td>
</tr>
</tbody>
</table>

captured through the Kinect™ camera. As a result from Figure 4.17, the left hand and right hand public variable have pointers connect to the model and other variables do not. Also, other values from the ZigSkeleton, which are rotation dumping value and scale value, need to define properly. Rotation damping is a value for adjusting the retardation between actual hands movement speed and program simulation hand from movement. The scale value is for adjusting boundary of the virtual hands. Consequently, rotation is adjusted to 30.0 to make the hands model move smoothly while users move their hands in front of the Kinect™ camera. Also, scale value is defined as: \( x = 0.025, y = 0.025 \) and \( z = 0 \). These values allow the hand movements to fit with 800x600 screen size.
Figure 4.15: A user interact with the system.

Figure 4.16: 3D hand models.
Figure 4.17: ZigSkeleton public variables.
4.2.2 Animation with Unity3D

Computer animation has been widely used in the movie and video game industries. It is used to create the illusion of movement. In this system, we use 3D animation to demonstrate biological contents such as the immune system and gene therapy. There are two animation techniques that we implemented: key frame animation and morphing.

4.2.2.1 Key Frame Animation

One of the most famous technique used for creating computer animation is key frame animation technique. This technique is valuable for our project, because it enabled us to make the spectacular animations of various objects. There is a timeline for a whole animation. Along the timeline, there are at least two key points need to be defined, which are starting point and ending point. Also, other key points can be added as many as possible between the starting and the ending point. For each key point, any public variables, that belong to the object, can be modified, and the key frame animation will generate smooth transition for the values between each key points. For our implementation in Unity3D, we use a component called Animation to play back animations that we created. We can assign animation clips, which store the sequences of animations, to the animation component. In order to create the animation on an animation clip, Unity3D provides us with a timeline (Figure 4.18). On this timeline, we can add key points as much as we want. For each key point, we can select object’s values and modify them along the timeline. Unity3D
will automatically generate a smooth transition between each key point. In addition, Unity3D provides animation event that lets the program call a script function like sending a message. The animation event can be added any where on the timeline similar to key point.

4.2.2.2 Morphing

Morphing is a technic that allows us to change a mesh or part of a mesh from one shape to other shapes through a seamless animation. It is always used for facial animation. When we started to implement animations for immune system, we found that certain parts of the animation needs to deform an object’s mesh. Thus, we determined morphing would be needed. Hence, MegaFiers have been chosen to support morphing technique. It is a mesh deformation and animation system written in C# for Unity3D. MegaFiers has around fifty modifiers that we can use to deform any mesh. It is convenient to use as well. First of all, we used Maya to create a
deformable object. In order to create it, we duplicated an original object several times. Then, we used tools in Maya to modify the shape of the duplicated objects to any shapes we desired (Figure 4.19a). Next, we used blend shape deformer which let us change the shape of the original object to the shapes of other duplicated objects. After that, the models will be imported Unity3D. In Figure 4.19b, We can deform the model to the desired shapes in Unity3D through key frame animation’s timeline.

Furthermore, in Figure 4.20, this is another example how we use morphing in our project. In this case, we need to show lipid bilayer form an endosome. We
decided to use the Bend modifier, which is one of the modifiers provided by Megafiers. With this modifier, we can attach it to any object and it will produce a uniform bend in an object’s mesh. We wrote a script that is used to combine morphing with key frame animation. This component allow us to control how the mesh deforms over the timeline of key frame animation.

4.3 Usability Implementation

In this section, we will explain how we implement the usability part. It consists of interaction with the user interface element, and hand gestures for interaction with the virtual objects.

4.3.1 User Interface Interaction

Without using a mouse and keyboard in the system, users can interact with the computer easier than before. In our system, we developed a UI element by
using 3D modelling software and import these into Unity3D. These UI elements can be accessed with the virtual hands, which are controlled by the user. In order to interact with UI, the connection between the user and software (Kinect) is required. After standing in front of the Kinect’s camera, the program will start to recognize the user’s position, and then map their hands to the virtual hands in the system. When user’s hands move, the virtual hands in program will change their position and mimic the user’s hands. As we mentioned in Chapter 3.3.2 (user interface interaction), we considered developing two kinds of methods for interacting with the UI element, which are pushing hand and holding hand method. After we finish developing these methods, we will test users both method and find the best suited method for the system. Thus, in the next section, we will explain more about how we develop them.

4.3.1.1 Hand Pushing

Zifgu provide components that can help us detect hands movement such as PushDetector, and SwipeDetector. PushDetector is used to detect the user’s hands when the user push one of the hand in forward direction, where as SwipeDetector is used help us to detect when the user swipe on of the hands to from the left to the right, or the right to the left. For this scenario, we used ZigPushDetector component to detect the hand pushing movement from a user. This component will broadcast a message whenever the user push his/her hand in the forward direction. Basically, there are a total of three types of message (which are “push”, “release” and “click”) that will be sequentially sent by ZigPushDetector. Therefore, we implemented a
component in Unity3D called COHandPushing. This Component will be used as an event listener for receiving the message from ZigPushDetector component. This message will be used to activate any of the UI elements on the scene.

4.3.1.2 Hand Holding

For interaction with any UI elements, users need to control the virtual hands. When one the virtual hand is over any of the UI elements, small dots around the hand (Figure 4.21) will appear. These dots indicate that the user is interacting with the UI Element. This process can be divided into 3 states; starting point (Figure 4.21a), in-between (Figure 4.21b), and ending point (Figure 4.21c). These states take place as a sequence. The starting point happens when a virtual hand has begun touching a UI Element. Then a red dot will appear in the next render loop. The in-between point occurs when the virtual hand is touching the UI element. In this point, the red dots will appear until they surround the virtual hand in the shape of a circle. Then, the ending point will be introduced. It indicates that user successfully performed UI interaction with the UI element.

To implement this method, we need to include the MessCollider component into both virtual hands and UI Element. The reason we need the MessCollider component is because it provides call back function as shown in Table 4.3. These call back functions will be called whenever any virtual objects (that have MessCollider component) collide to each others. For the call back function as show in Table 4.3, we need to write C# script to control and smoothen the processing process. Figure
4.22 shows an example of a code that we used to handle when collision appears.

Table 4.3: Lists of callback function for collision detection.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void OnCollisionEnter</td>
<td>This function is called when object that has collider or rigidbody components has begun touching another collider or rigidbody.</td>
</tr>
<tr>
<td>(Collision collision)</td>
<td></td>
</tr>
<tr>
<td>void OnCollisionStay</td>
<td>This function is called one per frame when any collider/rigidbody is touching another collider/rigidbody.</td>
</tr>
<tr>
<td>(Collision collision)</td>
<td></td>
</tr>
<tr>
<td>void OnCollisionExit</td>
<td>This function is called when a collider or rigidbody has stopped touching another collider/rigidbody.</td>
</tr>
<tr>
<td>(Collision collision)</td>
<td></td>
</tr>
</tbody>
</table>

To examine the code in Figure 4.22, the process starts from a collision between any virtual objects that have the MessCollider component. In this circumstance, one of the virtual objects must be a virtual hand, and the other should be any UI element.

After the collision between the virtual objects happen, the OnCollisionEnter function will be called. We write the code in this function to investigate the virtual objects, which collide with each other, whether one of the objects is a virtual hands or not. If it is one of the virtual hands, the color of the virtual object(UI element), which collide with the virtual hand, will be changed in order to help the users recognise that
void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.tag == "hand")
        gameObject.renderer.material.color = Color.white;
}

void OnCollisionStay(Collision collision)
{
    if (collision.gameObject.tag == "hand") {
        gameObject.renderer.material.color = Color.red;
        timeCount += Time.deltaTime;

        if (timeCount >= timeStop) {
            timeCount = 0;
            isHit = true;
        }
    }
}

void OnCollisionExit(Collision collision)
{
    if (renderer.enabled == false) {
        return;
    }

    gameObject.renderer.material.color = Color.white;
}

// Update is called once per frame
void Update()
{
    if (isHit) {
        //do something
    }
    isHit = false;
}

Figure 4.22: Example code for callback function.
they already start interacting with the UI element. After that, the OnCollisionStay function will be called once per frame. This function is used as a timer function since it is automatically called by the system as long as the virtual hand continues to keep touching to the UI element. We write a code snippet in this function to measure how long the virtual hand has been touching with the UI element. When the time meets the condition the function, the UI element will do what it need to do such as navigating the users to other parts of the system.

4.3.2 Hand Gestures

Since we designed that the users can interact with the virtual objects by using their hands, we developed hand gestures technique for our system. The gestures recognize what users want to interact with the virtual objects. In this system, there are three types of gestures that users can perform: scaling, rotating and stopping object manipulation mode.

The first one is about scaling the virtual objects. In order to scale the virtual objects on the screen, users firstly need to move the virtual hands to the same x and y coordinate with the virtual objects. Then the system will perform ray-casting (will be discuss more in section 4.3.3.2). After that, users can perform scaling the virtual objects. To zoom in, users must move their hands far away from each other (Figure 4.23a). To zoom out, users must move their hands close to each other (Figure 4.23b). Therefore, we need to find out the distance between both hands. Assume that left virtual hand is at x1, y1, z1 and right virtual hand is at x2, y2, z2 on the system.
world coordinate. The distance between \((x_1, y_1, z_1)\) and \((x_2, y_2, z_2)\) is given by equation 4.3.2.

\[
d = \sqrt{(\Delta x^2) + \Delta y^2 + \Delta z^2} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \quad (4.3.2)
\]

This formula is derived from Pythagorean theorem. It helps us find the distance between both stimulate hands. Hence, the more distance between the stimulate hands the bigger virtual objects are, whereas the less distance the smaller virtual objects are.

In addition to zoom in/out on the virtual objects, users can also rotate the virtual objects. This allows users to clearly see the virtual objects from any sides. Thus, to rotate, users must move their hands in opposite directions (Figure 4.24). For example, left hand up and right hand down will rotate counter clockwise to direction of hands movement. The speed of the hands also play a role. The user’s hand speed determines how fast the 3D objects should rotate or scale. As we store the position
Figure 4.24: Poses for objects rotating.
of hands by using Vector3 in unity3D, the speed is given by equation 4.3.2. Since
users need to operate the rotation by using both hands, we need to use the average
speed for both hands.

\[ v = \frac{\sqrt{\Delta x^2 + \Delta y^2 + \Delta z^2}}{\Delta \text{time}} \quad (4.3.2) \]

In unity3D, every object in the scene always has a component called “Transform”. Basically, it is used to store the position, rotation and scale of each virtual
object. When the system has the designed rotation and scale value from calculation,
these value are applied directly through the Transform component that attach to the
virtual objects on the screen.

Lastly, as users can determine what they want to do between UI interaction
or 3D object manipulation(Figure 3.6 ). We designed the system in such a way
that whenever users drop their hands (Figure 4.25a), they are no longer in object
manipulation mode as it shown in Figure ??.

4.3.3 Switching Mode

. When users access the system, they can use the virtual hands to scale and
rotate the virtual objects. Futhermore, they can also use their hands to interact with
the UI element such as a button. Therefore, as we discussed in chapter 3.2.2(Natural
User Interface), we developed two methods ,which are head shaking technique and
hand dropping technique. These two methods are useful to determining what users
are willing to do between object manipulation and UI interaction(Figure 3.6). The
next section is about how to implement these methods in Unity3D.
4.3.3.1 Head Shaking

Firstly, we developed a system, which enables users to switch how they want to interact with the system by shaking their heads a bit. Thus, whenever they shake their heads, the system will recognize this behavior and change the mode between UI interaction and object manipulation. In order to do this, we take advantage of the Kinect™ camera, which can capture the movement of users when they are in front of the camera. Normally, when the Kinect™ camera captures user movement, the rotation of the user’s head should be between 0 - 5 degree on the right hand side (clockwise) and 360-355 degree on the left hand side (counter clockwise). If the Kinect™ camera captures the rotation of users’ head different than stated, we determine that users probably tried to shake their heads. In Figure 4.27, this is the implementation for this method in C# is shown. As we mention in chapter two table 2.1, the update function is called every frame. If the angle of the user’s head meets the condition (more than 5 degree or less than 355), the system will determine that the users went to switch the mode between UI interaction and object manipulation by shaking their heads.

4.3.3.2 Hands Dropping

By observing the users when they tested our software (during implementation), we noticed that whenever users interact with 3D objects, their hands are usually height between their shoulders and chest region (Figure 4.25b). More over, the virtual hands in the software are in the same x and y region with the virtual
object in the scene. Therefore, to determine whether the user wants to interact with the UI elements or 3D objects, we developed ray-casting technique, which is one of the Computer Graphic algorithms that issued to solve a variety of solutions [25, 26]. It is obviously used in shooting games. Firstly, an invisible ray is emitted from a point in a specific direction. Then, the ray will travel towards the ray direction and detect whether any collider in the path of the ray.

In Unity3D, ray-casting is defined as a static function in Physics component. In order to create a ray cast in Unity3D, we must first create a new component or script, where we can define all information for the ray cast. We created a component called COHandRayCast. This component was attached to both stimulate hands in order to detect 3D objects which are parallel to the hands in the x and y axis. Basically, there is important required information, as shown in Table 4.4, that we need to setup for generating the command.

Figure 4.25: A user is interacting with the system.
Table 4.4: Detail about ray-casting function in Unity3D

Raycast(Vector3 origin, Vector3 direction, RaycastHit hitInfo, float distance = Mathf.Infinity, int layerMask = DefaultRaycastLayers);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>origin</td>
<td>The starting point of the ray in world coordinates.</td>
</tr>
<tr>
<td>direction</td>
<td>The direction of the ray.</td>
</tr>
<tr>
<td>distance</td>
<td>The length of the ray.</td>
</tr>
<tr>
<td>hitinfo</td>
<td>If true is returned, hitInfo will contain more information about where the collider was hit (See Also: RaycastHit).</td>
</tr>
<tr>
<td>layerMask</td>
<td>A Layer mask that is used to selectively ignore colliders when casting a ray.</td>
</tr>
</tbody>
</table>

For our system, we determined to use ray-casting technique to detect whether both stimulate hands are in the same x and y regions along with the 3D object in the 3D world coordinate. Thus, both virtual hand positions are used as the starting points for the rays.

Vector3 origin = hand.transform.position

The next step is to set up the direction of the rays. In our system, both stimulate hands are in the z-axis, which is always equal to 0, and all 3D objects have the minimum z value of less than 0. Therefore, the rays direction are defined by the negative z-axis.

Vector3 direction = new Vector3(0,0,-1.0f)

The ray distance is properly defined as 100. In addition, a layer mask is used to selectively ignore colliders, which are not targeted objects when casting a ray. In our system, we defined the layer mask for the targeted objects as eight so the rays will ignore the colliding objects that have layer masks less than eight.
Then, an “if statement” is used to determine whether the ray hit the objects or not. If it hit any targeted colliding objects, the Physics.Raycast function will return “true” as the code for COHandRayCast, shown in Figure 4.26. Therefore, we use ray-casting to detect stimulate hands. If both rays that emit from the stimulate hands collide with any 3D objects, users can manipulate those objects by using their hands.
Figure 4.26: Example code for CORayCast.
```c
void Update() {
    if (skeleton.Head.rotation.eulerAngles.z > 5.0f
        && skeleton.Head.rotation.eulerAngles.z <= minThresholh) {
        isInAction = true;
        timeCount += Time.deltaTime;
    }
    else if (skeleton.Head.rotation.eulerAngles.z < 355.0f
             && skeleton.Head.rotation.eulerAngles.z <= maxThreshold) {
        isInAction = true;
        timeCount += Time.deltaTime;
    }
    else {
        if (isInAction) {
            if (timeCount <= actionTime)
                //Head shaking
                //Do something
        }
        else {
            //No head shaking
        }
        isInAction = false;
        timeCount = 0;
    }
}
```

Figure 4.27: Example program for capturing the head shaking.
CHAPTER V
RESULT ANALYSIS

5.1 System Validation

For the development methodology, we used the waterfall model, which is a sequential process. It consists of the following software development stages:

- Requirement specification
- System architecture design
- Implementation/Coding
- System testing and debugging
- System verification
- Bugs fixing
- System installation

Typically, each stage finishes before the next stage can begin. There are many advantages for waterfall model. Firstly, developers can design and make plans for how the system should function. The developers and customers need to agree on the system requirements in the early stages to insure satisfaction. Secondly, output is generated after each stage, so it is easy to manage the project. For example, after
completion of the third stage (implementation/coding), a beta version of the system is the output for this stage. These are the reasons why we choose waterfall model for the system development.

In addition, system testing and verification are important procedures. In order to distribute the system to the clients, it must first pass the testing and verification process. In order to do that, we have student volunteers from the Computer Science Department at The University of Akron help us test and verify the system. During the testing stage, several errors were detected. Then, we fixed all of the errors and tested it again (until there is no error left).

5.2 Educational Results

Two groups of students at the North Akron high school participated in the study of the four educational modules (Cell, DNA, Immune System, and Gene Therapy) and were tested after each educational session. We expect the group using the Kinect™ based system will not perform better with the technology-enhanced teaching method (Table 3.1). The preliminary results (Table 5.1) shows that the group of students who learned with the Kinect™ based system had the mean quiz score of 16.32 out of 20 while the group of students who learned with the traditional method had the mean quiz score of 14.04 out of 20.

The results from t-test (Table 5.2) show that the group of students who learned with the Kinect™ based system had the higher quiz score than the group of students who learned with the traditional method (Mean difference of 2.28). The
p-value is less than 0.05. This means that the difference is statistically significant at a 0.05 level (95% confident level) \( t = 6.363, p\text{-value} = 0.000 \), so we can accept the Hypothesis that visual information (in visuo-spatial sketchpad), along with integrated gestures, result in more accurate retrieval of the contextual information. We can conclude that the Kinect\(^\text{TM} \) based system has a beneficial affect on learning.

Table 5.1: Paired samples statistics.

<table>
<thead>
<tr>
<th></th>
<th>Mean</th>
<th>N</th>
<th>Std. Deviation</th>
<th>Std. Error Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kinect Score</td>
<td>16.32</td>
<td>25</td>
<td>3.05123</td>
<td>.61025</td>
</tr>
<tr>
<td>Control Score</td>
<td>14.04</td>
<td>25</td>
<td>2.65330</td>
<td>.53066</td>
</tr>
</tbody>
</table>

Table 5.2: Paired samples test.

<table>
<thead>
<tr>
<th></th>
<th>Paired Differences</th>
<th>95 % Confidence Interval of the Difference</th>
<th>t</th>
<th>df</th>
<th>Sig. (2-tailed)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Sd.</td>
<td>Std. Error Mean</td>
<td>Lower</td>
<td>Upper</td>
</tr>
<tr>
<td>Kinect Score - Control Score</td>
<td>2.28</td>
<td>1.79</td>
<td>.35833</td>
<td>1.54044</td>
<td>3.01956</td>
</tr>
</tbody>
</table>

5.3 System Performance

Laptops with Intel Core\(^\text{TM} \) i5-3230M processor and MacBook Pro 2011\(^\text{TM} \) were selected to test the performance of the system. These laptops use Windows as an operating system, whereas MacBook Pro 2011\(^\text{TM} \) uses OS X as an operating system. We do not compare these two hardware in term of how fast they are. We mainly compare how much time(in percentage) is spent on each function within the system.
Thus, Unity Profiler, which is a built-in tool for Unity3D, is introduced to measure the performance of the system on the different platforms. This tool keeps record of how much time is spent on each part of the system for both CPU and GPU. It also reports how much memory is used. Figure 5.1 is an example of the Profiler window.

Thus, two different scenes from the system were selected for testing. The first one is the living cell from the cell module. This scene has totally 485,300 triangles and approximately 441,200 to 445,000 vertices. These data have high impact on the performance because all of them are sent to CPU and GPU for rendering. The second one is beta sheet from the immune system module. This scene has less detail comparing to the first one. It has only 56,300 triangles and approximately 55,800 vertices. We decided to take a snapshot every thirty frames, since Unity Profiler captures every rendering frame. CPU usage time, GPU usage time and memory
usage are represented on each snapshot. The exact time that is spent on all specific game codes is represented as well. Since we test our system performance on both Mac OS X and Microsoft Windows, we cannot compare the results from both platforms in term of CPU usage time because of the different hardware specifications. Thus, we will look beyond CPU usage into the specific program codes.

5.3.1 Performance on Microsoft Windows

The results from Table 5.3 shows CPU usage for every captured frame from both selected scenes in millisecond. The average CPU usage for scene one (the living cell) is 25.16 milliseconds. Similarly, the average CPU usage for scene two (the beta sheet) is 24.97 milliseconds. Scene one requires rendering more triangles than scene two, so this result was surprising. Let take a look at Figure 5.2. It represents the most time consuming functions for all captured frames. ZigInput.Update function seems to consume most CPU usage, followed by WaitForTargetFPS, and COCheck2Hands.Update functions. WaitForTargetFPS is a function provided by Unity3d to control the refresh rate or frame per second. It is not a function that we can rewrite or attempt to improve the performance since it is provided by Unity3D. Whereas ZigInput.Update and COCheck2Hands.Update are the functions that relate to what we implement for the system. ZigInput.Update is implemented by Zigfu team. It is mainly used to process user movements captured by the Kinect™ camera. For COCheck2Hands.Update, the main purpose is to interpret user hands movement to system commands such as scaling and rotating virtual objects on a scene.
Table 5.3: CPU usage for each captured frame on Microsoft Windows 7 operating system.

<table>
<thead>
<tr>
<th>Snapshot number</th>
<th>Camera.Render</th>
<th>GPU time(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Scene 1</td>
<td>Scene2</td>
</tr>
<tr>
<td>1</td>
<td>31.69</td>
<td>33.69</td>
</tr>
<tr>
<td>2</td>
<td>25.92</td>
<td>25.29</td>
</tr>
<tr>
<td>3</td>
<td>32.04</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>18.82</td>
<td>33.74</td>
</tr>
<tr>
<td>5</td>
<td>25.61</td>
<td>27.99</td>
</tr>
<tr>
<td>6</td>
<td>19.58</td>
<td>24.35</td>
</tr>
<tr>
<td>7</td>
<td>29.27</td>
<td>20.03</td>
</tr>
<tr>
<td>8</td>
<td>20.14</td>
<td>19.01</td>
</tr>
<tr>
<td>9</td>
<td>30.6</td>
<td>20.3</td>
</tr>
<tr>
<td>10</td>
<td>17.97</td>
<td>15.3</td>
</tr>
</tbody>
</table>

Table 5.4: GPU usage for each captured frame on Microsoft Windows 7 operating system.

<table>
<thead>
<tr>
<th>Snapshot number</th>
<th>Camera.Render</th>
<th>GUI.Repaint</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Scene1</td>
<td>Scene2</td>
</tr>
<tr>
<td>1</td>
<td>97.8</td>
<td>95</td>
</tr>
<tr>
<td>2</td>
<td>98.2</td>
<td>93.6</td>
</tr>
<tr>
<td>3</td>
<td>97.7</td>
<td>95.1</td>
</tr>
<tr>
<td>4</td>
<td>97.9</td>
<td>95.6</td>
</tr>
<tr>
<td>5</td>
<td>98.3</td>
<td>94.3</td>
</tr>
<tr>
<td>6</td>
<td>98.4</td>
<td>95</td>
</tr>
<tr>
<td>7</td>
<td>98.3</td>
<td>95.9</td>
</tr>
<tr>
<td>8</td>
<td>98.5</td>
<td>94.9</td>
</tr>
<tr>
<td>9</td>
<td>98.3</td>
<td>95.1</td>
</tr>
<tr>
<td>10</td>
<td>97.9</td>
<td>95.4</td>
</tr>
</tbody>
</table>
(a) Scene 1: Living cell with 485,300 triangles and approximately 441,200 to 445,000 vertices.

(b) Scene 2: beta-sheet 56,300 triangles and approximately 55,800 vertices.

Figure 5.2: The most time-consuming functions and CPU usage for Microsoft Windows 7 operating system.
5.3.2 Performance on Mac OS X

The results from Table 5.5 shows CPU usage time for every captured frame (from both selected scene in milliseconds). The average CPU usage for scene1 is 27.79 milliseconds. Similarly, the average CPU usage for scene2 is 23.97 milliseconds. In addition, Figure 5.3 represents the most time consuming functions for all captured frame. It indicates that ZigInput.Update function consumes the most CPU usage followed by StackTraceUtility and COCheck2Hands.update. ZigInput.Update and COCheck2Hands.update are the functions that relate to what we have implemented as we mentioned before, whereas StackTraceUtility is the function, which is provided by Unity3D Profiler for tracking CPU usage.

Table 5.5: CPU usage for each captured frame on Mac OS X operating system.

<table>
<thead>
<tr>
<th>Snapshot number</th>
<th>CPU time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Scene 1</td>
</tr>
<tr>
<td>1</td>
<td>28.45</td>
</tr>
<tr>
<td>2</td>
<td>36.5</td>
</tr>
<tr>
<td>3</td>
<td>36.81</td>
</tr>
<tr>
<td>4</td>
<td>24.81</td>
</tr>
<tr>
<td>5</td>
<td>23.48</td>
</tr>
<tr>
<td>6</td>
<td>23.42</td>
</tr>
<tr>
<td>7</td>
<td>23.94</td>
</tr>
<tr>
<td>8</td>
<td>35.51</td>
</tr>
<tr>
<td>9</td>
<td>22.68</td>
</tr>
<tr>
<td>10</td>
<td>22.42</td>
</tr>
</tbody>
</table>
(a) Scene 1: living cell with 485,300 triangles and approximately 441,200 to 445,000 vertices.

(b) Scene 2: beta-sheet 56,300 triangles and approximately 55,800 vertices.

Figure 5.3: The most time-consuming functions and CPU usage for Mac OS X operating system.
5.4 Usability Results

In this experiment, ten students from The University of Akron participated. We gave each student five different tasks for each design, (between the hand pushing and the hand holding method) in order to measure how good the usability designs are. We expect to find a suitable method in term of usability for the system. The results from Figure 5.4 and Figure 5.5 show that the level of success for the hand pushing method is significantly lower than the hand holding method. There is no failure (which is indicated by red color on the stacked bars) for the hand holding method. In contrast, there are some failures for the hand pushing method. In addition, the times that the students spend on each task were averaged (Figure 5.6). We found that the average time for the hand holding method is lower than that of the hand pushing method in every given tasks. Besides the level of success and the average times; efficiency is measured by using the Common Industry Format for Usability Test Reports (ISO/IEC 25062:2006). According to the results from Figure 5.9, Task four has the highest efficiency for the hand holding method whereas the highest efficiency for the hand pushing method is on Task three. If we look into each given task, we can obviously see that the students perform the tasks by using the hand holding method with more efficiency (than the hand pushing method).

In addition, we also did another experiment about mode switching between UI interaction mode and virtual object manipulation mode. Every students went though five different tasks two times. For the first time the students used the head shaking
Figure 5.4: Stacked bar chart showing different levels of success based on task completion for the hand pushing method.

Figure 5.5: Stacked bar chart showing different levels of success based on task completion for the hand holding method.
method to switch between UI interaction mode and virtual object manipulation mode, whereas they used the hand dropping method for the second time. We expect that this experiment will help us find a suitable method for the mode switching. Figure 5.7 shows the level of success results for the head shaking method, whereas Figure 5.8 shows the level of success results for the hand dropping method. There are three different colors on the stacked bar. Green color on the stacked bar indicates that students complete the given tasks successfully without any difficulty or inefficiency. The yellow color indicates the students partial success to perform the given tasks. They may not succeed in performing the tasks at the first time. When given assistance for the program, they seem to perform better and succeed the given tasks. The red color indicates that students cannot complete the given tasks. Base on results, when students perform the tasks with the hand dropping method, they tend to succeed
more than using the head shaking method. Unfortunately, they cannot complete the
tasks successfully, they still partially succeed when performing the tasks with the
hand dropping method. Additionally, the efficiency metric (Figure 5.9) shows that
the hand dropping method is more efficient than the head shaking method. Thus, we
can conclude that the hand dropping method is better than the head shaking method
for switching between UI interaction and virtual object manipulation.
Figure 5.8: Stacked bar chart showing different levels of success based on task completion for hands dropping method.

Figure 5.9: Stacked bar chart showing efficiency for the hand dropping method and the head shaking method.
CHAPTER VI
SUMMARY AND FUTURE WORK

6.1 Summary

This Kinect™ based education system is designed and developed using Autodesk Maya, Blender3D and Unity3D. All 3D virtual models are created by using Autodesk Maya and Blender3D, where as the coding is done using C# and JavaScript in Unity3D. The system can be run on both Microsoft Windows and Mac OS operating systems. The Unity3D Profiler is used to get the performance data from both operating systems. The results show that ZipInput.Update and COCheck2Hand.update consume the most CPU usage. Thus, system performance can be improved by optimising these functions. In addition, this system is also designed to measure learning outcome for groups of students between the traditional teaching method and the Kinect™ based method. From the results, the group of students who were involved in the Kinect™ based teaching method score better than the group of students who were involved in the traditional teaching method. Hence, this system can be used to improve learning outcomes for students. It also can be adapted with other subjects such as Physics, Mathematics, etc. Two different UI interaction (the hand holding, and the hand pushing) are created for the system. Our study shows that the hand
holding method is better than the hand pushing method. Students spend less time to finish all given tasks using the hand holding method than that using the hand pushing method. The errors from the hand holding method is less than the hand pushing method as well. Additionally, two different modes of switching (the head shaking, and the hand dropping) between UI and 3D virtual object interactions are created for the system. Base on the results, the hand dropping method is better than the head shaking method. Students spend less time to finish all given tasks using the hand dropping method than that using the head shaking method. Lastly, we published a paper “Interactive Learning to Stimulate the Brain’s Visual Center and to Enhance Memory Retention” in IADIS Proceedings of the International Conference on E-Learning[27].

6.2 Future Work

The contents of the system are related to Biology only. In the future, other subjects, such as Physics could be implemented in the system. Since the system is designed to support any kind of 3D virtual objects, thus, developer just need to create new assets and animations and import them to the system. Moreover, system optimization could be improved over ZipInput.Update and COCheck2Hand.update functions. They are very expensive for CPU processing. We can also port the system to other platforms, such as iPad and smart phones, and enable the system to support other user interface devices, such as Leap Motion (a sensory device that supports
hand and finger movements). It would be interesting to compare the performance and effects of different platforms and devices on the learning outcomes.


using UnityEngine;
using System.Collections;

public class COCheck2HandForScale : MonoBehaviour {

    public COHandRayCast[] hands;

    public bool TutorialMode = false;
    public bool TutorialMode3 = false;

    public bool enableScale = false;
    public bool enableRotate = false;

    public COHandClickButton[] handsRotate;

    public float scaleFactor = 1.0f;
    public float rotationFactor = 2.0f;

    float scaleTempPoint = 0;
    float distanceHands = 0;
    public bool m_isZoom = false;
}
bool m_isRotate = false;

float rotation = 0;
float rotationY = 0;
float rotationZ = 0;

int rotationX1 = 0;
public int rotationY1 = 0;
public int rotationZ1 = 0;

public float yFactor = 5.0f;

public bool FreezeScale = false;
public bool FreezeRotate = false;

public Transform refPoint;

Vector3 perpendicularV;
float LeftHandZ;
float RightHandZ;

ZigEngageSingleUser user;

public int testX = 0;
//public int testY = 0;
//Vector3 perpendicularVChange;

float degree = 0;

GameObject txt;

GameObject zoomObject;

public bool ExperimentMode = false;
// Use this for initialization

void Start () {
    user = GameObject .FindGameObjectWithTag ("staticOpenNI") .GetComponent <ZigEngageSingleUser> ();
    ZigJointId id = ZigJointId. Head;

    txt = GameObject .FindGameObjectWithTag ("expTxt2");

    //Hpos = user .engagedTrackedUser .Skeleton[8].Position; //ZigJointId. Head]. Position;
}

public void activeRayToPlane () {
    hands[0] . isReadyToRay = true;
    hands[1] . isReadyToRay = true;
}

public void deactiveRayToPlane () {
hands[0].isReadyToRay = false;
hands[1].isReadyToRay = false;

public void reset() {
    rotation = 0;
    rotationY = 0;
    rotationZ = 0;
}

// Update is called once per frame

void exmode() {
    if (GetComponent<COCheckHeadRotation>().objectMode) {
        if (!m_isZoom) {
            GameObject objCheck = GameObject.FindWithTag("modelZoom");
            if (objCheck) {
                if (objCheck.GetComponent<COData>() == null) {
                    objCheck.AddComponent<COData>();
                }
            }
            if (objCheck != zoomObject) {
                rotation = 0;
                rotationY = 0;
                rotationZ = 0;
            }
        }
    }
    zoomObject = objCheck;
scaleTempPoint = zoomObject.transform.localScale.x; // obj.

GetComponent<COData>().setScaleData().x;

distanceHands = Vector3.Distance(hands[0].transform.position, hands[1].transform.position);

m_isZoom = true;

}

}

// if ()

if (!m_isRotate) {

GameObject obj = GameObject.FindWithTag("modelZoom");

if (obj) {

m_isRotate = true;

hands[0].GetComponent<COHandClickButton>().StartCheckRotate();

hands[1].GetComponent<COHandClickButton>().StartCheckRotate();

if (obj.GetComponent<COData>() == null) {

obj.AddComponent<COData>();

}

}

Vector3 temp = hands[0].transform.position - hands[1].transform.position;


perpendicularV = Vector3.Cross(temp, Vector3.forward).normalized;

//perpendicularVChange = new Vector3(perpendicularV.x, perpendicularV.y, perpendicularV.z);
//degree = Vector3.Angle(hands[0].transform.position, hands[1].transform.position);

if (obj != null) {
    if (!obj.GetComponent<COData>().rotate) {
        m_isRotate = false;
    }
    else {
    }
}

if (TurtorialMode3) {
    m_isRotate = true;
    m_isZooom = true;
}

//obj.transform.rotation = target;
}

} // end check on pad
else {
    if (TurtorialMode) {
/print ("here1");
if (txt) {
    /print ("here2");
    txt.GetComponent<TextMesh>().text = "Move your hand to the center of the object."
}

if (m_isZoom) {
    /print("false−2");
    if (zoomObject == null) {
        m_isZoom = false;
        if (TurtorialMode3) {
            }
        } else {
            return;
            }
        } //print("false−1");
        //test Rotate
        
        if (user.EngagedUsers.Count > 0 && user.engagedTrackedUser != null)
        { //&& user.engagedTrackedUser.SkeletonTracked) { 
        }
LeftHandZ = user.engagedTrackedUser.Skeleton[(int)ZigJointId].LeftHand].Position.z;
RightHandZ = user.engagedTrackedUser.Skeleton[(int)ZigJointId].RightHand].Position.z;

if (LeftHandZ - RightHandZ > 300) {
    rotationY++; rotationY1 = 1;
    // print ("ychange1");
}
else if (LeftHandZ - RightHandZ < -300) {
    rotationY--; rotationY1 = -1;
    // print ("ychange1.2");
}
else if (LeftHandZ - RightHandZ >= -300 && LeftHandZ - RightHandZ <= 300) {
    rotationY1 = 0;
    // print ("ychange1.3");
}

//

///// set global checkmark;
GameObject.FindGameobjectWithTag("staticOpenNI").GetComponent<
COGeneralData>().isFreezeButton = true;
if (Mathf.Abs(hand[0].transform.position.y - hand[1].transform.position.y) < rotationFactor) {
    float tempDis = Vector3.Distance(hand[0].transform.position, hand[1].transform.position);
    float scalePoint = (((tempDis - distanceHands) / distanceHands) * scaleFactor + 1.0f) * scaleTempPoint;
    if (!FreezeScale) {
        if (!TutorialMode && !TutorialMode3) {
            zoomObject.transform.localScale = new Vector3(scalePoint, scalePoint, scalePoint);
        }
        else {
            // check
            if (enableScale) {
                // show text
                if (txt) {
                    txt.GetComponent<TextMesh>().text = "Drop your hands down to stop performing the action."
                }
            }
            //
            zoomObject.transform.localScale = new Vector3(scalePoint, scalePoint, scalePoint);
        }
    }
}
// print("false0");

if (TurtorialMode3) {
    // print(hands[0].transform.localPosition.y);
    if (hands[0].transform.localPosition.y < 1.0f && hands[1].transform.localPosition.y < 1.0f) {
        // print("false2");
        hands[0].hitOnPad = false;
        hands[1].hitOnPad = false;
    }
}

if (hands[0].transform.position.y < 1.0f && hands[1].transform.position.y < 1.0f) {
    // print("false");
    // print("false1");

    m_isZoom = false;

    if (m_isRotate) {
        m_isRotate = false;
        handsRotate[0].m_isRotate = false;
        handsRotate[1].m_isRotate = false;
        hands[0].GetComponent<COHandClickButton>().StopRotate();
        hands[1].GetComponent<COHandClickButton>().StopRotate();
        hands[0].hitOnPad = false;
        hands[1].hitOnPad = false;
    }
}
231   //hands[0].isReadyToRay = true;
232   //hands[1].isReadyToRay = true;
233   }
234
235
236
237   }
238
239   else {
240     GameObject.FindGameObjectWithTag("staticOpenNI").GetComponent<
241       COGeneralData>().isFreezeButton = false;
242   }
243
244   if (m.isRotate) {
245     GameObject obj = GameObject.FindWithTag("modelZoom");
246     if (handsRotate[0].m_isRotate) {
247       rotationY++;
248       rotationY1 = 1;
249       print("ychange2");
250     }
251     else {
252       //rotationY1 = 0;
253       //print("ychange2.1");
254     }
255   }
if (handsRotate[1].m_isRotate) {
    // rotation++;
    rotationY--; 
    rotationY1 = -1;
}
else {
    // rotationY1 = 0;
    // print ("ychange2.2");
}

// if (hands[0].transform.position.y > 12.5f || hands[1].transform.
  position.y > 12.5f)
// rotationZ++; 

Vector3 temp2 = hands[0].transform.position - hands[1].transform.
  position;
Vector3 perpendicular = Vector3.Cross(temp2, Vector3.forward);
/*if (perpendicular != perpendicularV) {
  print(perpendicularV +"nnnnnn" + perpendicular);
  print(Vector3.Angle(perpendicularV, perpendicular));
}*/

// LeftRightTest.AngleDir();
float degreeTemp = Vector3.Angle(perpendicularV, perpendicular);
float flag = 1;

// print ("adjust cannon 11");
if (hands[0].transform.position.y > hands[1].transform.position.y)
{
  // print ("adjust cannon 22");
  if (hands[0].transform.position.y - hands[1].transform.position.
y > yFactor)
  {
    rotationZ++;
    rotationZ1 = 1;
    // print ("adjust cannon 000");
    if (TurtorialMode3 && hands[0].hitOnPad && hands[1].hitOnPad)
    {
      // print (hands[0].hitOnPad && hands[1].hitOnPad);
      GameObject[] guns;
      guns = GameObject.FindGameObjectsWithTag("gun");
      if (GameObject.Find("cannonBT").GetComponent<COCannonRotateBT>());
        .enableRotate)
        {
          foreach (GameObject g in guns) {
            g.transform.RotateAround(new Vector3(0, 0, 1), 0.02f);
            if (GameObject.Find("angel")) {
              GameObject.Find("angel").GetComponent<TextMesh>().text =
              string.Format("{0:0.0}", ((90.0f - g.transform.
              localEulerAngles.x)) + " ");
          }
        }
    }
}
else {
    rotationZ1 = 0;
}

flag = 1;
}
else {
    if (GetComponent<COCheckHeadRotation>().uiMode) {
        rotationZ --;
        rotationZ1 = -1;
        // print ("adjust cannon 00");
        if (TurtorialMode3 && hands[0].hitOnPad && hands[1].hitOnPad) {
            // if (hands[0].hitOnPad && hands[1].hitOnPad) {
            // print ("adjust cannon");
            GameObject[] guns;
            // print (hands[0].hitOnPad && hands[1].hitOnPad);
            guns = GameObject.FindGameObjectsWithTag("gun");
            if (GameObject.Find("cannonBT").GetComponent<
                 COCannonRotateBT>().enableRotate) {
                foreach(GameObject g in guns) {
                    g.transform.RotateAround(new Vector3(0, 0, 1), -0.02f);
                    if (GameObject.Find("angel")) {
                        GameObject.Find("angel").GetComponent<TextMesh>().text
                            = string.Format("{0:0.0}", (90.0f - g.transform.
```
localEulerAngles.x)); + " ");

    }
  }
}

else {
  rotationZ1 = 0;
}

flag = -1;

if (!FreezeRotate) {
  if (refPoint) {
    print ("yeah");
    Vector3 veref = new Vector3();
    veref.x = rotation;
    veref.y = rotationY;
    veref.z = rotationZ;

    Vector3 originalR = obj.GetComponent<COData>().getRotationData();
    Vector3 nowR = obj.transform.rotation.eulerAngles;
obj.transform.rotation = Quaternion.FromToRotation(nowR, originalR);

if(rotationY1==1) {
    obj.transform.RotateAround(refPoint.transform.position, Vector3.up, Time.deltaTime * 20.0f);
}
else if(rotationY1==-1) {
    obj.transform.RotateAround(refPoint.transform.position,- Vector3.up, Time.deltaTime * 20.0f);
}
if(rotationZ1==1) {
    obj.transform.RotateAround(refPoint.transform.position, Vector3.forward, Time.deltaTime * 20.0f);
    // print("Y++");
}
else if(rotationZ1==-1) {
    obj.transform.RotateAround(refPoint.transform.position,- Vector3.forward, Time.deltaTime * 20.0f);
    // print("Y--");
}
obj.transform.rotation = Quaternion.FromToRotation(obj.
    transform.rotation.eulerAngles,nowR);
}

else {

if (!TurtorialMode && !TurtorialMode3) {
    // rotate object (normal style)
    if (obj.GetComponent<CORotateAroundRef>() == null) {
        Quaternion target = Quaternion.Euler(rotation, rotationY, rotationZ);
        obj.transform.rotation = Quaternion.Slerp(obj.transform.rotation, target, Time.deltaTime * 4.0f);
    }
    else {
        print("y1:"+rotationY1);
        CORotateAroundRef rotateFunc = obj.GetComponent<
            CORotateAroundRef>();
        rotateFunc.RotationV.x = rotationX1;
        rotateFunc.RotationV.y = rotationY1; // test Y
        rotateFunc.RotationV.z = rotationZ1;
        print("rotate by using ref point");
    }
}
else {
    if (enableRotate) {
        if (txt) {
            txt.GetComponent<TextMesh>().text = "Drop your hands down to stop performing the action.";
        }
    }
}
Quaternion target = Quaternion.Euler(rotation, rotationY, rotationZ); //

obj.transform.rotation = Quaternion.Slerp(obj.transform.
rotation, target, Time.deltaTime * 4.0f);

void Update () {
    Profiler.BeginSample("a");
    for (int i=0; i<10; i++) {
        int a = 1+3;
    }
    Profiler.EndSample();
    if (ExperimentMode) {
        exmode();
    }
    else {
        if (hands[0].hitOnPad && hands[1].hitOnPad) {

if (!m_isZoom) {
    GameObject objCheck = GameObject.FindWithTag("modelZoom");
    if (objCheck) {
        if (objCheck.GetComponent<COData>() == null) {
            objCheck.AddComponent<COData>();
        }
        if (objCheck != zoomObject) {
            rotation = 0;
            rotationY = 0;
            rotationZ = 0;
        }
    }
    zoomObject = objCheck;
    scaleTempPoint = zoomObject.transform.localScale.x;// obj. GetComponent<COData>().ScaleData().x;
    distanceHands = Vector3.Distance(hands[0].transform.position, hands[1].transform.position);
    m_isZoom = true;
}

if (!m_isRotate) {
    //if ()
    if (!m_isRotate) {

101
 GameObject obj = GameObject.FindWithTag("modelZoom");

if (obj) {
    m_isRotate = true;
    hands[0].GetComponent<COHandClickButton>().StartCheckRotate();
    hands[1].GetComponent<COHandClickButton>().StartCheckRotate();
    if (obj.GetComponent<COData>() == null) {
        obj.AddComponent<COData>();
    }

    Vector3 temp = hands[0].transform.position - hands[1].transform.position;

    perpendicularV = Vector3.Cross(temp, Vector3.forward).normalized;
    // perpendicularVChange = new Vector3(perpendicularV.x, perpendicularV.y, perpendicularV.z);
    // degree = Vector3.Angle(hands[0].transform.position, hands[1].transform.position);

    if (obj != null) {
        if (!obj.GetComponent<COData>().rotate) {
            m_isRotate = false;
        }
    }
    else {

if (TutorialMode3) {
    m_isRotate = true;
    m_isZoom = true;
}

// object.transform.rotation = target;

} // end check on pad

else {
    if (TutorialMode) {
        // print("here1");
        if (txt) {
            // print("here2");
            txt.GetComponent<TextMesh>().text = "Move your hand to the center of the object."
        }
    }
}

if (m_isZoom) {
    // print("false\-2");
    if (zoomObject == null) {
        m_isZoom = false;
    }
if (TutorialMode3) {

}

else {
    return;
}
}

// print("false-1");
// test Rotate

if (user.EngagedUsers.Count > 0 && user.engagedTrackedUser != null) { //&& user.engagedTrackedUser != null ) {
    LeftHandZ = user.engagedTrackedUser.Skeleton[int]ZigJointId.
    LeftHand].Position.z;

    RightHandZ = user.engagedTrackedUser.Skeleton[int]ZigJointId.
    RightHand].Position.z;

    if (LeftHandZ - RightHandZ > 300.0f) {
        rotationY++;
        rotationY1 = 1;
        // print("ychange1");
    }
}
else if (LeftHandZ - RightHandZ < -300.0f){
    rotationY--;
    rotationY1 = -1;
    // print("ychange1.2");

else if (LeftHandZ - RightHandZ >= -300.0f && LeftHandZ -
    RightHandZ <= 300) {
    rotationY1 = 0;
    // print ("ychange1.3");
}

//

//////////set global checkmark;
GameObject.FindGameObjectWithTag("staticOpenNI").GetComponent<
    COGeneralData>().isFreezeButton = true;

if (Mathf.Abs(hands[0].transform.position.y - hands[1].transform.
    position.y) < rotationFactor) {
    float tempDis = Vector3.Distance(hands[0].transform.position,
        hands[1].transform.position);
    float scalePoint = (((tempDis - distanceHands)/distanceHands)*
        scaleFactor + 1.0f) * scaleTempPoint;
    if (!FreezeScale) {
        if (!TurtorialMode && !TurtorialMode3) {
            zoomObject.transform.localScale = new Vector3(scalePoint,
                scalePoint, scalePoint);
        }
        else {
            // check
if (enableScale) {
    // show text
    if (txt) {
        txt.GetComponent<TextMesh>().text = "Drop your hands
down to stop performing the action."
    }
    //
    zoomObject.transform.localScale = new Vector3(scalePoint,
        scalePoint, scalePoint);
}

// print("false0");
if (TutorialMode3) {
    // print(hands[0].transform.localPosition.y);
    if (hands[0].transform.localPosition.y < 1.0f && hands[1].
        transform.localPosition.y < 1.0f) {
        // print("false2");
        hands[0].hitOnPad = false;
        hands[1].hitOnPad = false;
    }
}

if (hands[0].transform.position.y < 1.0f && hands[1].transform.
    position.y < 1.0f) {
m_isZoom = false;

if (m_isRotate) {
    m_isRotate = false;
    handsRotate[0].m_isRotate = false;
    handsRotate[1].m_isRotate = false;
    hands[0].GetComponent<COHandClickButton>().StopRotate();
    hands[1].GetComponent<COHandClickButton>().StopRotate();
    hands[0].hitOnPad = false;
    hands[1].hitOnPad = false;
    // hands[0].isReadyToRay = true;
    // hands[1].isReadyToRay = true;
}

else {
    GameObject.FindGameObjectWithTag("staticOpenNI").GetComponent<
        COGeneralData>().isFreezeButton = false;
}
if (m_isRotate) {
    GameObject obj = GameObject.FindWithTag("modelZoom");
    if (handsRotate[0].m_isRotate) {
        rotationY++;  //rotationY1 = 0;
        rotationY1 = 1;
        print("ychange2");
    } else {
        //rotationY1 = 0;
        //print("ychange2.1");
    }

    if (handsRotate[1].m_isRotate) {
        rotationY--;  //rotation++;  //rotationY1 = 0;
        rotationY1 = -1;
    } else {
        //rotationY1 = 0;
        //print("ychange2.2");
    }
}
// if (hands[0].transform.position.y > 12.5f || hands[1].transform.
position.y > 12.5f)
    //rotationZ++; 
Vector3 temp2 = hands[0].transform.position - hands[1].transform.
posiotion;
Vector3 perpendicular = Vector3.Cross(temp2, Vector3.forward);
/* if (perpendicular != perpendicularV) {
print(perpendicularV +"nnnnnn"+ perpendicular);
print(Vector3.Angle(perpendicularV, perpendicular));
}*/

//LeftRightTest.AngleDir();
float degreeTemp = Vector3.Angle(perpendicularV, perpendicular);
float flag = 1;
//print ("adjust cannon 11");
if (hands[0].transform.position.y > hands[1].transform.position.y)
{
    //print ("adjust cannon 22");
    if (hands[0].transform.position.y - hands[1].transform.position.
y > yFactor) {
        rotationZ++; 
        rotationZ1 = 1;
    //print ("adjust cannon 000");
    if (TurtorialMode3 && hands[0].hitOnPad && hands[1].hitOnPad)
    {

//print (hands[0].hitOnPad && hands[1].hitOnPad);

GameObject[] guns;

guns = GameObject.FindGameObjectsWithTag("gun");

if (GameObject.Find("cannonBT").GetComponent<COCannonRotateBT>().enableRotate) {
    foreach (GameObject g in guns) {
        g.transform.RotateAround(new Vector3(0, 0, 1), 0.02f);

        if (GameObject.Find("angel")) {
            GameObject.Find("angel").GetComponent<TextMesh>().text =
            string.Format("{0:0.0}", ((90.0f - g.transform.
            localEulerAngles.x)) + " ");
        }
    }
}

else {
    rotationZ1 = 0;
}

flag = 1;
}
else {
    if (hands[0].transform.position.y - hands[1].transform.position.
        y < -yFactor) {
        rotationZ--;
rotationZ1 = -1;

// print ("adjust cannon 00");

if (TurtorialMode3 && hands[0].hitOnPad && hands[1].hitOnPad)
    {
        if (hands[0].hitOnPad && hands[1].hitOnPad) {
            // print ("adjust cannon");

        GameObject[] guns;

        // print (hands[0].hitOnPad && hands[1].hitOnPad);

        guns = GameObject.FindGameObjectsWithTag("gun");

        if (GameObject.Find("cannonBT").GetComponent<COCannonRotateBT>().enableRotate)
            foreach (GameObject g in guns) {

                g.transform.RotateAround(new Vector3(0, 0, 1), -0.02f);

                if (GameObject.Find("angel")) {

                    GameObject.Find("angel").GetComponent<TextMesh>().text
                        = string.Format("{0:0.0}", ((90.0f - g.transform.localEulerAngles.x)) + " ");

                }

            }

        }

    }

else {

    rotationZ1 = 0;

}

flag = -1;

}
if (!FreezeRotate) {
    if (refPoint) {
        print ("yeah");
        Vector3 vecref = new Vector3();
        vecref.x = rotation;
        vecref.y = rotationY;
        vecref.z = rotationZ;

        Vector3 originalR = obj.GetComponent<COData>().getRotationData();
        Vector3 nowR = obj.transform.rotation.eulerAngles;

        obj.transform.rotation = Quaternion.FromToRotation(nowR, originalR);

        if (rotationY1==1) {
            obj.transform.RotateAround(refPoint.transform.position, Vector3.up, Time.deltaTime * 20.0f);
        }
        else if (rotationY1==-1) {
            obj.transform.RotateAround(refPoint.transform.position, Vector3.up, Time.deltaTime * 20.0f);
        }
    }
}
if (rotationZ1 == 1) {
    obj.transform.RotateAround(refPoint.transform.position,
        Vector3.forward, Time.deltaTime * 20.0f);
    // print("Y++");
}
else if (rotationZ1 == -1) {
    obj.transform.RotateAround(refPoint.transform.position,
        -Vector3.forward, Time.deltaTime * 20.0f);
    // print("Y--");
}
obj.transform.rotation = Quaternion.FromToRotation(obj.
    transform.rotation.eulerAngles, nowR);
} else {
    if (!TurtorialMode && !TurtorialMode3) {
        // rotate object (normal style)
        if (obj.GetComponent<CORotateAroundRef>() == null) {
            Quaternion target = Quaternion.Euler(rotation, rotationY,
                rotationZ);
            obj.transform.rotation = Quaternion.Slerp(obj.transform.
                rotation, target, Time.deltaTime * 4.0f);
        }
    } else {

print ("y1:"+rotationY1);
CORotateAroundRef rotateFunc = obj.GetComponent<CORotateAroundRef>();
rotateFunc.RotationV.x = rotationX1;
rotateFunc.RotationV.y = rotationY1;  // test Y
rotateFunc.RotationV.z = rotationZ1;
print ("rotate by using ref point");
}
}
else {
    if (enableRotate) {
        if (txt) {
            txt.GetComponent<TextMesh>().text = "Drop your hands down to stop performing the action."
        }
    }
    Quaternion target = Quaternion.Euler(rotation, rotationY, rotationZ);  //
    obj.transform.rotation = Quaternion.Slerp(obj.transform.rotation, target, Time.deltaTime * 4.0f);
}
//rotationX1, y1 z1

/*
Vector3 nowR = obj.transform.rotation.eulerAngles;

vector3 originalR = obj.GetComponent<COData>().getRotationData();
obj.transform.rotation = Quaternion.Euler(originalR); // rotate back to original

///rotate by hand
float ry = 0;
if (rotationY1 == 1) {
  ry = 10.0f;
}
else if (rotationY1 == -1) {
  ry = -10.0f;
}
else {
}

float rz = 0;
if (rotationZ1 == 1) {

\[
rz = 10.0f;
\]

}  

}  

else if (rotationZ1 == -1) {  

rz = -10.0f;  

}  

else {  

}  

Quaternion target = Quaternion.Euler(rotation, ry, rz)  

://  

Quaternion rr = Quaternion.Slerp(Quaternion(originR),  

target, Time.deltaTime * 4.0f);  

//rotate back from old value  
//Vector3 afterR = obj.transform.rotation.eulerAngles;  

obj.transform.rotation = Quaternion.Slerp(rr, target, Time  

deltaTime * 4.0f);  

*/  

//rotate to z  
//increasae  

//rotate back  

}  

}  

}
```csharp
using UnityEngine;

using System.Collections;

public class COCheckHeadRotation : MonoBehaviour {

    public ZigMapJointToSession mapJoint;

    ZigJointId jointId = ZigJointId.Head;

    public bool objectMode = false;

    public bool uiMode = true;

    float idleTime = 2.0f;

    float time = 0;

    public float cancleActionAngle1 = 330.0f;

    public float cancleActionAngle2 = 30.0f;

    // Use this for initialization

    void Start () {
    }
```
/*ZigMapJointToSession[] mapJoints = transform.parent.GetComponents
<ZigMapJointToSession>();*/

foreach (ZigMapJointToSession j in mapJoints)
{
    if (j.joint == jointId)
    {
        mapJoint = j;
    }
}

public void Session.headJoint(Vector3 rotation)
{
    if ((rotation.z > 45 && rotation.z < 80) || (rotation.z < 345 &&
        rotation.z > 315))
    {
        if (time >= idleTime)
        {
            time = 0;
            uiMode = !uiMode;
            objectMode = !objectMode;
            print(rotation);
            if (objectMode)
            {
                GameObject.Find("textMode").GetComponent<TextMesh>().text = "Object Mode";
            }
            else
            {
                GameObject.Find("textMode").GetComponent<TextMesh>().text = "GUI Mode";
            }
        }
    }
}
// Update is called once per frame

void Update () {
    time += Time.deltaTime;

    if (GetComponent<COCheck2HandForScale>().m_isZoom) {

        //print(mapJoint.userTemp.Skeleton[(int)jointId].Rotation.
        eulerAngles.z);
        /*if (mapJoint.userTemp != null) {
            print (mapJoint.userTemp.Skeleton[(int)jointId].Rotation.
            eulerAngles.z);
            if (mapJoint.userTemp.Skeleton[(int)jointId].Rotation.eulerAngles.
            z < cancelActionAngle) {
                print("zzz");
                GetComponent<COCheck2HandForScale>().m_isZoom = false;
                for (int i=0 ; i<transform.childCount; i++) {
                    transform.GetChild(i).GetComponent<COHandOnScalePad>().
                    isStopScaleOnArea = true;
                }
            }
            */
        }
    }
}

COButtonRTMN.cs ...
using UnityEngine;
using System.Collections;

public class COButtonNew_RTMM : MonoBehaviour {

    bool isHit = false;

    public bool isReset;
    public bool isShowText;
    public bool isMenu;
    public Transform[] model;
    bool m_isShowText = false;

    float timeCount;
    bool isDebug;
    COTouchMiniCircle touch;
    // Use this for initialization
    void Start () {
        isDebug = GameObject.FindGameObjectWithTag("staticOpenNI").GetComponent<COGeneralData>().debug;
    }

    void OnCollisionEnter(Collision collision) {
        if (collision.gameObject.tag == "hand") {
            
        
    }
if (isDebug) {
    if (collision.gameObject.tag == "hand") {
        touch = collision.gameObject.transform.GetChild(0).GetComponent<COTouchMiniCircle>();
        touch.startCheck();
    }
}

void OnCollisionStay(Collision collision)
{
    if (GameObject.FindGameObjectWithTag("staticOpenNI").GetComponent<COGeneralData>().isFreezeButton) {
        return;
    }
}

if (collision.gameObject.tag == "hand")
{
    if (isDebug) {
        if (touch != null) {
            if (touch.isClickDone) {
                isHit = true;
                touch.endAction();
                touch = null;
            }
        }
    }
}
GameObject obj = GameObject.FindGameObjectsWithTag("MainCamera");

obj.audio.Play();

/timeCount += Time.deltaTime;
if (timeCount >= 1.0f) {
  timeCount = 0;
  isHit = true;
}
/*
else {
  COHandClickButton handClick = collision.gameObject.GetComponent&lt;COHandClickButton&gt;();
  if (!handClick.isAlreadyStart()) {
    handClick.StartCheckClickButton();
  }
  else {
    if (handClick.isClick()) {
      isHit = true;
      handClick.StopCheckClickButton();
      GameObject obj = GameObject.FindGameObjectsWithTag("MainCamera");
      obj.audio.Play();
    }
73         }
74         }
75         }
76         }
77         }
78
79     void OnCollisionExit(Collision collisionInfo)
80     {
81         if (isDebug) {
82             if (touch != null) {
83                 touch.endAction();
84                 touch = null;
85             }
86         }
87         }
88     else {
89         collisionInfo.gameObject.GetComponent<COHandClickButton>().StopCheckClickButton();
90     }
91     }
92
93     // Update is called once per frame
94
95     public void checkShowText() {
96         GameObject[] txts=GameObject.FindGameObjectsWithTag("lineGroup");
97         foreach(GameObject a in txts) {
98             if (m_isShowText)
a.renderer.enabled = true;

else
    a.renderer.enabled = false;
}
GameObject model = GameObject.FindWithTag("modelZoom");
model.transform.GetComponent<COData>().setBack();
}

public bool isShowTextFunc() {
    return m_isShowText;
}

void Update() {
    if(Input.GetKeyDown(KeyCode.T)) {
        GameObject model = GameObject.FindWithTag("modelZoom");
        if (model) {
            model.transform.GetComponent<COData>().setBack();
        }
        GameObject[] txts = GameObject.FindGameObjectsWithTag("lineGroup");
        foreach(GameObject a in txts) {
            a.renderer.enabled = true;
        }
        if (isHit) {
            isHit = false;
            if (isReset) {
                GameObject model = GameObject.FindWithTag("modelZoom");
            }
        }
    }
}
if (model) {
    if (model.transform.GetComponent<COData>() != null) {
        model.transform.GetComponent<COData>().setBack();
        GameObject.FindGameObjectWithTag("hand").transform.parent.GetComponent<COCheck2HandForScale>().reset();
    }
}

if (isShowText) {
    m_isShowText = !m_isShowText;
    GameObject[] txts = GameObject.FindGameObjectsWithTag("lineGroup");
    foreach (GameObject a in txts) {
        if (m_isShowText)
            a.renderer.enabled = true;
        else
            a.renderer.enabled = false;
    }
}

GameObject model = GameObject.FindWithTag("modelZoom");
if (model) {
    model.transform.GetComponent<COData>().setBack();
}

if (isMenu) {
    GameObject pdaObject = GameObject.Find("PlayerData");
}
if (pdaObject) {
    PlayerData pda = pdaObject.GetComponent<PlayerData>();
    if (pda.currentModule != 0) {
        pda.endModuleSession();
    }
}

Application.LoadLevel("main0Scene");

/*
GameObject[] models=GameObject.FindGameObjectsWithTag("model");
foreach(GameObject a in models)
{
    Destroy(a);
}

models=GameObject.FindGameObjectsWithTag("modelZoom");
foreach(GameObject a in models)
{
    Destroy(a);
}

models=GameObject.FindGameObjectsWithTag("button");
foreach(GameObject a in models)
{
    Destroy(a);
}
models = GameObject.FindGameObjectsWithTag("modelZoomLevel");

foreach (GameObject a in models)
{
    Destroy(a);
}

GameObject zoomBT = GameObject.FindWithTag("zoomButton");
if (zoomBT)
    Destroy(zoomBT);

Instantiate(model[GameObject.FindGameObjectWithTag("staticOpenNI")].GetComponent<COGeneralData>().iconMainTopicIndex);

GameObject[] sideMenu = GameObject.FindGameObjectsWithTag("sideMenu");

GameObject obj = GameObject.FindGameObjectWithTag("back");
if (obj != null) {
    obj.renderer.enabled = true;
obj.GetComponent<COMoveButton>().enabled = true;
}

obj = GameObject.FindGameObjectWithTag("forward");
if (obj != null) {
    obj.renderer.enabled = true;
    obj.GetComponent<COMoveButton>().enabled = true;
}

foreach(GameObject gg in sideMenu) {
    if (gg != null) {
        gg.GetComponent<COSideMenu>().deactive();
    }
}

obj = GameObject.FindGameObjectWithTag("scalePad");
if (obj != null) {
    obj.GetComponent<COScalePad1>().deactive();
}

obj = GameObject.FindGameObjectWithTag("ZoomLevelParentBT");
obj.transform.GetChild(0).GetComponent<MeshRenderer>().enabled = false;
obj.transform.GetChild(1).GetComponent<MeshRenderer>().enabled = false;

obj = GameObject.FindGameObjectWithTag("staticOpenNI");
obj.GetComponent<ZigDepthViewer>().enabled = true;

*/

} } }

}

} }

COTouchMiniCircle.cs ...

using UnityEngine; using System.Collections;

public class COTouchMiniCircle : MonoBehaviour {

public float timeMax = 1.5f;

public bool isClickDone = false;

float timeCount = 0;

int ballCount = 0;

int ballMax = 10;

bool isStart = false;

// Use this for initialization

void Start () {

}

public void startCheck() {

isStart = true;

}
public void endAction() {

GameObject[] hands = GameObject.FindGameObjectsWithTag("hand");

foreach(GameObject t in hands) {
    t.transform.GetChild(0).GetComponent<COTouchMiniCircle>().reset();
}
}

public void reset() {

for (int i = 0; i < transform.childCount; i++) {
    transform.GetChild(i).renderer.enabled = false;
}

ballCount = 0;
timeCount = 0;
isClickDone = false;
isStart = false;
}

// Update is called once per frame
void Update() {
    if (isStart) {
        timeCount += Time.deltaTime;

        if (timeCount >= timeMax / ballMax) {
            if (ballCount >= ballMax) {
                endAction();
            }
        }
    }
}
public class COMoveToPoint : MonoBehaviour {
    Vector3 currentPos;
    // Use this for initialization
    void Start () {
        currentPos = transform.position;
    }
}
public void MoveTo(Vector3 pos) {
    currentPos = pos;
}

// Update is called once per frame
void Update () {
    if(transform.position.x != currentPos.x || transform.position.y != currentPos.y) {
        Vector3 tPos = Vector3.Lerp(transform.localPosition, currentPos, Time.deltaTime * 4.0f);
        transform.localPosition = tPos;
        if (Mathf.Abs(transform.position.x - currentPos.x) < 1.0 && Mathf.Abs(transform.position.y - currentPos.y) < 1.0f) {
            transform.position = currentPos;
        }
    }
}

COTouchShowBox.cs ...

using UnityEngine;
using System.Collections;

public class COTouchShowBox : MonoBehaviour {

}
bool isHit = false;

bool isDebug = false;

float timeCount = 0;

public Transform txtBox;

// Use this for initialization

void Start () {
    if (txtBox.renderer) {
        txtBox.renderer.enabled = false;
    }

    if (txtBox.childCount > 0) {
        if (txtBox.GetChild(0).transform.renderer) {
            txtBox.GetChild(0).transform.renderer.enabled = false;
        }
    }

    isDebug = GameObject.FindGameObjectWithTag("staticOpenNI").GetComponent<COGeneralData>().debug;
}

void OnCollisionEnter(Collision collision)
30 {  
31     print(“aa”);  
32     if(collision.gameObject.tag==”hand”)  
33         gameObject.renderer.material.color=Color.white;  
34 }  
35 void OnCollisionStay(Collision collision)  
36 {  
37     if (renderer.enabled == false) {  
38         return;  
39     }  
40     if(collision.gameObject.tag==”hand”) {  
41         gameObject.renderer.material.color=Color.red;  
42         if (txtBox.renderer) {  
43             txtBox.renderer.enabled = true;  
44             for (int i=0;i<txtBox.GetChildCount();i++) {  
45                 txtBox.GetChild(i).transform.renderer.enabled = true;  
46             }  
47         }  
48     }  
49 }  
50 else {  
51     txtBox.GetComponent<GUITexture>().enabled = true;  
52 }  
53  
54 }  
55}
void OnCollisionExit(Collision collision) {
    if (renderer.enabled == false) {
        return;
    }

    if (txtBox.renderer) {
        txtBox.renderer.enabled = false;
        for (int i = 0; i < txtBox.GetChildCount(); i++) {
            txtBox.GetChild(i).transform.renderer.enabled = false;
        }
        gameObject.renderer.material.color = Color.white;
    } else {
        txtBox.GetComponent<GUITexture>().enabled = false;
    }
}

// Update is called once per frame
void Update() {
    if (isHit) {
        isHit = false;
        if (txtBox) {
            txtBox.renderer.enabled = !txtBox.renderer.enabled;
        }
    }
using UnityEngine;
using System.Collections;

public class COImmuneSystem : MonoBehaviour {
    public Transform Macrophage;
    public Transform[] TCell;
    public Transform BCell;
    public Transform PlasmaCell;
    public Transform Virus;
    public Transform Microphage;

    public Transform reset;

    bool isVirus = false;
    bool isProtein = false;
    bool isTCell = false;
    bool isTCellClone = false;
    bool isTCellMoveOut = false;
    bool isPlasmaCell = false;
    bool isLinkToBCell = false;
    bool isBCell = false;
}
bool isBCell03 = false;
bool isReset = false;

public int state = 0;
// 1 = virus move into cell
// 2 = helperTCell move to cell
// 3 = helperTCell move to BCell

// Use this for initialization
void Start () {
}

public void trigger () {
    state++;
}

// Update is called once per frame
void Update () {
    if (state==1) {
        if (!isVirus) {
            isVirus = true;
            Microphage.GetComponent<Animation>().Play();
        }
        for (int i=0; i< Virus.GetChildCount(); i++) {
            Virus.GetChild(i).GetComponent<Animation>().Play();
        }/*
    }
}
else if (state == 2) {
    if (!isProtein) {
        isProtein = true;
        GameObject[] immuePros = GameObject.FindGameObjectsWithTag("immueProtein");
        GameObject immueGap = GameObject.FindGameObjectWithTag("immueGap");

        foreach (GameObject immuePro in immuePros) {
            for (int i = 0; i < immuePro.transform.GetChildCount(); i++) {
                immuePro.transform.GetChild(i).GetComponent<Animation>().Play();
            }
        }
    } //immueGap.transform.GetChild(i).GetComponent<Animation>().
        Play();
}
else if (state == 3) {
    if (!isTCell) {
        isTCell = true;
        GameObject[] plugRen = GameObject.FindGameObjectsWithTag("immueplugRen");
    }
for (int i = 0; i < plugRen.Length; i++) {
    plugRen[i].renderer.enabled = true;
}

GameObject plug = GameObject.FindGameObjectWithTag("immuePlug");
for (int i = 0; i < plug.transform.GetChildCount(); i++) {
    plug.transform.GetChild(i).GetComponent<Animation>().Play();
    int j = 0;
    foreach (AnimationState animState in TCell[0].animation) {
        if (j == 0) {
            TCell[0].animation.Play(animState.name); // (animState);
            break;
        }
        j++;
    }
    // TCell[i].animation.Play();
}

/*
 else if (state == 4) {
    if (!isTCellMoveOut) {
        isTCellMoveOut = true;
        for (int i = 0; i < TCell.Length; i++) {
            int j = 0;
...
foreach (AnimationState animState in TCell[i].animation) {
    // print (animState.name);
    if (j == 0) {
        animState.wrapMode = WrapMode.Once;
        TCell[i].animation.Play(animState.name); // (animState);
        break;
    }
    j++;
}

this.animation.Play();
BCell.animation.Play();

else if (state == 5) {  // TCell to many cell
    if (!isTCellClone) {
        isTCellClone = true;
        TCell[0].GetComponent<COHelperTCell>().clone();
        // TCell[0].animation.Play();
    }
}
else if (state == 6) {  // TCell Link to BCell
    // if (!isLinkToBCell) {
    //    isLinkToBCell = true;
    //}
if (!isBCell) {
    isBCell = true;
    int j = 0;
    foreach (AnimationState animState in BCell.animation) {
        if (j == 1) {
            animState.wrapMode = WrapMode.Once;
            BCell.animation.Play(animState.name); // (animState);
            break;
        }
        j++;
    }
    // PlasmaCell.animation.Play();
}

} else if (state == 7) {
    if (!isBCell03) {
        isBCell03 = true;
        GameObject.Find("B").GetComponent<MorphControl>().isStart = true;
    }
}

} else if (state > 8) {
    if (!isReset) {
        isReset = true;
    }
using UnityEngine;
using System.Collections;

public class COBendSet : MonoBehaviour {
    public float Angle;
    public Vector3[] originalPos;
    // Use this for initialization
    void Start () {
        originalPos = new Vector3[7];
        for (int i = 0; i < transform.childCount; i++) {
            transform.GetChild(i).GetComponent<MegaBend>().angle = Angle;
            originalPos[i] = transform.GetChild(i).position;
        }
    }

    public void resetPosition() {
        // Instantiate(reset);
        // Destroy(transform.parent.gameObject);
    }
}
```csharp
for (int i = 0; i < transform.childCount; i++) {
    transform.GetChild(i).position = originalPos[i];
}

// Update is called once per frame
void Update()
{
    animation[tag].speed = 0;
}
```

---

**EventAnimStop.js**

```javascript
function StopAnim (tag: String)
{
    animation[tag].speed = 0;
}
```

---

**COHandRayCast.cs**

```csharp
using UnityEngine;
using System.Collections;

public class COHandRayCast : MonoBehaviour
{
    RaycastHit hit;
    Vector3 fwd = new Vector3(0, 0, -1.0f);
    public bool hitOnPad = false;
    public bool isReadyToRay = true;

    // Use this for initialization
```
void Start () {

}

// Update is called once per frame
void Update() {

  // if (isReadyToRay) {
    int myMask = 1<<8;
    if (Physics.Raycast(transform.position, fwd, out hit, 100, myMask)) {
      Debug.DrawLine(transform.position, hit.point);
      if (hit.collider.gameObject.tag == "scalePad") {
        hitOnPad = true;
      } else {
        hitOnPad = false;
      }
    } else {
      int myMask = 1<<9;
      if (Physics.Raycast(transform.position, fwd, out hit, Mathf.Infinity, myMask)) {
        Transform storedTransform = hit.collider.gameObject.transform;
    } //}

}
Debug.Log(storedTransform.name);
}
*/

COSideMenu.cs ...

function StopAnim (tag: String) {
    using UnityEngine;
    using System.Collections;

    public class COSideMenu : MonoBehaviour {
        public Vector3 workPos;
        public Vector3 restPos;
        // Use this for initialization
        public bool isActive = false;
        void Start () {
        }

        public void active () {
            isActive = true;
        }

        public void deactivate () {

isActive = false;

// Update is called once per frame
void Update () {
    if (isActive) {
        if (transform.position.x != workPos.x) {
            transform.position = Vector3.Lerp(transform.position, workPos,
                                                Time.deltaTime * 4.0f);
            if (Mathf.Abs(transform.position.x - workPos.x) < 3.0f) {
                transform.position = workPos;
            }
        }
    } else {
        if (transform.position.x != restPos.x) {
            transform.position = Vector3.Lerp(transform.position, restPos,
                                                Time.deltaTime * 4.0f);
            if (Mathf.Abs(transform.position.x - restPos.x) < 3.0f) {
                transform.position = restPos;
            }
        }
    }
}
using UnityEngine;
using System.Collections;
using System;

public class playStepbyStep : MonoBehaviour {
    public Transform click;
    public string event_t;
    bool m_isStay = false;
    public CODNALPEI dnalpei;
    COTouchMiniCircle touch;
    bool isDebug = false;

    public bool isBackBT = false;

    void Start () {
        renderer.material.color=Color.red;
        isDebug = GameObject.FindGameObjectWithTag("staticOpenNI").GetComponent<COGeneralData>().debug;
    }

    void OnCollisionEnter(Collision collision)
    {
        if (collision.gameObject.tag=="hand")
        {
            // Code
        }
    }
}
{ 
   renderer.material.color=Color.green;
   if (isDebug) {
      if (collision.gameObject.tag=="hand") {
         touch = collision.gameObject.transform.GetChild(0).GetComponent<COTouchMiniCircle>();
         touch.startCheck();
      }
   }
}

void OnCollisionStay(Collision collision)
{
   renderer.material.color=Color.green;
   if (GameObject.FindGameObjectsWithTag("staticOpenNI").GetComponent<
      COGeneralData>().isFreezeButton) {
      return;
   }
}

if (collision.gameObject.tag=="hand") {
   if (isDebug) {
      if (touch!=null) {
         if (touch.isClickDone) {

```
m_isStay = true;
touch.endAction();
touch = null;
}

/*
timeCount += Time.deltaTime;
if (timeCount >= 1.0f) {
timeCount = 0;
isHit = true;
}
*/

else {
        COHandClickButton handClick = collision.gameObject.GetComponent
                        <COHandClickButton>();
        if (!handClick.isAlreadyStart()) {
                    handClick.StartCheckClickButton();
        }
        else {
                    if (handClick.isClick()) {
                                handClick.StopCheckClickButton();
                                m_isStay = true;
                                handClick.StopCheckClickButton();
                                GameObject obj = GameObject.FindGameobjectWithTag("MainCamera");

obj.audio.Play();

}

}

}

}

}

}

void OnCollisionExit(Collision collision)
{
    renderer.material.color=Color.red;
    m_isStay = false;
    if (isOpen) {
        if (touch != null) {
            touch.endAction();
            touch = null;
        }
    }
}

// Update is called once per frame
void Update () {
    if(m_isStay)
    {

CODNALPEI.cs...

```csharp
using UnityEngine;
using System.Collections;

public class CODNALPEI : MonoBehaviour {
    public bool isOnlyDNA = false;
    public bool isReverse = false;

    public TextMesh describeTxt;

    string s1 = "1. Add polymer solution \n in organic solution";
```
string s2 = "2. Add DNA–LPEI\n in water solution.";
string s3 = "3. Spin.";
string s4 = "4. Add water.";
string s5 = "5. Spin and nanoparticles\n are created.";
string s6 = "6. Pour nanoparticle\n in a breaker.";
string s7 = "7. Organic evaporates.";
string s8 = "8. Transfer nanoparticle\n into centrifuge.";
string s9 = "9. Spin to concentrate \n the nanoparticles..";

//

//
Vector3 centerPos = new Vector3(4.03f, 1.32f, 23.85f);
Vector3 restPos = new Vector3(33.03f, 1.32f, 23.85f);

//

//
public Transform GlassOne;
public Transform GlassTwo;
public Transform GlassThree;

//

public int step = 0;
public float timeCount = 0;
float timeMax = 1.5f;

public Transform polymerPar;
    public Transform polymerLayer;
    bool endStepOne = false;

    public Transform dnaPar;
    public Transform dnaLayer;
    bool endStepTwo = false;

    public Transform polyDnaLayer;
    public Transform stick;
    public Transform tinyBlueDot;
    bool endStepThree = false;

    public Transform waterPar;
    public Transform waterLayer;
    bool endStepFour = false;

    public Transform tinyDot;
    public Transform wpLayer;
    bool endStepFive;

    public Transform bigDot;
    bool endStepSix = false;

    public Transform g2BigDot;
public Transform g2TinyDot;

public Transform eva;

public Transform g2WPLayer;

public Transform g2WPLayerRemain;

public Transform spinObj;

bool endStepSeven = false;

bool endStepEight = false;

public Transform g3Dot;

public Transform g3DotRemain;

bool endStepNine = false;

public Transform dnalpeiText;

// Use this for initialization

void Start () {
  if (isReverse) {
    setToEnd();
  }
}

void enablePlayBT() {
  GameObject.Find("play").transform.localPosition = new Vector3(-7.4f, 5.79f, -0.055f);
GameObject.Find("restart").transform.localPosition = new Vector3(7.87f, 5.79f, -0.055f);
}
// Update is called once per frame

public void setToEnd()
{
    step = 9;
    // describeTxt.text = s8;
    endStepNine = false;
    g3Dot.particleSystem.renderer.enabled = true;
    g3DotRemain.particleSystem.renderer.enabled = false;
    for(int i = 0; i < dnalpeiText.childCount; i++)
    {
        dnalpeiText.GetChild(i).renderer.enabled = false;
    }

    // change position
    // }
    // else if (step == 8)
    { 
        describeTxt.text = s7;
        endStepEight = true;
        GlassTwo.localPosition = restPos;
        GlassThree.localPosition = centerPos;
        // }
    // else if (step == 7)
    {
        describeTxt.text = s6;
        endStepSeven = true;
    }
g2TinyDot.particleSystem.renderer.enabled = false;
g2WPayer.renderer.enabled = true;

// change position back

//

// else if (step == 6) {
   describeTxt.text = s5;
   endStepSix = true;
   GlassOne.localPosition = restPos;
   GlassTwo.localPosition = restPos;
//}

// else if (step == 5) {
   endStepFive = true;
   describeTxt.text = s4;
   bigDot.particleSystem.renderer.enabled = true;
   polyDnaLayer.renderer.enabled = false;
   wpLayer.renderer.enabled = true;
//}

// else if (step == 4) {
   endStepFour = true;
   describeTxt.text = s3;
   waterLayer.renderer.enabled = true;
//}

// else if (step == 3) {
   endStepThree = true;
   describeTxt.text = s2;
   tinyDot.particleSystem.renderer.enabled = true;
polyDnaLayer.renderer.enabled = true;
dnaLayer.renderer.enabled = false;
polymerLayer.renderer.enabled = false;

//
// else if (step == 2) {
endStepTwo = true;
describeTxt.text = s1;
dnaLayer.renderer.enabled = true;
//}
// else if (step == 1) {
endStepOne = true;
describeTxt.text = " ";
polymerLayer.renderer.enabled = true;
//}
describeTxt.text = s8;
}

public void backward() {
if (step == 10) {
}
else if (step == 9) {
describeTxt.text = s8;
endStepNine = false;
endStepEight = false;
g3Dot.particleSystem.renderer.enabled = true;
}
g3DotRemain.particleSystem.renderer.enabled = false;

for (int i = 0; i < dnalpeiText.childCount; i++) {
    dnalpeiText.GetChild(i).renderer.enabled = false;
}

// change position

else if (step == 8) {
    describeTxt.text = s7;
    endStepEight = false;
    endStepSeven = false;
    GlassTwo.localPosition = centerPos;
    GlassThree.localPosition = restPos;
}

else if (step == 7) {
    describeTxt.text = s6;
    endStepSeven = false;
    endStepSix = false;
    g2TinyDot.particleSystem.renderer.enabled = true;
    g2WPLayer.renderer.enabled = true;
    // change position back
}

else if (step == 6) {
    describeTxt.text = s5;
    endStepSix = false;
    endStepFive = false;
GlassOne.getLocalPosition = centerPos;
GlassTwo.getLocalPosition = restPos;

} else if (step == 5) {
  endStepFive = false;
  endStepFour = false;
  describeTxt.text = s4;
  bigDot.particleSystem.renderer.enabled = false;
  polyDnaLayer.renderer.enabled = true;
  wpLayer.renderer.enabled = false;
}
else if (step == 4) {
  endStepFour = false;
  endStepThree = false;
  describeTxt.text = s3;
  waterLayer.renderer.enabled = false;
  tinyDot.particleSystem.renderer.enabled = false;
}
else if (step == 3) {
  endStepThree = false;
  endStepTwo = false;
  if(isOnlyDNA) {
    s2 = "2.Add DNA \nin water solution.";
  }
  describeTxt.text = s2;
  tinyDot.particleSystem.renderer.enabled = false;
polyDnaLayer.render.enabled = false;
dnaLayer.render.enabled = true;
polymerLayer.render.enabled = true;
dnaPar.particleSystem.render.enabled = true;
}
else if (step == 2) {
    endStepTwo = false;
    describeTxt.text = s1;
    dnaLayer.render.enabled = false;
}
else if (step == 1) {
    endStepOne = false;
    describeTxt.text = " ";
    polymerLayer.render.enabled = false;
}
step--;
}

void Update () {
if (Input.GetKeyDown(KeyCode.N)) {
    step++;
    print("next");
}
if (Input.GetKeyDown(KeyCode.B)) {

backward();
print("back");

if(step==0) {
}
else if (step ==1) {
    //polymerPar and Layer
describeTxt.text = s1;
    if (!endStepOne) {
        if(!polymerPar.particleSystem.renderer.enabled) {
            polymerPar.particleSystem.renderer.enabled = true;
        }
        //polymerPar.particleSystem.Play();
timeCount += Time.deltaTime;
        if(timeCount >= timeMax) {
            polymerLayer.renderer.enabled = true;
            polymerPar.particleSystem.renderer.enabled = false;
            timeCount = 0;
            endStepOne = true;
            enablePlayBT();
        }
    }
}
else if (step == 2) {
    //dna lpei
if(isOnlyDNA) {
    s2 = "2. Add DNA in water solution."
}
describeTxt.text = s2;
if(!endStepTwo) {
    if(!dnaPar.particleSystem.renderer.enabled) {
        dnaPar.particleSystem.renderer.enabled = true;
    }
}
timeCount += Time.deltaTime;
if(timeCount >= timeMax) {
    dnaPar.particleSystem.renderer.enabled = false;
    dnaLayer.renderer.enabled = true;
    timeCount = 0;
    endStepTwo = true;
    enablePlayBT();
}
else if (step == 3) {
    // stickSpin
    describeTxt.text = s3;
    if(!endStepThree) {
        if(!stick.animation.isPlaying) {
            stick.animation.Play();
        }
    }
timeCount += Time.deltaTime;

// tiny blue dot & polymer-dna layer
if (timeCount >= timeMax) {
    if (!tinyDot.particleSystem.renderer.enabled) {
        tinyDot.particleSystem.renderer.enabled = true;
        polyDnaLayer.renderer.enabled = true;
        dnaLayer.renderer.enabled = false;
        polymerLayer.renderer.enabled = false;
        stick.animation.Stop();
        timeCount = 0;
        endStepThree = true;
        if (isOnlyDNA) {
            enablePlayBT();
        } else {
            enablePlayBT();
        }
    }
}
else if (step == 4) {
    // poly vinyl alcohol (water)
describeTxt.text = s4;

if(!endStepFour) {
    if (!waterPar.particleSystem.renderer.enabled) {
        waterPar.particleSystem.renderer.enabled = true;
    }

    timeCount += Time.deltaTime;
    if(timeCount >= timeMax) {
        waterLayer.renderer.enabled = true;
        waterPar.particleSystem.renderer.enabled = false;
        timeCount = 0;
        endStepFour = true;
        enablePlayBT();
    }
}

else if (step == 5) {
    // spin
    describeTxt.text = s5;
    if(!endStepFive) {
        if(!stick.animation.isPlaying) {
            stick.animation.Play();
        }

        timeCount += Time.deltaTime;
    }
/tinybluedot & polymer-dnalpeilayer

if (timeCount >= timeMax) {
    if (!bigDot.particleSystem.renderer.enabled) {
        bigDot.particleSystem.renderer.enabled = true;
        polyDnaLayer.renderer.enabled = false;
        wpLayer.renderer.enabled = true;
        stick.animation.Stop();
        timeCount = 0;
        endStepFive = true;
        enablePlayBT();
    }
}

else if (step == 6) {
    //Glass2 -> center
    describeTxt.text = s6;
    if (!endStepSix) {
        Vector3 TempG1 = Vector3.Lerp(GlassOne.localPosition, restPos, Time.deltaTime*3);
        GlassOne.localPosition = TempG1;
        Vector3 TempG2 = Vector3.Lerp(GlassTwo.localPosition, centerPos, Time.deltaTime*3);
        GlassTwo.localPosition = TempG2;
        timeCount += Time.deltaTime;
    }
if (timeCount >= timeMax) {
    endStepSix = true;
    timeCount = 0;
    enablePlayBT();
}

} else if (step == 7) {
    // evaporate
    describeTxt.text = s7;
    if (!endStepSeven) {
        if (!eva.particleSystem.isPlaying) {
            eva.particleSystem.Play();
            spinObj.animation.Play();
        }
        timeCount += Time.deltaTime;
    }

    if (timeCount >= timeMax*2) {
        if (g2TinyDot.particleSystem.renderer.enabled) {
            // g2BigDot.particleSystem.renderer.enabled = false;
            g2TinyDot.particleSystem.renderer.enabled = false;
            g2WPLayer.renderer.enabled = false;
            g2WPLayerRemain.renderer.enabled = true;
        }
    }
spinObj.animation.Stop();
endStepSeven = true;
timeCount = 0;
enablePlayBT();
}
}
}

else if (step == 8) {
    //Glass3 -> center
describeTxt.text = s8;
    if (!endStepEight) {
        Vector3 TempG2 = Vector3.Lerp(GlassTwo.localPosition, restPos, 
            Time.deltaTime*3);
        GlassTwo.localPosition = TempG2;
        Vector3 TempG3 = Vector3.Lerp(GlassThree.localPosition, 
            centerPos, Time.deltaTime*3);
        GlassThree.localPosition = TempG3;

        timeCount += Time.deltaTime;

        if (timeCount >= timeMax) {
            endStepEight = true;
            timeCount = 0;
            enablePlayBT();
        }
    }
else if (step == 9) {

    //spin glass3
    describeTxt.text = s9;
    if (!endStepNine) {
        if (!GlassThree.GetChild(0).transform.animation.isPlaying) {
            GlassThree.GetChild(0).transform.animation.Play();
        }
    }

    timeCount += Time.deltaTime;

    if (timeCount >= timeMax*2) {
        endStepNine = true;
        GlassThree.GetChild(0).transform.animation.Stop();
        g3Dot.particleSystem.renderer.enabled = false;
        g3DotRemain.particleSystem.renderer.enabled = true;
        timeCount = 0;
        for (int i = 0; i < dnalpeiText.childCount; i++) {
            dnalpeiText.GetChild(i).renderer.enabled = true;
        }
        enablePlayBT();
    }
}
else if (step == 10) {
    if (isOnlyDNA) {
        Application.LoadLevel("C4_DNA_Zoom");
    }
    else {
        Application.LoadLevel("C4_DNALPEI_Zoom");
    }
}

Zig.cs developed by ZigFu. For more information, please visit http://zigfu.com.

using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Zig : MonoBehaviour {
    public ZigInputType inputType = ZigInputType.Auto;
    public ZigInputSettings settings = new ZigInputSettings();
    public List<GameObject> listeners = new List<GameObject>();
    public bool Verbose = true;

    void Awake () {

#if UNITY_WEBPLAYER

#endif

#if UNITY_EDITOR

Debug.LogError("Depth camera input will not work in editor when target platform is Webplayer. Please change target platform to PC/Mac standalone.");

return;
#endif

ZigInput.InputType = inputType;
ZigInput.Settings = settings;
ZigInput.Instance.AddListener(gameObject);

void notifyListeners(string msgname, object arg) {
    //SendMessage(msgname, arg, SendMessageOptions.DontRequireReceiver);
    //Zig.cs doesn’t send message to self
    for (int i = 0; i < listeners.Count; ) {
        GameObject go = listeners[i];
        if (go) {
            go.SendMessage(msgname, arg, SendMessageOptions.DontRequireReceiver);
            i++;
        }
    } else {

170
void Zig_UserFound(ZigTrackedUser user) {
    if (Verbose) Debug.Log("Zig: Found user " + user.Id);
    notifyListeners("Zig_UserFound", user);
}

void Zig_UserLost(ZigTrackedUser user) {
    if (Verbose) Debug.Log("Zig: Lost user " + user.Id);
    notifyListeners("Zig_UserLost", user);
}

void Zig_Update(ZigInput zig) {
    notifyListeners("Zig_Update", zig);
}
public class ZigEngageSingleUser : MonoBehaviour {
    public bool SkeletonTracked = true;
    public bool RaiseHand;

    float timeCount = 0;
    bool isLost = true;
    public float logoffTime = 300.0f;

    public List<GameObject> EngagedUsers;

    Dictionary<int, GameObject> objects = new Dictionary<int, GameObject>();

    public ZigTrackedUser engagedTrackedUser { get; private set; } 

    void Start() {
        // make sure we get zig events
        ZigInput.Instance.AddListener(gameObject);
    }

    void EngageUser(ZigTrackedUser user) {
        if (null == engagedTrackedUser) {
            engagedTrackedUser = user;
            foreach (GameObject go in EngagedUsers) user.AddListener(go);
        }
    }
}
SendMessage("UserEngaged", this,
SendMessageOptions.DontRequireReceiver);
}
}

void DisengageUser(ZigTrackedUser user) {
    if (user == engagedTrackedUser) {
        foreach (GameObject go in EngagedUsers) user.RemoveListener(
go);
        engagedTrackedUser = null;
        SendMessage("UserDisengaged", this,
        SendMessageOptions.DontRequireReceiver);
    }
}

void Zig_UserFound(ZigTrackedUser user) {
    timeCount = 0;
isLost = false;

    // create gameObject to listen for events for this user
    GameObject go = new GameObject("WaitForEngagement" + user.Id);
    go.transform.parent = transform;
    objects[user.Id] = go;

    // add various detectors & events
if (RaiseHand) {
    ZigHandRaiseDetector hrd = go.AddComponent<
        ZigHandRaiseDetector>();
    hrd.HhandRaise += delegate {
        EngageUser(user);
    };
}

// attach the new object to the new user
user.AddListener(go);

void Zig_UserLost(ZigTrackedUser user) {
    isLost = true;
    DisengageUser(user);
    Destroy(objects[user.Id]);
    objects.Remove(user.Id);
}

void Zig_Update(ZigInput zig) {
    if (SkeletonTracked && null == engagedTrackedUser) {
        foreach (ZigTrackedUser trackedUser in zig.TrackedUsers.
            Values) {
            if (trackedUser.SkeletonTracked) {
                EngageUser(trackedUser);
            }
        }
    }
public void Reset() {
    if (null != engagedTrackedUser) {
        DisengageUser(engagedTrackedUser);
    }
}

public void Update() {
    if (isLost) {
        timeCount += Time.deltaTime;
        if (timeCount > logoffTime) {
            timeCount = 0;
            PlayerData pda = GameObject.Find("PlayerData").GetComponent<PlayerData>();
            if (pda) {
                pda.endSession();
            }
        }
    }
}
class ZigEngageSingleSession : MonoBehaviour {

    public GameObject EngagedUser;
    public List<GameObject> listeners = new List<GameObject>();
    Dictionary<int, GameObject> objects = new Dictionary<int, GameObject>();

    public bool StartOnSteady = false;
    public bool StartOnWave = true;
    public bool RotateToUser = true;

    // bounds in mm
    public Vector3 SessionBoundsOffset = new Vector3(0, 250, -300);
    public Vector3 SessionBounds = new Vector3(1500, 700, 1000);

    ZigTrackedUser engagedTrackedUser;

    void Start() {
        // make sure we get zig events
    }
ZigInput.Instance.AddListener(gameObject);
}

bool EngageUser(ZigTrackedUser user) {
    if (null == engagedTrackedUser) {
        engagedTrackedUser = user;
        if (null != EngagedUser) user.AddListener(EngagedUser);
        SendMessage("UserEngaged", this,
        SendMessageOptions.DontRequireReceiver);
        return true;
    }
    return false;
}

bool DisengageUser(ZigTrackedUser user) {
    if (user == engagedTrackedUser) {
        if (null != user) { // force end session for a user on disengage
            objects[user.Id].GetComponent<ZigHandSessionDetector>().EndSession();
        }
        if (null != EngagedUser) user.RemoveListener(EngagedUser);
        engagedTrackedUser = null;
        SendMessage("UserDisengaged", this,
        SendMessageOptions.DontRequireReceiver);
        return true;
    }
}
void Zig_UserFound(ZigTrackedUser user) {
    // create gameObject to listen for events for this user

    // if (user.SkeletonTracked) {
    //    print (user.Skeleton[(int)ZigJointId.Head].Rotation.eulerAngles.z);
    // }

    GameObject go = new GameObject("WaitForEngagement" + user.Id);
    go.transform.parent = transform;
    objects[user.Id] = go;

    ZigHandSessionDetector hsd = go.AddComponent<
        ZigHandSessionDetector>();

    hsd.SessionBounds = SessionBounds;
    hsd.SessionBoundsOffset = SessionBoundsOffset;
    hsd.StartOnSteady = StartOnSteady;
    hsd.StartOnWave = StartOnWave;
    hsd.RotateToUser = RotateToUser;

    hsd.SessionStart += delegate {

    }
Debug.Log("EngageSingleSession: Session start");

if (EngageUser(user)) {
    foreach (GameObject listener in listeners) {
        hsd.AddListener(listener);
    }
}

hsd.SessionEnd += delegate {
    Debug.Log("EngageSingleSession: Session end");
    if (DisengageUser(user)) {
        foreach (GameObject listener in listeners) {
            hsd.RemoveListener(listener);
        }
    }
};

user.AddListener(go);
```
using UnityEngine;
using System;
using System.Collections;
using System.Collections.Generic;

public class ZigSkeleton : MonoBehaviour
{
    public Transform Head;
    public Transform Neck;
    public Transform Torso;
    public Transform Waist;
    public Transform LeftCollar;
    public Transform LeftShoulder;
    public Transform LeftElbow;
    public Transform LeftWrist;
    public Transform LeftHand;
    public Transform LeftFingertip;
    
    if (null != engagedTrackedUser) {
        DisengageUser(engagedTrackedUser);
    }
}
```

ZigSkeleton.cs developed by ZigFu. For more information, please visit http://zigfu.com.
public Transform RightCollar;
public Transform RightShoulder;
public Transform RightElbow;
public Transform RightWrist;
public Transform RightHand;
public Transform RightFingertip;

public Transform LeftHip;
public Transform LeftKnee;
public Transform LeftAnkle;
public Transform LeftFoot;

public Transform RightHip;
public Transform RightKnee;
public Transform RightAnkle;
public Transform RightFoot;

public bool mirror = false;
public bool UpdateJointPositions = false;
public bool UpdateRootPosition = false;
public bool RotateToPsiPose = false;
public float RotationDamping = 30.0f;
public Vector3 Scale = new Vector3(0.001f, 0.001f, 0.001f);
public Vector3 PositionBias = Vector3.zero;
```csharp
private Transform[] transforms;
private Quaternion[] initialRotations;
private Vector3 rootPosition;

ZigJointId mirrorJoint(ZigJointId joint)
{
    switch (joint) {
    case ZigJointId.LeftCollar:
        return ZigJointId.RightCollar;
    case ZigJointId.LeftShoulder:
        return ZigJointId.RightShoulder;
    case ZigJointId.LeftElbow:
        return ZigJointId.RightElbow;
    case ZigJointId.LeftWrist:
        return ZigJointId.RightWrist;
    case ZigJointId.LeftHand:
        return ZigJointId.RightHand;
    case ZigJointId.LeftFingertip:
        return ZigJointId.RightFingertip;
    case ZigJointId.LeftHip:
        return ZigJointId.RightHip;
    case ZigJointId.LeftKnee:
        return ZigJointId.RightKnee;
    case ZigJointId.LeftAnkle:
        return ZigJointId.RightAnkle;
```
case ZigJointId.LeftFoot:
    return ZigJointId.RightFoot;

case ZigJointId.RightCollar:
    return ZigJointId.LeftCollar;

case ZigJointId.RightShoulder:
    return ZigJointId.LeftShoulder;

case ZigJointId.RightElbow:
    return ZigJointId.LeftElbow;

case ZigJointId.RightWrist:
    return ZigJointId.LeftWrist;

case ZigJointId.RightHand:
    return ZigJointId.LeftHand;

case ZigJointId.RightFingertip:
    return ZigJointId.LeftFingertip;

case ZigJointId.RightHip:
    return ZigJointId.LeftHip;

case ZigJointId.RightKnee:
    return ZigJointId.LeftKnee;

case ZigJointId.RightAnkle:
    return ZigJointId.LeftAnkle;

case ZigJointId.RightFoot:
    return ZigJointId.LeftFoot;

default:
    return joint;

    }

}

public void Awake()
{
    int jointCount = Enum.GetNames(typeof(ZigJointId)).Length;

    transforms = new Transform[jointCount];
    initialRotations = new Quaternion[jointCount];

    transforms[(int)ZigJointId.Head] = Head;
    transforms[(int)ZigJointId.Neck] = Neck;
    transforms[(int)ZigJointId.Torso] = Torso;
    transforms[(int)ZigJointId.Waist] = Waist;
    transforms[(int)ZigJointId.LeftCollar] = LeftCollar;
    transforms[(int)ZigJointId.LeftShoulder] = LeftShoulder;
    transforms[(int)ZigJointId.LeftElbow] = LeftElbow;
    transforms[(int)ZigJointId.LeftWrist] = LeftWrist;
    transforms[(int)ZigJointId.LeftHand] = LeftHand;
    transforms[(int)ZigJointId.LeftFingertip] = LeftFingertip;
    transforms[(int)ZigJointId.RightCollar] = RightCollar;
    transforms[(int)ZigJointId.RightShoulder] = RightShoulder;
    transforms[(int)ZigJointId.RightElbow] = RightElbow;
    transforms[(int)ZigJointId.RightWrist] = RightWrist;
transforms[(int)ZigJointId.RightHand] = RightHand;
transforms[(int)ZigJointId.RightFingertip] = RightFingertip;
transforms[(int)ZigJointId.LeftHip] = LeftHip;
transforms[(int)ZigJointId.LeftKnee] = LeftKnee;
transforms[(int)ZigJointId.LeftAnkle] = LeftAnkle;
transforms[(int)ZigJointId.LeftFoot] = LeftFoot;
transforms[(int)ZigJointId.RightHip] = RightHip;
transforms[(int)ZigJointId.RightKnee] = RightKnee;
transforms[(int)ZigJointId.RightAnkle] = RightAnkle;
transforms[(int)ZigJointId.RightFoot] = RightFoot;

// save all initial rotations
// NOTE: Assumes skeleton model is in "T" pose since all rotations are relative to that pose

foreach (ZigJointId j in Emm.GetValues(typeof(ZigJointId))) {
    if (transforms[(int)j]) {
        // we will store the relative rotation of each joint from the gameobject rotation
        // we need this since we will be setting the joint's rotation (not localRotation) but we
        // still want the rotations to be relative to our game object
        initialRotations[(int)j] = Quaternion.Inverse(transform.rotation) * transforms[(int)j].rotation;
    }
}
void Start()
{
    // start out in calibration pose
    if (RotateToPsiPose) {
        RotateToCalibrationPose();
    }
}

void UpdateRoot(Vector3 skelRoot)
{
    // +Z is backwards in OpenNI coordinates, so reverse it
    rootPosition = Vector3.Scale(new Vector3(skelRoot.x, skelRoot.y, skelRoot.z), doMirror(Scale)) + PositionBias;
    if (UpdateRootPosition) {
        transform.localPosition = (transform.rotation * rootPosition);
    }
}

void UpdateRotation(ZigJointId joint, Quaternion orientation)
{
    joint = mirror ? mirrorJoint(joint) : joint;
    // make sure something is hooked up to this joint
    if (!transforms[(int)joint]) {

return;
}

if (UpdateOrientation) {
    Quaternion newRotation = transform.rotation * orientation *
        initialRotations[(int)joint];
    if (mirror)
    {
        newRotation.y = -newRotation.y;
        newRotation.z = -newRotation.z;
    }
    transforms[(int)joint].rotation = Quaternion.Slerp(transforms[(int)
        joint].rotation, newRotation, Time.deltaTime * RotationDamping);
}

Vector3 doMirror(Vector3 vec)
{
    return new Vector3(mirror ? -vec.x : vec.x, vec.y, vec.z);
}

void UpdatePosition(ZigJointId joint, Vector3 position)
{
    joint = mirror ? mirrorJoint(joint) : joint;
    // make sure something is hooked up to this joint
    if (!transforms[(int)joint]) {
        return;
    }
if (UpdateJointPositions) {
    Vector3 dest = Vector3.Scale(position, doMirror(Scale)) -
    rootPosition;
    transforms[(int)joint].localPosition = Vector3.Lerp(transforms[(
        int)joint].localPosition, dest, Time.deltaTime *
    RotationDamping);
}
}

public void RotateToCalibrationPose()
{
    foreach (ZigJointId j in Emu.GetValues(typeof(ZigJointId))) {
        if (null != transforms[(int)j]) {
            transforms[(int)j].rotation = transform.rotation *
            initialRotations[(int)j];
        }
    }

    // calibration pose is skeleton base pose ("T") with both elbows
    // bent in 90 degrees
    if (null != RightElbow) {
        RightElbow.rotation = transform.rotation * Quaternion.Euler
        (0, -90, 90) * initialRotations[(int)ZigJointId.
        RightElbow];
    }
if (null != LeftElbow) {
    LeftElbow.rotation = transform.rotation * Quaternion.Euler
        (0, 90, -90) * initialRotations[(int)ZigJointId.
            LeftElbow];
}

public void SetRootPositionBias()
{
    this.PositionBias = -rootPosition;
}

public void SetRootPositionBias(Vector3 bias)
{
    this.PositionBias = bias;
}

void Zig_UpdateUser(ZigTrackedUser user)
{
    UpdateRoot(user.Position);
    if (user.SkeletonTracked) {
        foreach (ZigInputJoint joint in user.Skeleton) {
            if (joint.GoodPosition) UpdatePosition(joint.Id, joint.Position);
        }
    }
}
using UnityEngine;
using System;
using System.Collections.Generic;

public class ZigMapJointToSession : MonoBehaviour {
    public ZigJointId joint = ZigJointId.None;
    bool InSession;
    public ZigTrackedUser userTemp;

do void Zig_UpdateUser(ZigTrackedUser user) {
    gameObject.SendMessage("testSession", 3, SendMessageOptions.DontRequireReceiver);
    if (!InSession && user.SkeletonTracked && joint != ZigJointId.None) {
        InSession = true;
    }
}
userTemp = user;
    SendMessage("Session_Start", user.Skeleton[(int)joint].Position, SendMessageOptions.DontRequireReceiver);
}

if (InSession) {
    if (!user.SkeletonTracked) {
        SendMessage("Session_End", SendMessageOptions.DontRequireReceiver);
        InSession = false;
    }
    userTemp = null;
} else {
    print ("track");
    userTemp = user;
    if (GameObject.Find("hands")) {
        GameObject.Find("hands").GetComponent<COCheckHeadRotation>()
            .Session_headJoint(user.Skeleton[(int)ZigJointId.Head].Rotation.eulerAngles);
    }
    SendMessage("Session_headJoint", user.Skeleton[(int)ZigJointId.Head].Rotation.eulerAngles, SendMessageOptions.DontRequireReceiver);
    SendMessage("Session_Update", user.Skeleton[(int)joint].Position, SendMessageOptions.DontRequireReceiver);
}
```csharp
void Reset() {
    InSession = false;
}
```

ZigMapJointToSession.cs developed by ZigFu. For more information, please visit http://zigfu.com...

ZigHandSessionDetector.cs developed by ZigFu. For more information, please visit http://zigfu.com...

```csharp
using UnityEngine;

using System;

using System.Collections.Generic;

public class SessionStartEventArgs : EventArgs
{
    public Vector3 FocusPoint { get; private set; }

    public SessionStartEventArgs(Vector3 fp) {
        FocusPoint = fp;
    }
}
```

```csharp
public class SessionUpdateEventArgs : EventArgs
{
    public Vector3 FocusPoint { get; private set; }

    public SessionUpdateEventArgs(Vector3 fp) {
        FocusPoint = fp;
    }
}
```

```csharp
public class SessionStartEventArgs : EventArgs
{
}
```

```csharp
public class SessionUpdateEventArgs : EventArgs
{
}
```
public Vector3 HandPoint { get; private set; }

public SessionUpdateEventArgs(Vector3 hp) {
    HandPoint = hp;
}

public class ZigHandSessionDetector : MonoBehaviour {
    public bool StartOnSteady = false;
    public bool StartOnWave = true;
    public bool RotateToUser = true;

    public List<GameObject> listeners = new List<GameObject>();

    public Vector3 SessionBoundsOffset = new Vector3(0, 250, -300);
    public Vector3 SessionBounds = new Vector3(1500, 700, 1000);

    GameObject leftHandDetector;
    GameObject rightHandDetector;
    ZigJointId jointInSession;
    Vector3 focusPoint;
    ZigTrackedUser trackedUser;
    bool InSession;
    Bounds currentSessionBounds;

    public event EventHandler<SessionStartEventArgs> SessionStart;
public event EventHandler<SessionUpdateEventArgs> SessionUpdate;

public event EventHandler SessionEnd;

protected virtual void OnSessionStart(Vector3 focusPoint) {
    notifyListeners("Session_Start", focusPoint);
    if (null != SessionStart) {
        SessionStart.Invoke(this, new SessionStartEventArgs(
            focusPoint));
    }
}

protected virtual void OnSessionUpdate(Vector3 handPoint) {
    notifyListeners("Session_Update", handPoint);
    if (null != SessionUpdate) {
        SessionUpdate.Invoke(this, new SessionUpdateEventArgs(
            handPoint));
    }
}

protected virtual void OnSessionEnd() {
    notifyListeners("Session_End", null);
    if (null != SessionEnd) {
        SessionEnd.Invoke(this, new EventArgs());
    }
}
void Awake() {

    leftHandDetector = new GameObject("LeftHandDetector");
    leftHandDetector.transform.parent = gameObject.transform;
    ZigMapJointToSession leftMap = leftHandDetector.AddComponent<
        ZigMapJointToSession>();
    leftMap.joint = ZigJointId.LeftHand;

    rightHandDetector = new GameObject("RightHandDetector");
    rightHandDetector.transform.parent = gameObject.transform;
    ZigMapJointToSession rightMap = rightHandDetector.AddComponent<
        ZigMapJointToSession>();
    rightMap.joint = ZigJointId.RightHand;

    if (StartOnSteady) {
        ZigSteadyDetector steadyLeft = leftHandDetector.AddComponent<
            ZigSteadyDetector>();
        steadyLeft.Steady += delegate(object sender, EventArgs ea) {
            CheckSessionStart((sender as ZigSteadyDetector).
                steadyPoint, ZigJointId.LeftHand);
        };

        ZigSteadyDetector steadyRight = rightHandDetector.
            AddComponent<ZigSteadyDetector>();
        steadyRight.Steady += delegate(object sender, EventArgs ea) {
            CheckSessionStart((sender as ZigSteadyDetector).
                steadyPoint, ZigJointId.RightHand);
        };

    }
}
CheckSessionStart((sender as ZigSteadyDetector).
steadyPoint, ZigJointId.RightHand);
}

if (StartOnWave) {
    ZigWaveDetector waveLeft = leftHandDetector.AddComponent<
    ZigWaveDetector>();
    waveLeft.Wave += delegate(object sender, EventArgs ea) {
        Debug.Log("Wave from left");
        CheckSessionStart((sender as ZigWaveDetector).wavePoint,
        ZigJointId.LeftHand);
    };

    ZigWaveDetector waveRight = rightHandDetector.AddComponent<
    ZigWaveDetector>();
    waveRight.Wave += delegate(object sender, EventArgs ea) {
        Debug.Log("Wave from right");
        CheckSessionStart((sender as ZigWaveDetector).wavePoint,
        ZigJointId.RightHand);
    };
}

public void AddListener(GameObject listener) {
    if (listeners.Contains(listener)) return;
}
listeners.Add(listener);

if (InSession) {
    listener.SendMessage("Session_Start", focusPoint,
            SendMessageOptions.DontRequireReceiver);
}

public void RemoveListener(GameObject listener) {
    if (InSession) {
        listener.SendMessage("Session_End", SendMessageOptions.
            DontRequireReceiver);
    }
    listeners.Remove(listener);
}

void Zig_Attach(ZigTrackedUser user) {
    trackedUser = user;
    user.AddListener(leftHandDetector);
    user.AddListener(rightHandDetector);
}

void Zig_UpdateUser(ZigTrackedUser user) {
    if (InSession) {
        // get hand point for this frame, rotate if neccessary
        Vector3 hp = userSkeleton[(int)jointInSession].Position;
        // ...
    }
}
if (RotateToUser) hp = RotateHandPoint(hp);

// make sure hand point is still within session bounds
currentSessionBounds.center = (RotateToUser) ?

currentSessionBounds.center += SessionBoundsOffset;
if (!currentSessionBounds.Contains(hp)) {
    EndSession();
    return;
}
OnSessionUpdate(hp);
}
}

void Zig_Detach(ZigTrackedUser user) {
    user.RemoveListener(leftHandDetector);
    user.RemoveListener(rightHandDetector);
    EndSession();
    trackedUser = null;
}

public void EndSession()
{
    if (InSession) {
        InSession = false;
        OnSessionEnd();
    }
void CheckSessionStart(Vector3 point, ZigJointId joint) {
    if (InSession) { Debug.Log("CheckSessionStart when already in session, leaving"); return; }

    Vector3 boundsCenter = (RotateToUser) ? RotateHandPoint(
    boundsCenter += SessionBoundsOffset;
    currentSessionBounds = new Bounds(boundsCenter, SessionBounds);
    Vector3 fp = (RotateToUser) ? RotateHandPoint(point) : point;
    if (currentSessionBounds.Contains(fp)) {
        focusPoint = fp;
        jointInSession = joint;
        InSession = true;
        OnSessionStart(fp);
    }
}

void notifyListeners(string msgname, object arg) {
    SendMessage(msgname, arg, SendMessageOptions.DontRequireReceiver);
    for (int i = 0; i < listeners.Count; ) {
        GameObject go = listeners[i];
if (go) {
    go.SendMessage(msgname, arg, SendMessageOptions.DontRequireReceiver);
    i++;
}
else {
    listeners.RemoveAt(i);
}

Vector3 RotateHandPoint(Vector3 handPoint)
{
    //TODO: Smoothing on CoM (so sudden CoM changes won’t mess with
    the hand
    // point too much)
    Vector3 rotateTarget = trackedUser.Position.normalized;
    // use line between com and sensor as Z, decompose rotation into
    // rotations around the Y and X axes
    Vector3 firstTarget = rotateTarget;
    // project onto XZ plane
    firstTarget.y = 0;
    firstTarget = firstTarget.normalized;
    // find rotation around the X axis
Quaternion xRotation = Quaternion.FromToRotation(rotateTarget, firstTarget);

// rotation around Y axis
Quaternion yRotation = Quaternion.FromToRotation(firstTarget, Vector3.forward);

return yRotation * xRotation * handPoint;
}

ZigDepthViewer.cs developed by ZigFu. For more information, please visit http://zigfu.com...

using UnityEngine;

using System.Collections;

public enum ZigResolution
{
    QQVGA_160x120,
    QVGA_320x240,
    VGA_640x480,
}

public class ResolutionData
{
    protected ResolutionData(int width, int height)
    {
    }
Width = width;
Height = height;

public int Width { get; private set; }
public int Height { get; private set; }

public static ResolutionData FromZigResolution(ZigResolution res)
{
    switch (res) {
        default: // fallthrough - default to QQVGA
            case ZigResolution.QQVGA_160x120:
                return new ResolutionData(160, 120);
            case ZigResolution.QVGA_320x240:
                return new ResolutionData(320, 240);
            case ZigResolution.VGA_640x480:
                return new ResolutionData(640, 480);
    }
}

public class ZigDepthViewer : MonoBehaviour {
    public Renderer target;
    public ZigResolution TextureSize = ZigResolution.QQVGA_160x120;
    public Color32 BaseColor = Color.yellow;
    public bool UseHistogram = true;
    Texture2D texture;
ResolutionData textureSize;

float[] depthHistogramMap;
Color32[] depthToColor;
Color32[] outputPixels;

public int MaxDepth = 10000; //DO NOT MODIFY IN RUNTIME!!

// Use this for initialization
void Start () {
    if (target == null) {
        target = renderer;
    }
    textureSize = ResolutionData.FromZigResolution(TextureSize);
    texture = new Texture2D(textureSize.Width, textureSize.Height);
    texture.wrapMode = TextureWrapMode.Clamp;
    depthHistogramMap = new float[MaxDepth];
    depthToColor = new Color32[MaxDepth];
    outputPixels = new Color32[textureSize.Width * textureSize.Height];
    ZigInput.Instance.AddListener(gameObject);

    if (null != target) {
        target.material.mainTexture = texture;
    }
}

void UpdateHistogram(ZigDepth depth)
            int i, numOfPoints = 0;

            System.Array.Clear(depthHistogramMap, 0, depthHistogramMap.Length);

            short[] rawDepthMap = depth.data;

            int depthIndex = 0;

            // assume only downscaling
            // calculate the amount of source pixels to move per column and row in
            // output pixels

            int factorX = depth.xres / textureSize.Width;
            int factorY = ((depth.yres / textureSize.Height) - 1) * depth.xres;

            for (int y = 0; y < textureSize.Height; ++y, depthIndex += factorY) {
                for (int x = 0; x < textureSize.Width; ++x, depthIndex += factorX) {
                    short pixel = rawDepthMap[depthIndex];

                    if (pixel != 0) {
                        depthHistogramMap[pixel]++;
                        numOfPoints++;
                    }
                }
            }
depthHistogramMap[0] = 0;

if (numOfPoints > 0) {
    for (i = 1; i < depthHistogramMap.Length; i++) {
        depthHistogramMap[i] += depthHistogramMap[i - 1];
    }
    depthToColor[0] = Color.black;
    for (i = 1; i < depthHistogramMap.Length; i++) {
        float intensity = (1.0f - (depthHistogramMap[i] / numOfPoints));
        // depthHistogramMap[i] = intensity * 255;
        depthToColor[i].r = (byte)(BaseColor.r * intensity);
        depthToColor[i].g = (byte)(BaseColor.g * intensity);
        depthToColor[i].b = (byte)(BaseColor.b * intensity);
        depthToColor[i].a = 255; // (byte)(BaseColor.a * intensity)
    }
}

void UpdateTexture(ZigDepth depth)
{
    short[] rawDepthMap = depth.data;
    int depthIndex = 0;
    int factorX = depth.xres / textureSize.Width;
int factorY = ((depth.yres / textureSize.Height) - 1) * depth.xres;

// invert Y axis while doing the update
for (int y = textureSize.Height - 1; y >= 0; --y, depthIndex += factorY) {
    int outputIndex = y * textureSize.Width;
    for (int x = 0; x < textureSize.Width; ++x, depthIndex += factorX, ++outputIndex) {
        outputPixels[outputIndex] = depthToColor[rawDepthMap[depthIndex]];
    }
}

texture.SetPixels32(outputPixels);
texture.Apply();

void Zig_Update(ZigInput input)
{
    if (UseHistogram) {
        UpdateHistogram(input.Depth);
    }
    else {
        //TODO: don’t repeat this every frame
        depthToColor[0] = Color.black;
        for (int i = 1; i < MaxDepth; i++) {
            float intensity = 1.0f - (i/(float)MaxDepth);
        }
    }
}
    depthHistogramMap[i] = intensity * 255;
    depthToColor[i].r = (byte)(BaseColor.r * intensity);
    depthToColor[i].g = (byte)(BaseColor.g * intensity);
    depthToColor[i].b = (byte)(BaseColor.b * intensity);
    depthToColor[i].a = 255; // (byte)(BaseColor.a * intensity);

    UpdateTexture(ZigInput.Depth);

    void OnGUI() {
        if (null == target) {
            GUI.DrawTexture(new Rect(Screen.width - texture.width - 10,
                                   Screen.height - texture.height - 10, texture.width,
                                   texture.height), texture);
        }
    }
}