AN INTEGRATED DISPLAY AND DETECTION 3D INTERACTION SYSTEM

[1] *Cheng-Hao Chu* (*祝成豪*), [2] *Chiou-Shann Fuh* (*傅楸善*)

[1] Dept. of Electrical Engineering,
National Taiwan University, Taiwan
E-mail: nickchu35@gmail.com
[2] Dept. of Computer Science and Information Engineering,
National Taiwan University, Taiwan
E-mail: fuh@csie.ntu.edu.tw

ABSTRACT

No matter it is in the industry or academy. Human Computer Interface (HCI) is a trending topic. Microsoft had put in tremendous effort on all kinds of HCI applications. There are multiple ways to develop an instinctive HCI. One can simply use hands, or with the aids of other objects. 3D displays have also become more and more popular among consumers and the industries. Though there are many obstacles to overcome, such as the stress it causes to human eyes and cognitive system, or the limitation of the space where viewers are able receive clearer and more realistic images. When combining these two fields together, the possibility of what can be achieved is truly fascinating. However, the difficulties also add up. In this paper, we will make use of several instruments and programs to help us focus on the major problems that we wish to inspect on.

Keywords *Human Computer Interface; Microsoft Kinect; Unity 3D; Human Detection; Hand Detection; 3D Interaction; 3D Display; Cognitive Science;*

1. INTRODUCTION

When communicating with computers or devices using an interface. The Major problem is always the precision of detection. Many companies released many products for detection, e.g. Microsoft Kinect, Leap Motion, ASUS Xtion, and so on. Each has its own limits and advantages over others. And each suits different kinds of application scenarios, the prices are different also.

When people are watching a 3D television or go to the theaters to watch 3D movies, the experience is totally different from each other. Most people would agree that the 3D experience they received in the theater is much better than at home. It is probably because that when people are in a theater. All they perceive is the images on the giant screen. Other distractions which are not on the screen will be eliminated by turning off the lights. That is why when a person uses his phone during the movie, the light of the phone will be extremely uncomfortable. If we wish to bring 3D display technology into daily lives, it is no doubt that the display will have to be limited to a sufficiently small area. However, while stereoscopic displays are more convenient and direct, we choose to use a 3D television with active glasses to achieve a better 3D experience and to reduce the cost (compared with stereoscopic display).

In our experiment, we wish to create a scene as realistic as it could be. We will establish multiple experiments to observe the factors that we think might affect our cognitive system, and give us more realistic feelings about a virtual scene. Also, the stress that 3D displays also put on our cognitive system has to be reduced to minimum. We will try to adjust the setups of our virtual scenes to generate an environment where user can clearly focus on the objects without going through much stress. We will see how the disparity of objects and the convergence of eyes affect the performance of virtual 3D scenes and the degree of stress.

We will use Unity 3D program to generate the virtual scene we need in our experiments. Therefore we have to solve the problems between two coordinate systems. One is the Kinect coordinate system, the other one is the Unity 3D coordinate system.

Our main problems can then be summarized into four targets:

1. Does Microsoft Kinect device provide enough precision to provide us smooth interaction?
2. What are the factors that make a virtual scene more realistic than others?

3. How to maintain a clear 3D image of object while simultaneously does not impose huge stress on the user's cognitive system?
4. How do we transform the coordinates from Kinect system to Unity system, or vice versa?

2. CREATION OF VIRTUAL SCENES

In our experiments, we use Unity 3D to create some simple scenes. It's really easy to use Unity to create virtual environments since there are tremendous amount of 3D models which can be used in the projects. The programming languages it accepts are JavaScript and C#. We use C# in our experiments.

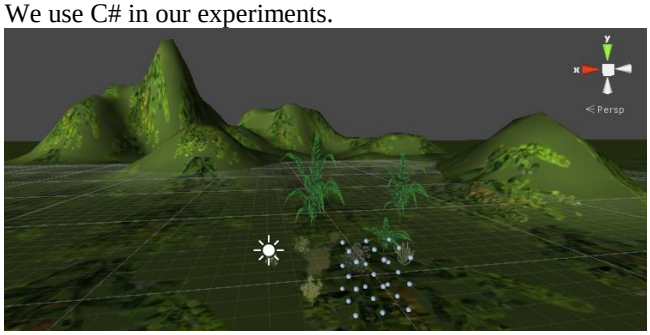

Fig. 1: Simple Unity 3D scene containing multiple balls in the middle with trees and mountains in the back.

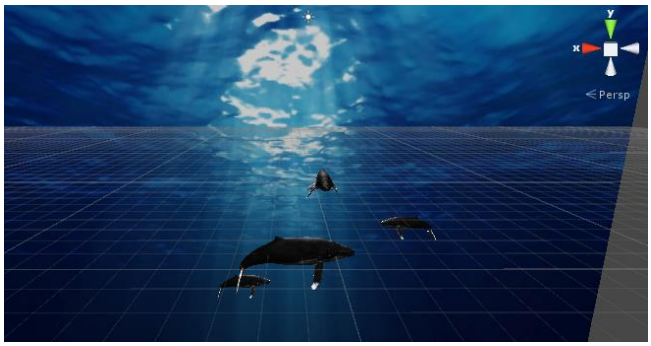

Fig. 2: Unity 3D scene of undersea whales.

One of the scenes we created was a scene with many balls. And the ball in the middle has a z coordinate of zero. Originally, we wanted a scene with bubbles since the interaction between human and bubbles are quite straightforward. Bubbles can float in the air freely. And when they touch any other objects, they will pop and then disappear. However, the texture of bubbles and optical phenomena caused by them are too hard to simulate. We simply use balls instead of bubbles as our interaction targets. Those balls will float in the air and move in random directions. If a ball is too close to the boundaries of our views or too close to the cameras, it will pop. And another new ball will be created near the origin with some random offsets.

Our second scene is an underwater scene with several whales swimming around. The main purpose of creating this scene is that we wish to see whether a scene is more realistic than the other will affect our perception and give us a better 3D experience. We found a quite realistic underwater background with quite high definition. Along with the models of blue whale, the scene is surprisingly beautiful. Another difference between animals and bubbles is that people expect animals to move their body parts while bubbles only need to float from one place to another.

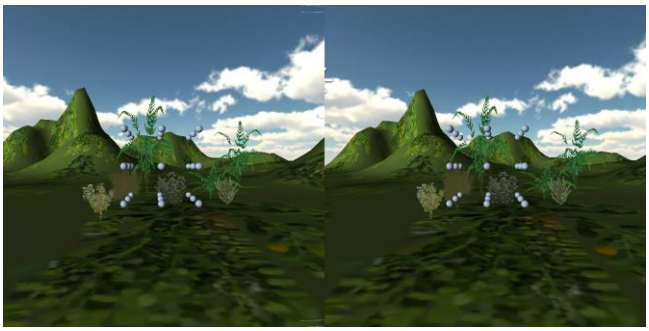

Fig. 3: Balls scene seen in a left/right eyes fashion.

Fig. 4: Whales scene seen in a left/right eyes fashion.

3. 3D PERFORMACE OF UNITY VIRTUAL SCENES

As we have shown in the previous part. In Unity, we can just create a virtual environment and put cameras in it. By setting the camera correctly, we can split our screens into two halves. With a 3D television which can be set to left and right 3D mode, we can then observe the scenes through our active glasses in 3D. Here's a brief explanation to the construction of 3D objects.

When we look at one object. Our left and right eyes always receive different images. If we only use on eye and switch to the other one very quickly. We will feel that some objects have moved. And the closer the object is, the longer the distance they move. The distance between a same object in two images perceived by eyes is called disparity. When the disparity is huge, our brain will assume the object is closer than the objects with smaller disparity.

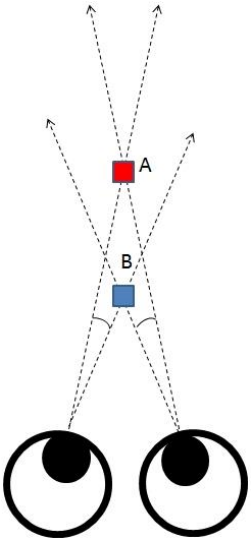

Fig. 5: Point A and B generate different disparity.

In Unity, we can easily configure our settings to split our scenes into two. We have several convenient ways to adjust the disparity of objects in the scenes.

1. Change the camera distance. Camera distance in Unity is the equivalent of eye distance of human. The bigger the camera distance is, the larger the disparity will be.

2. Modify the convergence of eyes. In our experiences, focusing on different objects does not change our perceptions of depth. However, since Unity generates 3D vision from two different scenes. As long as there are some differences between the scenes, our perception will be affected.

We tried three kinds of setting to observe the 3D effect performance. First, we let the cameras fix on the infinity plane, therefore making their views parallel to each other. Second, we modified the angels of cameras to make their views converge at the z = 0 plane. Third, we put a moving object into the scene and let the cameras focus on that object dynamically. Our expectation was a clear image of the object and the changing of depth perception. However, the result was terribly far from it. The movement of the background was too severe and we were no longer able to perceive the scene as 3D anymore

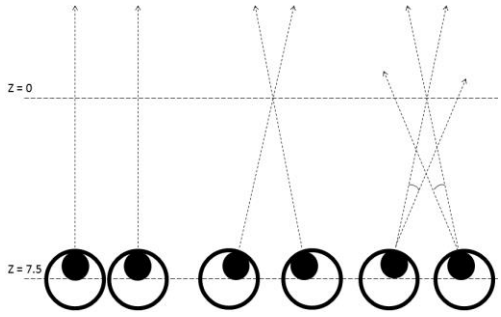

Fig. 6: Demonstration of our three settings.

Overall, the middle setup in Figure 6 gave us the best experience. Later on, we tried another setting. Because the bubbles in our scene move dynamically and will have positive z coordinates for most of the time, focusing on the z = 0 plane became less reasonable. Therefore, we observed the bubbles' coordinates and we found out that the average z coordinate in the long term is about two. We then setup a new scene with cameras focusing on the z = 2 plane and expected that we should get clear images of all the balls no matter it is in front or in the back of the plane. The result was the best out of them all.

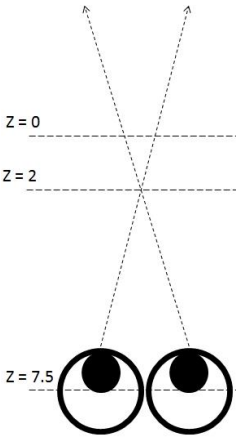

Fig. 7: Cameras focusing on z = 2 plane.

4. KINECT IMPLEMENTATION

Kinect has a major drawback in our scenario though. It was designed for playing sport games without consoles. Therefore it assumes people will be standing in a relatively far distance. However, this is not the case in all experiments where we assume the users will be close to the television and Kinect. In spite of this, we still chose Kinect over other motion detection devices such as Leap Motion because there are even more limitations. In our scenario, the information we need is not much. Position of hands or fingers is the first thing need. We may also need the depth of our heads to change the settings of Unity scene. Otherwise, if the users move their heads forward or backward, the experience will become awkward and not realistic enough.

The positions of human main bones can be easily acquired through the Kinect Software Development Kit (SDK). Even though we wish to reach a higher precision and allow the interaction between human fingers and virtual object, we started with simple bone parts. We used the wrist position to substitute for the hand position. Using Kinect SDK, we can easily detect human bodies and identify different body parts. The positions can then be acquired.

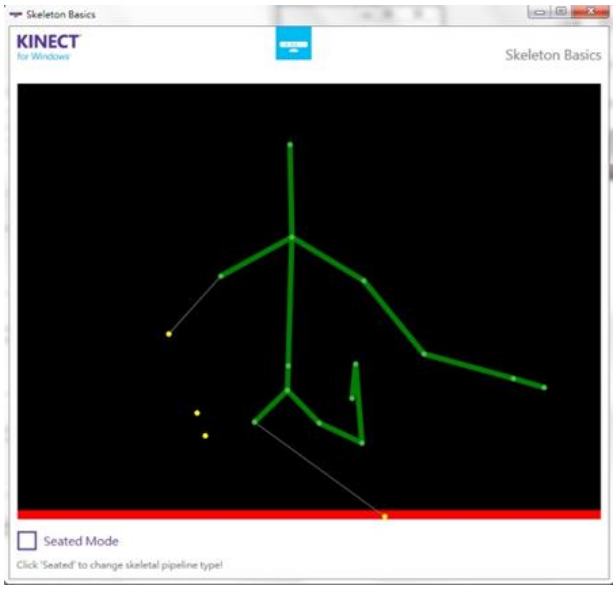

Fig. 8: A skeleton detected by Kinect.

Start with Kinect skeleton information, we can combine with other algorithms to achieve hand tracking feature. In our scenario, the positions of hands are mostly in front of human body for a proper distance. We should be able to discard most of the information behind the hands without causing noticeable error.

4.1. Fingers tracking

Since we are still using Microsoft Kinect for Xbox 360, we have to implement our own hand tracking system from scratch.

First, we have to extract the hands part from the whole depth map. We set some simple thresholds derived from human knowledge to achieve this goal.

1. The depths of hands should always be smaller than the depths of wrists, which can be detected by Kinect SDK. Therefore we can remove the information which has depth larger than wrists.
2. The area below our wrists can be ignored, since we assume our hands will always be above our wrists.

These two restrictions can quite effectively remove the data that does not belong to the hands region.

The first restriction is achieved by detecting the depths of wrists using Kinect skeleton SDK. Set the bigger one of the two depths as a threshold. All pixels with depths larger than the threshold in the depth map will be set to zero value.

The second restriction is even simpler. After receive the positions of wrists. We set another threshold to the bigger of the two positions. Therefore everything below the wrists will then be excluded.

However, since the positions of wrists detected by Kinect is not stable and accurate enough. It will cause some problems. It might leave holes in our hands since it thinks the depths of our palms are larger than our wrists. We all know that those values are really close so it is actually a reasonable mistake. To make sure we receive a full and accurate shape of hands. We added some offsets to the judgments of depths. Anything with depths smaller than the depths of wrists plus an offset will be viewed as hands. Also, anything with positions lower than wrists plus an offset will be excluded.

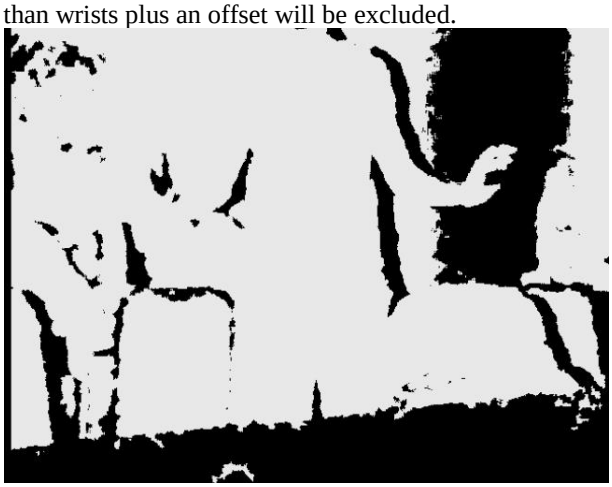
Fig. 9: Original depth map.

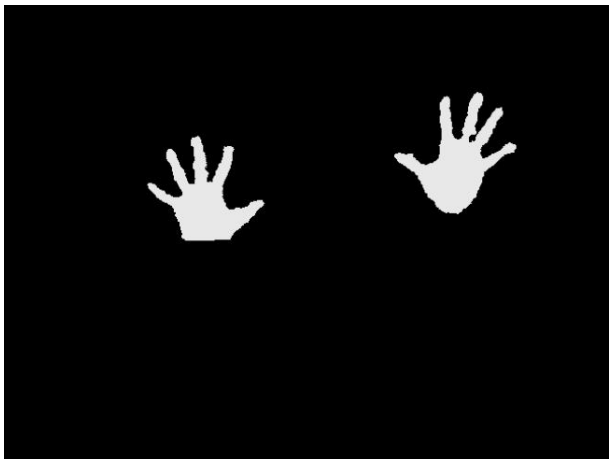
Fig. 10: Depth map with noise removed.

Now we have a map that only shows our hands. We want to get the positions of our fingertips. In order to do this, we need the contours of our hands. By examining the contours, we can calculate the approximate positions of our fingertips.

We choose an easy to implement algorithm to trace the contours, which is Square Tracing Algorithm. This is probably the simplest algorithm to find contours. Even though it does not have the ability to locate holes within contours, it will not affect our cases since we don't expect the presence of holes in our hands.

Some people might choose to combine OpenCV library with Kinect SDK. However, it requires including many other libraries. We choose to implement the algorithm ourselves to keep the project light.

4.2. Square Tracing Algorithm

The pseudo code of Square Tracing Algorithm is shown below:

```
Input: A square tessellation, T, containing a connected component P of black cells.
Output: A sequence B (b1, b2 ,..., bk) of boundary pixels i.e. the contour.
Begin
   Set B to be empty.
   From bottom to top and left to right scan the cells of T until a black pixel, s, of P is found.
   Insert s in B.
   Set the current pixel, p, to be the starting pixel, s.
   Turn left i.e. visit the left adjacent pixel of p.
   Update p i.e. set it to be the current pixel.
   While p not equal to s do
      If the current pixel p is black
        insert p in B and turn left (visit the left adjacent pixel of p).
        Update p i.e. set it to be the current pixel.
      else
        turn right (visit the right adjacent pixel of p).
        Update p i.e. set it to be the current pixel.
   end While
End
```

This algorithm is really easy to implement. On the other hand, it can cause problems easily. Our main goal of this project is not to solve those problems, so we added an assumption to our project, which is that two of our hands will appear separately in the left and right half of the space. This will make the process of finding contours much faster and easier.

4.3. Finding Fingertips

By examining the depth map that contains only the information of hands, we can use a simple algorithm to find the positions of fingertips, The pseudocode is shown below.

```
Input: A sequence A1 (a11, a12 ,..., a1k) of boundary pixels of left hand and a sequence A2 (a21, a22 ,..., a2k) of boundary pixels of right hand.
Output: A sequence B (b1, b2 ,..., bk) of fingertips coordinates.
Begin
   Set B to be empty.
   Set a degree threshold, a sample distance and a jumping distance.
   For every pixels p in A1
      Let the index of p in A1 be i.
      Let p2, p3 be the pixels with index i + (sample distance) and i + 2 * (sample distance).
      If the angle ∠(p)(p2)(p3) is smaller than the degree threshold, we assume p2 is a fingertip and add it to sequence B.
      Immediately jump to one other pixel which is determined by the jumping distance. If this hadn't been done, we would find multiple points for fingertips on one same finger.
   Do the same on A2.
End
```

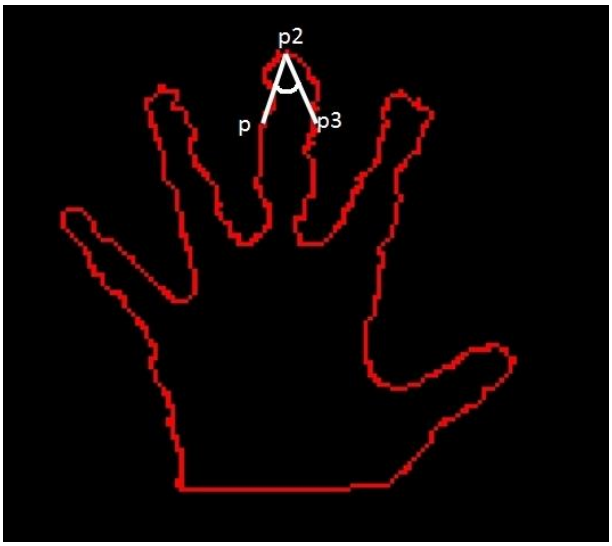

Fig. 11: Judgement of fingertips.

4.4. Communicating with Unity

All of the body parts we detected and data acquired in Kinect are sent to Unity through simple socket programming to achieve real-time interaction. However, since the position detected in Kinect is not stable and will fluctuate seriously, if we send all the raw data over to Unity. The interaction might lack of precision. Therefore we need some processing of the data before sending them.

The simple thought is to use some simple thresholds to exclude obvious detection errors and send the average of the coordinates detected over an amount of time.

Another problem that we have to handle about Kinect is the number of users. Kinect SDK can recognize up to six users facing the sensor. However, it can only track details of two users among them. In our experiments, we restricted the amount of users to only one user. Just to make the case simple.

5. COORDINATE TRANSFORM

There are totally three coordinate systems in our scenario.

1. Real-world coordinate system experienced by users.
2. Kinect coordinate system.
3. Unity Coordinate system.

We can simply ignore the real-world coordinate system, since even if we have knowledge about it wouldn't help much. For the positions of our heads and hands or fingers, we can just use their coordinates in the Kinect system.

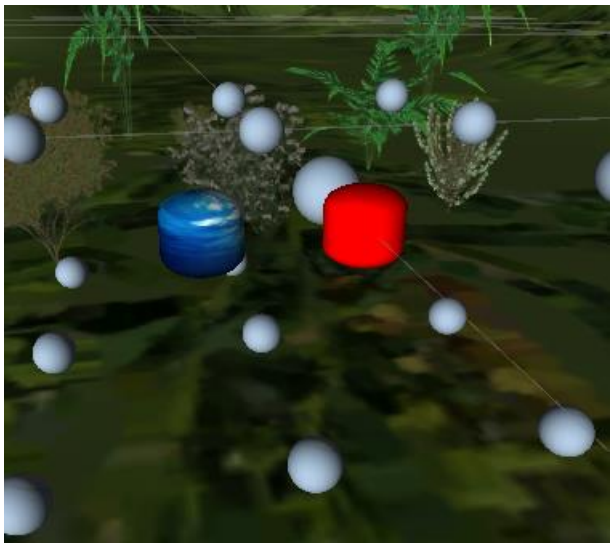

Fig. 12: Red and blue cylinders used to simulate right wrist and left wrist movements.

The tricky part is how to transform the detected finger positions into Unity coordinates. We created two small cylinders in the Unity scene of balls, and use the detected positions of our both wrists to control them. Therefore we can get a sense of the difference between positions of our real hands and the virtual hand, i.e., a sense of accuracy.

For now, we use the simplest method. Which is simply add some offsets and then scale the coordinates. For example, we send (x coordinate of hand in Kinect – 320)/125 to Unity as its x coordinate, the biggest weakness of this method is its lack of robustness. The relative position of Kinect and the display can never change.

6. EXPERIMENT

We designed a simple experiment to determine how serious can the accuracy of fingers detection affect our interaction. We created a new ball scene which shows only a single ball every time. After we touch it with one of our hands, the ball will disappear and then appear randomly at another position. We calculate the total time needed for a user to poke all twenty-seven balls. And allow them to score their objective experience on a scale of one to ten.

7. CONCLUSION

We successfully implemented a complete system that allows us to interact with virtual world using 3D television. This system doesn't have specific purposes, e.g., control a 3D model or play a game. We wished to create an environment that simply provides a natural experience of interaction. Even though the accuracy and the degree of realistic still need to improve, we verified that the system can already provide an experience which is natural enough.

REFERENCES

[1] F. Trapero Cerezo, "3D Hand and Finger Recognition using Kinect,".

[2] J.L. Raheja, A. Chaudhary, K. Singal, "Tracking of Fingertips and Centre of Palm using KINECT ", In proceedings of the 3rd IEEE International Conference on Computational Intelligence, Modelling and Simulation, Malaysia, 20-22 Sep, 2011, pp. 248-252.