Implementation of Headtracking and 3D Stereo with Unity and VRPN for Computer Simulations

Matthew A. Noyes*

University of Rochester, Rochester, NY, 14627

This paper explores low-cost hardware and software methods to provide depth cues traditionally absent in monocular displays. The use of a VRPN server in conjunction with a Microsoft Kinect and/or Nintendo Wiimote to provide head tracking information to a Unity application, and NVIDIA 3D Vision for retinal disparity support, is discussed. Methods are suggested to implement this technology with NASA's EDGE simulation graphics package, along with potential caveats. Finally, future applications of this technology to astronaut crew training, particularly when combined with an omnidirectional treadmill for virtual locomotion and NASA's ARGOS system for reduced gravity simulation, are discussed.

Nomenclature

ARGOS	Active Response Gravity Offload System
CAD	Computer-Assisted Design
CAVE	Cave Automatic Virtual Environment
DLL	Dynamic-Link Library
DOUG	Dynamic Onboard Ubiquitous Graphics
EDGE	Engineering DOUG Graphics for Exploration
EVA	Extra-Vehicular Activity
FOV	Field of View
GPU	Graphics Processing Unit
HMD	Head Mounted Display
HUD	Heads Up Display
LED	Light Emitting Diode
LOS	Line-of-Sight
ODT	Omni-Directional Treadmill
UIVA	Unity Indie VRPN Adapter
VR	Virtual Reality
VRPN	Virtual Reality Peripheral Network

*JSC Engineering Co-Op, Spacecraft Software Engineering

I. Introduction

Monocular displays provide most depth cues required to project a three-dimensional scene onto a two-dimensional surface. These cues are adequate for most types of operational and entertainment-related applications. However, the lack of real-world motion parallax and binocular cues dampens users' immersion, which dramatically impacts the efficacy of operational training [1]. Binocular cues can be simulated easily enough with polarized or active-shutter glasses and special monitors, producing an effect known as "3D stereo" or "stereopsis," while real-world motion parallax requires the integration of headtracking technologies. Adding real-world motion parallax provides a sense of true 3D when the user is moving relative to the screen; binocular cues provide depth when the user is stationary. A 1993 study [2] found that headtracking provides a much greater sense of immersion than 3D stereo alone, while the combination of the two provides significant reduction in search task errors. As the scope of human spaceflight advances to regions outside timely communication with ground control, the ability to handle complex tasks and unexpected problems becomes even more critical. It is therefore essential that training exercises provide as much realism as possible.

Immersion-driven technologies also have operational roles. Spacecraft windows require additional mass and can be a critical point of failure should leaks occur. However, astronaut crews consistently prefer windowed to non-windowed spacecraft due to the psychological effect of viewing astronautical phenomena, which cannot be replicated to the same degree on a static, two-dimensional camera. Because the use of stereopsis and headtracking provide the additional depth cues required to maximize immersion, effective replication of window-like behavior provides these benefits along with additional features (e.g., augmented reality HUDs or multiple vantage point feeds without vehicle reorientation) without the expense and hazards of physical windows.

Headtracking can be applied to 2D surfaces as well as 3D worlds. The technology can increase immersion with "home movies" from astronauts' families, which may have enormous psychological benefits on long voyages and are therefore an important human factors consideration.

The technology and methods behind headtracking and 3D stereo shall follow, as well as an easy-to-use, scalable implementation of these features.

II. Technology

A. Headtracking

1. Position Sampler

Several technologies can support real-world motion parallax. One approach includes HMD, which tracks head position accurately in a 360° panorama for a single user, though it requires bulky, expensive equipment. It is possible to provide separate, synced cameras to each eye, producing the 3D stereo effect. Due to cost and lack of portability, this technology is not favored for this study.

Depth cameras such as the Microsoft Kinect can also be used for headtracking. The Kinect projects a point cloud into the environment using a Class I infrared laser and diffraction grating. An infrared camera determines the distance to each reflected point by measuring the time delay between emission and detection. Several development libraries, such as the Microsoft Kinect SDK, OpenNI/NITE, and OpenKinect provide various joint-tracking and gesture recognition algorithms. A disadvantage for the Kinect is the large distance to its near clipping plane from the infrared sensor that limits it's use with a Desktop computer. The "Kinect for Windows" has a less restrictive range due

to a software modification through the Microsoft SDK; the functional hardware is identical to the original XBox 360 model. Linux compatibility requires the use of the OpenNI drivers which do not yet contain this augmentation. As a result, the Kinect was not used for this implementation; further research into the Kinect's use is encouraged.

A head-mounted LED array (see Figure 1), and stationary infrared camera (i.e. the Nintendo wiimote) were found to be the most flexible and easily implemented approach for headtracking. Although the wiimote requires the use of a "sensor bar," the latter is actually an LED array and relays no telemetry [3]. Each wiimote contains its own infrared camera at the forward tip of the device, and is capable of identifying four LED points simultaneously. In this setup [4] the user attaches a switchable LED keychain with a 940-950nm bulb to goggle frames (or active stereo 3D glasses). By measuring the baseline distance between these two points, the apparent distance to the camera identifies z distance, while the position of the average point determines (x,y) coordinates. Rotation about the z axis (roll) can also be calculated, however pitch and yaw cannot. In this implementation, it is generally assumed that a translation is always coupled with a rotation to point toward the center of the monitor. Given the wiimote's limited 45° FOV this is generally not a problem, however extreme rotations significantly decrease the apparent space between the LEDs, creating the illusion of increased z distance. Raw head position is then passed to a server via bluetooth for further processing.

Figure 1. Modified stereo glasses with velcro-fastened switched LED keychains with 950nm infrared bulbs. Each keychain is powered by two 3V lithium CMOS batteries.

Wiimote data manipulation is very simple and does not require the use of gesture/form recognition algorithms or calibration. Because LEDs are inherently directional, a wiimote placed at each screen in a display configuration is only active when a user looks at it. This is ideal for cockpits where space is limited. Depth cameras, while they do not require wearable equipment, may be confused by background objects or non-primary users. Furthermore, the use of multiple Kinects produce interference in regions where point clouds intersect–a significant problem in crowded environments.

Using head XYZ positions in a left-handed coordinate system (when looking toward a display, x increases to the right, y increases upwards, and z increases into the screen) with the wiimote camera as the origin. Two unique control methods can be used: head-coupled camera control, and VR window.

2. Head-Coupled Camera Control

A head-coupled camera control scheme is most advantageous for hands-free view manipulation. It provides traditional camera rotation and translation mapped to head position as if using a hand controller. The resulting effect is similar to a virtual window setup in that it allows the user to see "behind" the frame of a monitor in a virtual world, but

it maintains a symmetric view frustum at all times and tends to introduce distortion when rotating. See the camera rotation algorithm below:

$$\text{cam}_{xyz} = Vector(x, y, -z) \times translationScale$$
$$\text{eulerX} = Tan\left(\frac{x}{z}\right) \times rotationScale$$
$$\text{eulerY} = Tan\left(\frac{y}{z}\right) \times rotationScale$$
$$\text{cam}_{\theta} = Quaternion.Euler(eulerX, eulerY, 0)$$

Figure 2. Head-Coupled Camera control scheme. Scaled head rotations map to camera translations and rotations calculated trigonometrically. Euler angles are converted to quaternions before rendering.

As shown in Figure 2, camera translation is mapped to the x, y, and negative z positions of the user's head. By a left-handed coordinate system, this translates the camera right when the user moves to the right, up when the user moves up, and toward the user when the user moves backwards. This translation is scaled by an appropriate value.

Assuming the user always rotates to view the center of the screen given a translation, simple trigonometry determines the appropriate camera rotation. The angle formed by the user's LOS and the head position's z component forms a right triangle. Projecting the LOS onto the xz- and yz-planes, solve for the camera's eulerX and eulerY angles, and multiply by the appropriate scaling. The trigonometry produces the correct result due to the law of similar triangles.

See Figure 3 for a clearer view of how head movements map to virtual camera transformations in this control mode.

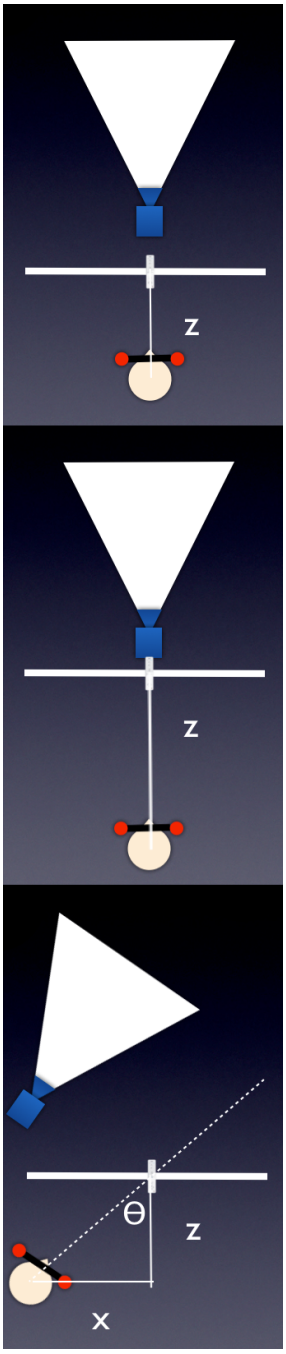

Figure 3. Camera behavior during head-coupled control mode.

3. VR Window Display

A virtual window aims to replicate the effect of perceiving a fraction of a large, distant scene through a small, nearby plane. This is different than simply changing camera rotation and position, and requires the use of a perspective projection matrix.

Perspective projection maps points in a 3D scene at various z-depths onto a 2D plane such that a farther object is of smaller apparent size than a nearer object, even if both objects have equal absolute size. This corresponds to how humans perceive the world and is a vital depth cue. In computer graphics, it is not possible to render all scene geometry onto the 2D plane of the monitor. Rather, a view frustum contains within its bounds all renderable geometry. A "frustum" refers to the portion of a solid between two parallel planes of intersection. A viewable region is modeled as a pyramid extending from the eye/camera (a single point) out to infinity (an infinite plane) with near and far planes such that geometry is only rendered if it exists within the region of the pyramid and between the two intersecting planes. The near plane is defined as the computer monitor, and the far plane as some large arbitrary distance, perhaps 1 kilometer, from the camera. The pyramid boundaries are often defined by a vertical FOV angle and a screen aspect ratio.

In most cases, view frusta are symmetric. Any two rays $\overrightarrow{l}$ and $\overrightarrow{r}$ drawn from the user's eye at the angles $\pm\theta$ in the plane of translation will cross the same distance before hitting the far plane, since the LOS is perpendicular to the viewing planes. However, a VR window display requires the creation of an asymmetric view frustum when viewing a scene off-center, such that $\overrightarrow{l}$ and $\overrightarrow{r}$ will not have equal magnitude because the LOS is no longer perpendicular to the viewing planes.

Camera rotation by itself preserves a symmetric view frustum, and is therefore analogous to picking up a window and rotating it to maintain a constant orientation with respect to a viewer. Because windows are fixed in the physical world, the perspective projection approach is more intuitive and relatively simple mathematically.

To calculate the perspective projection matrix, define the boundaries of the near plane and the distance to the near and far planes as shown in Figure 4:

$$
\begin{aligned}
l &= \frac{-\frac{1}{2} \times screenWidth - x}{z+1} \\
r &= \frac{\frac{1}{2} \times screenWidth - x}{z+1} \\
b &= \frac{-\frac{1}{2} \times screenHeight - y}{z+1} \\
t &= \frac{\frac{1}{2} \times screenHeight - y}{z+1} \\
n &= 0.5 \\
f &= 1000
\end{aligned}
$$

Figure 4. Variables used to determine the perspective projection matrix. (l,r,b,t) define the points on the near projection plane. Halve the screen width and height in meters to obtain the the initial boundaries and subtract the detected head position along the correct axis. Note that a head position directly in the center of the screen produces a symmetric frustum. Finally, divide by the distance (z) of the viewer's head from the screen, making the near plane infinitely large when the viewer's head enters the screen, with an area of zero at infinity. n and f are the distances from the virtual camera to the near and far projection planes (respectively) in meters.

As shown in Figure 4, screen width and screen height are defined in meters, since head position is also reported in meters. At $(x, y) = 0$, when measuring from the center of the screen, (l, r, b, t) correspond to points on the boundary of the monitor and produce a symmetric frustum.

Now that boundary points are available, calculate the perspective projection matrix as shown [5] in Figure 5:

$$
\begin{aligned}
x &= \frac{2 \times n}{r - l} & a &= \frac{r + l}{r - l} \\
y &= \frac{2 \times n}{t - b} & b &= \frac{t + b}{t - b} \\
e &= -1 & c &= -\frac{f + n}{f - n} \\
& & d &= -\frac{2 \times f \times n}{f - n}
\end{aligned}
$$

$$P = \begin{bmatrix} x & 0 & a & 0 \\ 0 & y & b & 0 \\ 0 & 0 & c & d \\ 0 & 0 & e & 0 \end{bmatrix}$$

Figure 5. The definition of the perspective projection matrix. This assumes a left-handed coordinate system (used by the Unity engine) in which x increases to the right, y increases upwards, and z increases into the screen when facing the monitor.

The perspective projection matrix defined in Figure 5 shapes the view frustum whose inner volume contains all geometry to be mapped onto the screen. Objects in the distance will appear smaller. An example of how the view frustum changes shape at various head positions can be found in Figure 6:

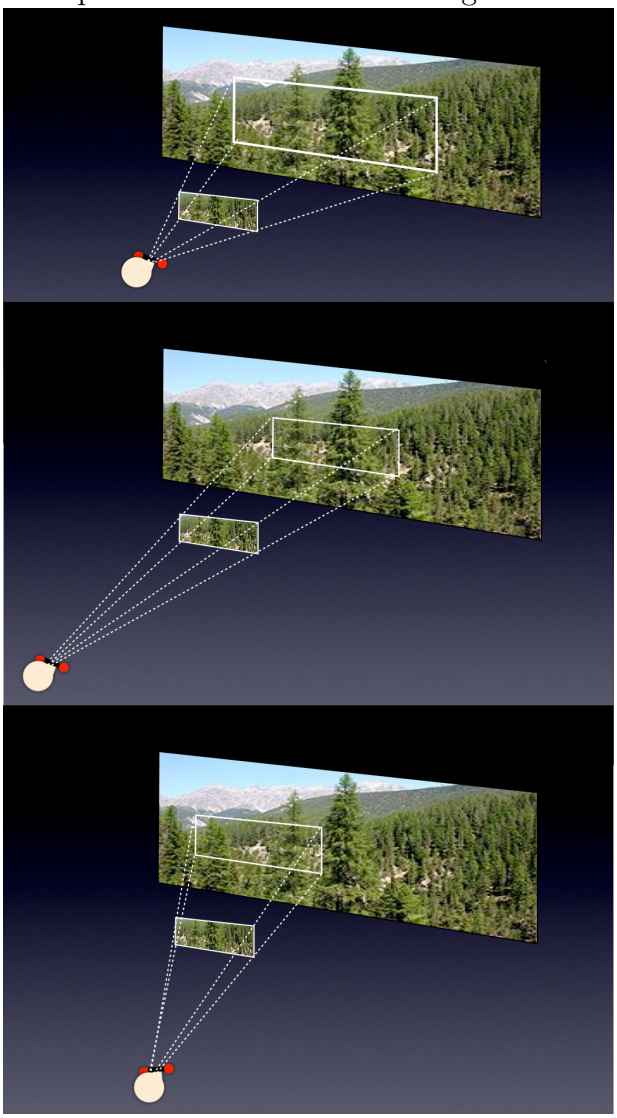

Figure 6. Resulting view frusta following perspective projection at various head positions.

B. 3D Stereo

3D stereo, or "stereopsis," provides binocular depth cues via the presentation of slightly different images to each eye. Assuming the images vary by small enough amount, the brain integrates these images together to produce depth. Unlike headtracking, which provides a sense of depth only when the view changes, binocular cues provide a sense of depth to static images. In practice, though 3D stereo is not as effective as headtracking to increase immersion, it is ideal when coupled with headtracking.

In computer graphics, frusta define views for each eye [6]. Camera translation and rotation commands are sent to a monocular camera existing logically at the midpoint between the user's eyes. On each rendering call, individual view frusta are calculated for each eye by translating the frustum origin on a plane perpendicular to the LOS by enough distance to cover each pupil.

1. NVIDIA 3D Vision

NVIDIA 3D Vision (IR emitter) and NVIDIA 3D Vision Pro (RF emitter) solutions use active shutter glasses (refer to Figure 1) to restrict even and odd frame displays to each eye. The solutions require a 120 Hz monitor, producing an effective frame rate of 60 Hz. In DirectX applications, the NVIDIA 3D Vision kit automatically interacts with the rendering pipeline to calculate and render geometry in the correct view frusta and syncing images with the shutter glasses. This solution is ideal for the Unity platform, which compiles Windows applications to render with DirectX. OpenGL applications (i.e. Linux support) requires the addition of quad buffering.

2. Quad Buffering

OpenGL optimizes rendering through the use of FRONT and BACK buffers, whereby as geometry in the FRONT buffer is passed to

the graphics card and rendered, new geometry is written to the BACK buffer and copied to FRONT when a new rendering cycle begins. This is known as "double buffering." "Quad buffering" uses FRONT and BACK buffers for each eye; the rendering function is executed twice per effective frame, once for each view frustum. This is fairly simple to implement, however shutter syncing requires a direct connection from the IR/RF sync emitter and the GPU via a 3-pin mini-DIN connector. This additional connection is not required for DirectX applications. These connections are absent on the Geforce line of GPUs and are only present on a subset of the professional-line Quadro cards, which are much more expensive and optimized for CAD displays rather than high-fidelity real-time simulations. Some work [7] has been done to support Geforce shutter sync in Linux by timing the switch byte to coincide with monitor refresh rate. This method is fairly volatile, as the refresh action and shutter sync tend to go out of phase.

One possible workaround appears to be the use of a simple circuit interceptor that shutters the 3D glasses based on a 120 Hz VSYNC pulse [8], however this method requires further study. Such a solution would be ideal due to its cost-effectiveness and the processing power required for NASA simulators.

III. Implementation

The following implementation describes the use of a decoupled client-server data delivery mechanism to the rendering application. All software libraries and applications used are available online free of charge, and most are open source.

A. VRPN

VRPN uses a client-server delivery system to abstract away the specifics of a device (such as a wiimote, Kinect or mouse) from the data it returns. Application software may work with data from various devices regardless of location, operating system, hardware libraries, and network topology. This model allows for a great level of flexibility, as the client portion need not be modified when adding a new device as long as head position can be properly calculated server side.

VRPN provides class templates to define new devices using combinations of primitive types or other devices (for example, a controller consisting of multiple buttons and analog joysticks) [9].

The implementation relies upon a previously built VRPN server for the wiimote. This server was chosen due to its cross-platform nature (tested on both Windows and Linux) and the addition of a useful GUI to control infrared camera sensitivity. The server is capable of high refresh rates depending on CPU resource utilization, reaching as high as 100Hz (almost as high as the monitor refresh rate). The server sends all wiimote status information over the network, including battery level and button states. The head tracker application only uses head position for calculations.

Data are delivered change only via function callbacks. If no state changes are detected, no network bandwidth is used.

B. UIVA

The free version of Unity does not permit loading non-native runtime DLLs. Loadable DLLs must be written in C#. Because VRPN is written in C++, the VRPNnet wrapper must be used to provide a middleware VRPN client that accepts head position data from the VRPN server and encodes and resends it on C# sockets to Unity. The implementation that follows is an extension of the UIVA con-

cept [10], expanded to support IR data feeds from the wiimote. See Figure 7 for the basic control flow:

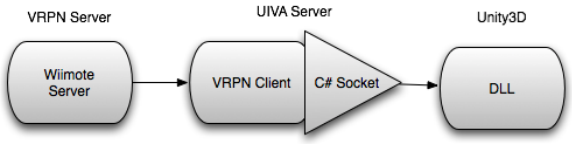

Figure 7. Data flow from the wiimote into Unity via the VRPN/UIVA bridge.

A separate VRPNnet client DLL in Unity then accepts this information and translates it into a form useable for headtracking.

C. Headtracker

The Unity head tracker is a single C# file which accepts input from the VRPNnet client DLL, performs the necessary computations outlined in Figure 2 or Figures 4 and 5 (depending on the selected control scheme), and transforms the view accordingly. Non-headtracker camera movements, such as "follow" scripts, are applied to a parent GameObject instead of the camera itself. This allows the generic head tracker configuration to apply across all Unity projects. For the full algorithm, see Figure 8:

wiiuse.dll $\leftarrow$ wiimote raw data
Calculate head (x, y, z) in VRPN server
VRPN client $\leftarrow$ (x, y, z) head position
Encode VRPN callback args as string
Send encoding to a C# socket
Unity $\leftarrow$ (x, y, z) head position encoding
if Head-Coupled Camera **then**
  Convert (x, y, z) to camera transform
else if VR Window **then**
  Convert (x, y, z) to perspective projection
end if

Figure 8. The Unity headtracker algorithm.

The control flow from datasource to display is outlined in Figure 9. Note that without Unity's C# DLL limitation, data would flow directly from the VRPN client to the rendering engine, bypassing the need for UIVA middleware.

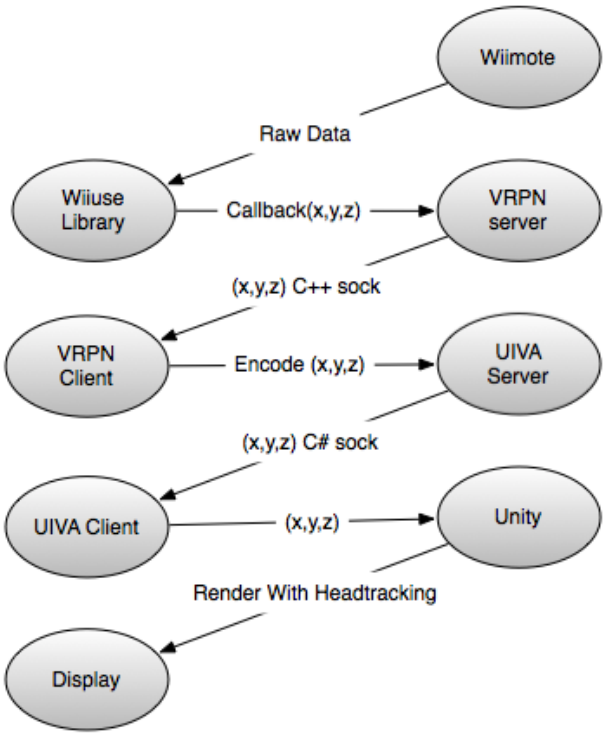

Figure 9. The Unity headtracker algorithm.

IV. Further Research

A. Kinect VRPN server

Although space considerations and point cloud interference limits the effectiveness of the Microsoft Kinect in cockpits, depth cameras may still be useful in EVA-type scenarios due to the lack of wearable hardware. In addition to providing headtracking capability, body positions and joints can be used to recreate movements in a 3D world, allowing astronauts in multiple rooms or locations to train together simultaneously. It would therefore be useful to develop a VRPN server for the Kinect.

1. *FAAST*

It should be noted that the FAAST library does currently provide support for Kinect VRPN using the OpenNI/NITE libraries [11].

However, the library does not currently support Windows 64-bit or Linux, a critical requirement for future NASA applications. Additionally, although binaries are freely downloadable, the software is closed-source. Key features to replicate include 26-joint recognition as well as basic gesture support (e.g. waves, swipes, etc.).

B. ODT/CAVE

Omnidirectional treadmills [12] and CAVE [13, 14] systems have also been used for operational training. Although studies have shown preference to headtracking systems over CAVE systems due to improved image quality and less expense, the union of omnidirectional treadmills to capture movement with head- and body-tracking will dramatically improve immersion. This system could also be utilized with NASA's ARGOS system to simulate lower-gravity environments.

The Swiss company MSE produces the Virtual Theatre, a 360° panoramic display coupled with an omnidirectional floor. This system uses an array of rollers and infrared cameras to simulate motion in an infinite space, and has been successfully used in the creation of battlefield simulators, with additional applications to human factors research and potentially EVA training. Use with the ARGOS system could provide high-fidelity training on planetary surfaces.

The Anti-Gravity Treadmill from the German company AlterG (originally designed by Dr. Robert Whalen and physician Alan Hargens in 1992 at the Ames Research Center for creating astronaut exercise routines) uses differential air pressure to reduce apparent gravity. This system could be used in conjunction with an omnidirectional treadmill to provide ARGOS-like functionality in planetary surface training.

C. EDGE Integration

EDGE, the NASA simulator graphics package, can benefit from headtracking and 3D stereo integration for crew training. Integration will require a method to sync active shutter glasses, and a VRPN client to render headtracking data. The effect needs a rendering capability of 120 Hz to prevent eyestrain; this may not be possible with all graphical options enabled due to spacecraft and planetary model complexity. Until rendering hardware improves or software optimizations are developed, a short-term alternative to EDGE may be OpenSceneGraph, an open-source Linux renderer that performs well on NVIDIA Quadro cards (and thus natively support NVIDIA 3D Vision) with support for VRPN [15]. The VRJugglua [16] framework is a useful content-authoring framework for the development of OpenSceneGraph applications supporting VRPN functionality.

V. Conclusion

This paper demonstrates an implementation of a cost-effective virtual window system using largely open source, free software components. The system uses VRPN to stream head position data from a wiimote to a client application, which encodes the data to a C# socket stream readable by the free Unity engine. These data define a perspective projection matrix used to simulate a virtual window in any Unity application. The effect is augmented with stereopsis using NVIDIA 3D Vision. The system is a prototype to demonstrate methodologies needed to implement the various facets of VR window displays. The VRPN delivery code is largely portable, while other technologies (the type of infrared camera, the rendering engine used, how to define a perspective projection matrix) can be re-implemented in the same fashion using hardware and software libraries more suitable to a spaceflight environment.

Acknowledgments

The following are thanked for their contributions:

- **Robert L. Hirsh**, who worked closely on 3D applications and procurement
- **Helen Neighbors**, who supported headtracking application development and procurement
- **Ryan A. Brown**, who worked side-by-side on headtracking and 3D stereo development
- the **JSC Co-Op Program** and the **South Dakota Space Grant Consortium** for funding the opportunity to work on this research

References

[1]Pausch, R., Proffitt, D., and Williams, G., "Quantifying immersion in virtual reality," *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1997, pp. 13–18.

[2]Ware, C., Arthur, K., and Booth, K. S., "Fish tank virtual reality," *Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*, CHI '93, ACM, New York, NY, USA, 1993, pp. 37–42.

[3]Lee, J. C., *Head Tracking for Desktop VR Displays using the WiiRemote, http://www.youtube.com/watch?v=Jd3-eiid-Uw*, December 2007.

[4]ASME, *A Modular Implementation of Wii Remote Head Tracking for Virtual Reality*, No. WINVR2010-3771. ASME, May 2010.

[5]Jens Garstka, G. P., "View-dependent 3D Projection using Depth-Image-based Head Tracking," .

[6]Bourke, P., "Calculating Stereo Pairs," July 1999.

[7]Somers, R., *NVIDIA 3D Vision using OpenGL on Linux*, Winter 2011.

[8]*VGA to 3-pin VESA Stereo Adapter, http://www.int03.co.uk/crema/hardware/stereo/*.

[9]Russell M. Taylor II, Thomas C. Hudson, A. S. H. W. J. J. A. T. H., "VRPN: A Device-Independent, Network-Transparent VR," .

[10]Wang, J., *Unity Indie VRPN Adapter, http://web.cs.wpi.edu/ gogo/hive/UIVA/*.

[11]Suma, E., Lange, B., Rizzo, A., Krum, D., and Bolas, M., "FAAST: The Flexible Action and Articulated Skeleton Toolkit," *Virtual Reality Conference (VR), 2011 IEEE*, march 2011, pp. 247 –248.

[12]*http://www.patentstorm.us/patents/5562572/fulltext.html*.

[13]Cruz-Neira, C., Sandin, D. J., and DeFanti, T. A., "Surround-screen projection-based virtual reality: the design and implementation of the CAVE," *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93, ACM, New York, NY, USA, 1993, pp. 135–142.

[14]Carolina Cruz-Neira, D. R. and Springer, J. P., "An Affordable Surround-Screen Virtual Reality Display," .

[15]*osgVRPN, http://mew.cx/osg/*.

[16]Ryan A. Pavlik, J. M. V., "VR JuggLua A Framework for VR Applications Combining Lua, OpenSceneGraph, and VR Juggler," 2011.