Realistic 3D Projection System for Interactive Gaming and Visualization

Gal Koren
Massachusetts Institute of Technology
Cambridge, MA
koren@mit.edu

ABSTRACT

In this paper I describe the design and implementation of an interactive 3D-projection based visualization system, viable for gaming and virtual reality platforms. The system utilizes a Kinect motion sensor to track the user's position and manipulate the displayed viewport accordingly, so that the viewer sees different parts of the underlying scene depending on perspective. The addition of an IR-based gesture sensor affords another mode of interaction and haptic feedback for users' object-manipulating hand gestures.

Author Keywords

Haptic Feedback; Virtual Reality; Object Manipulation; Multimodal User Interfaces; 3D Projections; Interactive Visualizations; Microsoft Kinect; Leap Motion; Unity 3D

INTRODUCTION

Current research of new methods of creating immersive 3D experiences is extensive and on the rise. Last year, Microsoft introduced its RoomAlive[5] platform, which turns any room into an augmented reality environment using a number of Kinect sensors and projectors, tracking the position and movements of a person in the room to alter the projected game view to match their perspective. That environment, however, requires significant work and calibration in order to set up, and is quite cost-prohibitive for the average gamer or visualization enthusiast. Another example, the popular Oculus Rift, requires the user to wear a head-mounted display, which is not only expensive and bothersome, but can cause fatigue and nausea after extensive usage[3].

The objective of this project is to outline and implement a way to create a similarly compelling VR environment using a few commodity hardware components, without the need for multiple projectors, sensors, remote controls, or wearable displays. Moreover, its result is a projection that, to the user, looks like an illusion of a realistic 3D scene, with the objects inside it "popping out", or appearing to have a variably convincing physical presence. The system can be extended for many different purposes; this paper showcases a simplistic interactive game that makes use of gestural input to manipulate objects while providing a realistic 3D experience.

VIDEO DEMONSTRATIONS

The system can be viewed in usage in the following videos:

1. 3D projection adapting to viewer's perspective:
   `https://www.youtube.com/watch?v=4-7gTOIoPvs`

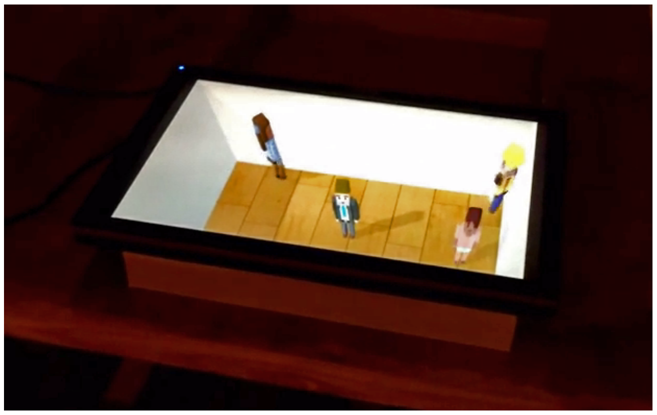

Figure 1: In this photo of the system in action, a user was looking at an animated 3D scene, displayed on a flat screen laid flat on a table. The scene used the user's head position in space to alter the viewport, so that the scene looks like it has physical depth.

2. Gesture-based interaction:
   `https://www.youtube.com/watch?v=DT2tWsNkTpY`

DESIGN AND ARCHITECTURE

Hardware

The projected image of the 3D scene is dependent on the variable perspective of the user, thus the system requires a sensor to be used for eye or head-tracking. Kinect for Windows is well suited for this purpose, as it can generate a depth map of the environment in front of it, which can be used by the system to recognize the user's head coordinates in space.

The second component is used to display the image, and comprises of a computer monitor or a projector and a blank surface. It is essential that the surface be flat, as any folds, wrinkles, or dirt can distort the image.

Lastly, a Leap Motion sensor is plugged to the monitor or to the computer running the program. The Leap allows the user to interact with the scene using manipulating gestures, in a more seamless fashion than with a joystick or remote control.

Software

Setting up the 3D environment

After evaluating the available development platforms and implementing the core functionality in Processing (see Section

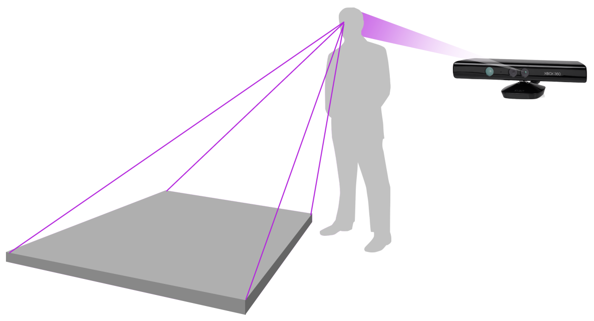

Figure 2: System hardware setup. The Kinect was placed behind the person walking about the surface that displays the 2-dimensional projection of the 3D scene (in this case, a simple flat screen monitor). The physical distances between the person and the Kinect, and the person and the monitor, were both carefully calculated, and the reliability of the projection relative to the viewer's varying perspective heavily depends on the precision and accuracy of these values.

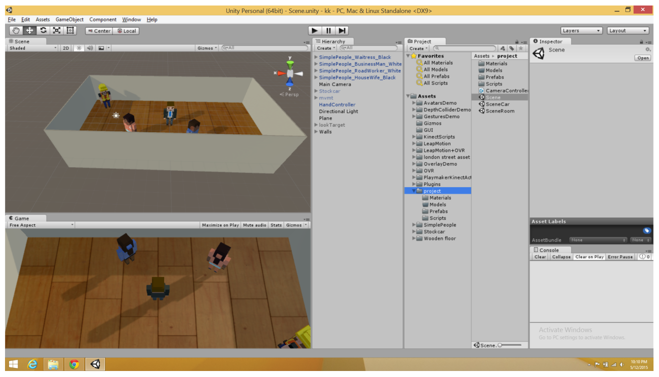

Figure 3: Screenshot of the main scene running inside of Unity. The top left viewport represents the editor mode, allowing the developer to replace and manipulate objects in the scene as well as change the lighting and camera position. The bottom left viewport displays the resulting game view, from the perspective of the camera currently used.

5), the chosen framework for the system was Unity 3D. Unity is a platform for creating elaborate 2D and 3D environments using a graphical user interface and C# scripting, and can facilitate many game developers' considerations by supplying them with ready-to-use modules for 3D object interaction and manipulation. For example, Unity has its own 3D physics engine, which can be used to animate objects similarly to their physical real-world equivalents. Objects have mass, just like in reality, which affects their interaction with other objects in the same scene - the environment that entails the current game mode or experience. Complicated interactions can be accomplished in Unity without writing a single line of code. While this quality makes Unity a particularly powerful platform to use for interactive experiences, it is also a tool that takes considerable amount of time and experience to learn and master well.

The main scene is called TinyRoom, a 3D space enclosed by 4 walls and a floor, with four 3D models of human-like characters placed inside the room. I applied each of the characters a walk behavior, and wrote a script that specified the behavior when two or more characters bumped into each other.

Each Unity scene must contain at least one camera, an object used by Unity as the eyes of the user, and restricts the viewport from the larger scene into a smaller window, mimicking how human eyes are only able to capture a certain angle and distance of our greater surrounding. Unity allows one to move and rotate the camera, adjust its field of view, as well as apply any number of scripts to extend its basic behavior. For this project, I created a single camera that I manipulated in runtime using a script called CameraController.

Head tracking

The system uses a Unity-based wrapper library[1] for the official Kinect SDK v1.8, and provides the developer with several classes, such as the KinectManager class, that use the Kinect's depth map to produce a skeleton joint labeling of the user's body. Once a human figure is detected and a skeleton mapping is generated, a script extracts the user's head coordinates, and smooths it by averaging the most recent head coordinates with a few of the last ones recorded. The number of coordinates averaged each time was determined by trial-and-error, as the more coordinates used for averaging, the smoother the overall signal became, but the system was slower to respond to the person's movements and head-position variability, which detracted from the user experience (see Section 4). After an appropriate threshold was found and set in Unity, the data is smoothed and subsequently calibrated to convert between real-world and screen-based coordinate system scales. The resulting coordinate tuple is then set to be the Unity scene camera's new position (as described before, the camera in Unity functions as the user's eyes in the real world).

3D projection manipulation

In computer graphics, the observer's field of view is defined using a number of intersecting frustum planes, with the region between the near and far clipping planes acting as the projected image (Figure 4). Usually, 3D games and movies make use of a symmetric, fixed system of frustum planes, with objects merely moving about the scene with no manipulation of the viewport relative to the viewer's perspective. Also referred to as on-axis projection, the frustums are all symmetric about the axis passing through the center of the projection surface and the camera. However, the objects in these environments rarely, if ever, appear to pop out of the screen, or to be convincingly present in the physical world, instead looking like flattened versions of 3D objects.

In order to create an illusion of a realistic 3D scene, the projected image needs to be transformed with every variation in the viewer's head position in space relative to the image. This can be achieved by manipulating the frustum planes to become asymmetric (also known as off-axis projection, as the

[1]`https://www.assetstore.unity3d.com/en/#!/content/7747`

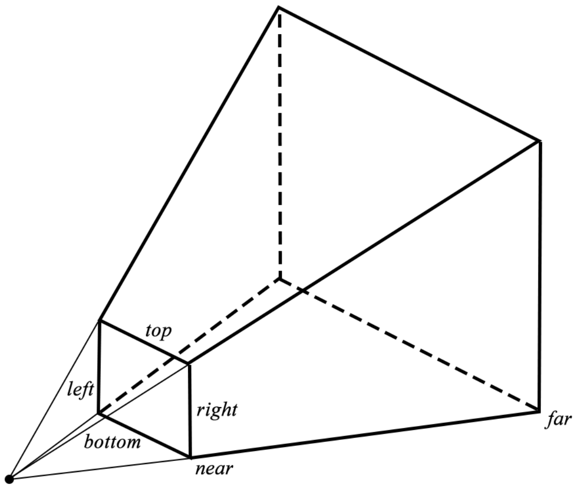

Figure 4: Diagram of a symmetric viewing frustum as part of a rectangular pyramid [2]. The frustum is confined by the bottom of the pyramid (the far clipping plane) and the truncating near plane, defined by the top, right, bottom, and left vectors[2]. The symmetry is about the z axis passing perpendicularly through the centers of the near and far planes.

center axis of the projection no longer aligns with the center axis of the surface). Asymmetric frustum planes can be calculated using a variable camera position (i.e. the viewer's eyes) and a fixed viewport, in order to create new projections of the same 3D scene from different perspectives. This process involves the manipulation of the magnitude and direction of four different vectors, originating from the camera origin and extending to each of the four corners of the viewport, which must be located in the region of the pyramind as in Figure 4 between the near and far clipping planes. This ultimately causes the projected image to skew as the viewer moves about the projection.

Typically in computer graphics, a 2D projection of a 3D model uses a projection matrix to define the viewing frustum (in a symmetrical frustum plane system, this is the identity matrix), and hence in order to alter the projection of the model, one has to multiply the projection matrix by a transformation matrix. In researching this problem, I was able to find a number of academic and non-academic resources that outlined the algebra involved in calculating the perspective projection[6][4], as well as basic Unity resources that implement it[1], all of which utilize the same transformation matrix T built into the OpenGL projection matrix-calculating `glFrustum` function:

$$T = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Figure 5: t, r, b, l refer to the frustum's top, right, bottom, and left vectors respectively, as shown in Figure 4. n and f are scalars indicating the distances on the z axis from the camera to the near and far clipping planes respectively.

Further improvements were made for the projection to appear realistic, such as moving the projected image (translation) and rotating it (rotation) appropriately as the viewer moves about the projection, by using a different set of transformation matrices.

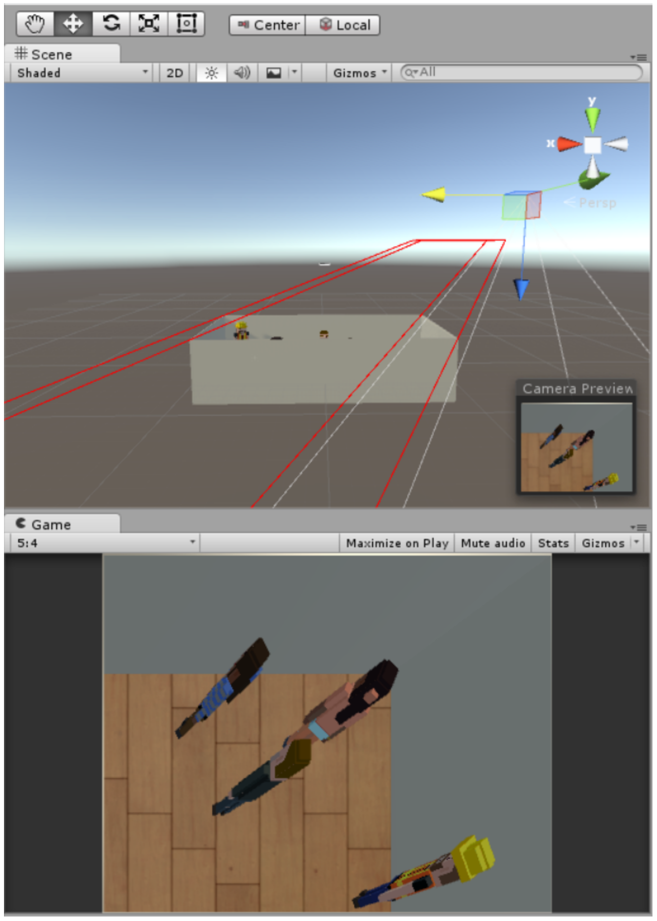

Figure 6: Screenshot of the Unity program exemplifying the frustum when the viewer looks at the projection from the right side (bottom left viewport is horizontally flipped). The four frustum vectors (top, left, right, bottom) as well as the near clipping plane are in red. The bottom left viewport in Unity shows the projected image that the user sees. Note how from this perspective, the characters look flat and stretched.

Gesture-based human-interface interaction

One of the requirements of a modern gaming environment is the affordance of interaction with the objects in the scene. To demonstrate the system's ability to facilitate gesture-based interaction, which is currently making strides in the gaming and VR industries, I decided to allow the user to move one of the characters in the TinyRoom scene by picking them up and placing them in different areas of the room.

This idea led me to explore different methods of gesture-based interaction. Because of the Kinect's placement behind the user and farther away from the projection (as shown in Figure 2), as well as the Kinect's difficulty tracking precise hand gestures, I decided to integrate the Leap Motion sensor and attach it directly to the monitor displaying the scene.

Using the Leap official SDK for Unity, I applied the hand gesture recognition technique from the class's Project 4 (*Battleship*) by writing a script that could control a character's placement once the user performed a grab or pinch gesture above the Leap. As with Project 4, I used averaging to smooth out the grab/pinch strength signal as well as the hand location signal reported by the Leap to minimize jitters and improve accuracy.

EVALUATION

As the system includes a few connected electronic components placed within exactly measured distances apart, it took me between 15 and 20 minutes to set up the hardware and run the program for user testing to take place. I set up the test at a large, unpopulated open space at the MIT Media Lab (both to allow enough room for the system's hardware setup, and to prevent the Kinect from detecting background motions of people not participating in the user testing).

Performance evaluation

During the first stage of the test, I explained to each of the test subjects the purpose and motivation of the system. I asked each subject to enter the Kinect's range of vision, and move about the screen for 30 seconds while looking at it. Though I didn't give them specific instructions on how to move and where, I asked them to report any seemingly unexpected behavior.

The results matched my initial expectation; all of the users reported that the system at times lagged in adapting the projection to their perspective as they moved, especially if their bodies weren't directly in front of the Kinect, and much more so when their movement was faster. Two of them reported obvious jitters in the image, i.e. a projection image got stuck for a number of seconds and then jumped to a new projection. The occurrence of these phenomena was unexpected by the users and was measured at a frequency of roughly 3 malfunctions per test (or 1 for every 10 seconds).

All the users reported that whenever the system adapted incorrectly to their perspectives, they found the projections to be less credible, but that the occurrence of this behavior was infrequent enough that it did not significantly detract from the overall experience. They were all visibly stunned at first glance by the accuracy of the projections and reported that, while they were aware that they were looking at a computer monitor, the image they were looking at was convincingly real for the majority of the experiment.

At the second stage, each user was asked to "grab" one of the characters, move it above the virtual space, and place it back into the TinyRoom scene. Watching their behavior from the side, I could tell that initially, some of them tried to grab a character by performing a grab or pinch gesture directly above the character, which failed to produce results as the Leap was placed on the corner of the monitor, out of range of the display. They quickly realized the problem, and continued to grab the character from above the Leap, which succeeded.

Users responded that the ability to interact with the scene using real gestures greatly enhanced the experience, but also that controlling the characters indirectly, by grabbing them from above the Leap, was unnatural and confusing. One of the users suggested mounting the Leap sensor directly above the monitor upside-down, such that the area covered by the Leap would include the entire perimeter of the screen, thus allowing the user to grab the characters directly from their location in the virtual space.

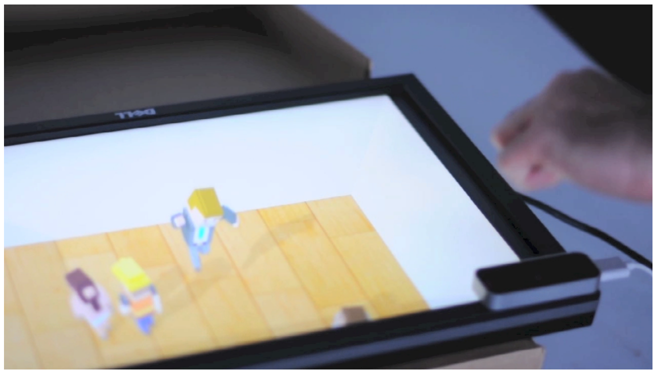

Figure 7: Example of a user grabbing and moving a character during the user testing, demonstrating the system's haptic feedback capabilities.

Usability evaluation

Afterwards, I asked the users if they found the system compelling for playing video games, such as GTA, Call of Duty, or Super Smash Bros. Although they weren't all video game players, they each said that the system would be an exciting opportunity to enhance the current mode of 3D gaming, which tends to be performed using a simplistic projection model of 3D scenes that doesn't adapt to the user's real world perspective of the game.

WORKFLOW AND CHALLENGES

Prior to this project, I had very limited knowledge of computer graphics or 3D-rendering, and I had to learn the fundamentals independently throughout the second half of semester. I began coding the project in Processing, and during my in-class user testing session I demoed a functioning, albeit very buggy, version of this system that I coded entirely in Processing. After thinking more deeply about the motivation behind this project, I decided that Processing is not an ideal platform, primarily as it lacks adequate support for object shadows, an integral part of creating realistic 3D environments, and it is not meant for developing games or interaction on a larger scale. In contrast, Unity is a development platform used primarily for creating 3D games, and so I spent considerable time studying the application's extensive UI functionality and API. Switching over to Unity was undoubtedly a wise decision, as it allowed me to take the project to a much more interesting and applicable direction.

FUTURE WORK

As a result of the user testing, I noted that mounting the Leap sensor upside-down above the projected image could facilitate a direct manipulating gesture when the user tries to grab and move the characters. I have not tested the Leap's accuracy in this upside-down mode, which could potentially present challenges. I would also like to create more compelling games for this platform, such as ones that involve navigating through space using an avatar, or perhaps adapt an existing video game like Super Smash Bros, where the characters are controlled using hand gestures with the Leap sensor.

CONCLUSION

Working on this project over the course of this term has greatly increased my knowledge of human-computer interaction through body tracking and gesture recognition, as well as presented an opportunity for me to explore a new area of interest, computer graphics. It was an ambitious task for a solo term project, but the system I created is a complete and usable proof-of-concept implementation of interactive 3D projections that adapt to their viewer's changing perspective over time.

REFERENCES

1. Cg programming/unity/projection for virtual reality, camera script. `http://en.wikibooks.org/wiki/Cg_Programming/Unity/Projection_for_Virtual_Reality`.
2. MSDN Documentation: What Is a View Frustum? `https://msdn.microsoft.com/en-us/library/ff634570.aspx`.
3. Dredge, S. Oculus warns Sony to solve motion sickness before launching a VR headset. *The Guardian* (2014).
4. Garstka, J., and Peters, G. View-dependent 3D Projection using Depth-Image-based Head Tracking. *8th IEEE International Workshop on ProjectorCamera Systems* (2011).
5. Jones, B., Sodhi, R., Murdock, M., Mehra, R., Benko, H., and Wilson, A. D. e. a. Roomalive: Magical experiences enabled by scalable, adaptive projector-camera units. In *UIST14*, ACM Press (2014).
6. Kooima, R. Determining frustum extents. *Louisiana State University* (2008).