VIEWER PERSPECTIVE KIOSK
A SOFTWARE SOLUTION FOR A “VIRTUAL WINDOW” ADVERTISING MEDIUM

A Project

Presented to the faculty of the Department of Computer Science
California State University, Sacramento

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

by

Gregory Tyrrel Cantrell

FALL
2013
VIEWER PERSPECTIVE KIOSK

A SOFTWARE SOLUTION FOR A “VIRTUAL WINDOW” ADVERTISING MEDIUM

A Project

by

Gregory Tyrrel Cantrell

Approved by:

__________________________________, Committee Chair
Dr. V. Scott Gordon

Approved by:

__________________________________, Second Reader
Dr. Cui Zhang

______________________________
Date
Student:  Gregory Tyrrel Cantrell

I certify that this student has met the requirements for format contained in the University format manual, and that this project is suitable for shelving in the Library and credit is to be awarded for the project.

______________________, Graduate Coordinator _______________________

Dr. Nikrouz Faroughi    Date

Department of Computer Science
Abstract

of

VIEWER PERSPECTIVE KIOSK

A SOFTWARE SOLUTION FOR A “VIRTUAL WINDOW” ADVERTISING MEDIUM

by

Gregory Tyrrel Cantrell

Large flat-screen displays have become a common site in many stores, malls, and venues across the country as an advertising medium. In order to grab the attention of a passer-by these digital advertisements must rely on interesting content that engages a viewer. Starting with static rotating images, then progressing to animated clips, advertisers are continually innovating the way they present their products to catch the eye of potential customers on these platforms.

By combining a webcam, a large flat-screen display, and an attached computer a kiosk can be created that will "see" the viewer standing before it. This project demonstrates how a combination of Computer Vision and 3D Graphics can be used to create a "virtual window" effect by tracking the viewer's head location and shifting the 3D perspective in real-time. Similar demonstrations can be found that only implement simple proof of concepts using this simulated 3D illusion, however this project aims to create software for a deployable advertising kiosk found in a retail store. Many hurdles needed to be overcome to integrate multiple open source and commercial libraries into a custom application that can display engaging content with respect to
the viewer's perspective. Two effective demonstration scenes were created to show potential end
use-cases for the VPK as an advertisement medium and digital movie-poster.

_______________________, Committee Chair
Dr. V. Scott Gordon

_______________________
Date
# TABLE OF CONTENTS

List of Figures ........................................................................................................................................... viii

Chapters

1. INTRODUCTION ........................................................................................................................................ 1

2. RELATED WORK ......................................................................................................................................... 5

   2.1 Head Tracking for Desktop VR Displays using the Wii™ remote .................................................. 5

   2.2 3D Display Simulation Using Head Tracking with Microsoft Kinect™ ...................................... 6

   2.3 Magic Mirror Display Technology - UNIQLO & Sharp Electronics ............................................ 7

   2.4 4K Glasses-Free 3D Display - Phillips ............................................................................................... 7

3. TECHNOLOGY .......................................................................................................................................... 8

   3.1 Face Tracking ....................................................................................................................................... 8

   3.1 3D Graphics Game Engine ................................................................................................................ 11

4. SYSTEM ARCHITECTURE ....................................................................................................................... 16

   4.1 Face Tracking Engine (vpkTracker) ................................................................................................. 17

   4.1 3D Game Engine (vpkGame) ........................................................................................................... 19

5. RESULTS .................................................................................................................................................. 22

6. FUTURE WORK ...................................................................................................................................... 26

7. CONCLUSIONS ..................................................................................................................................... 28

Appendix. - Source Code ......................................................................................................................... 30

References .................................................................................................................................................. 56
<table>
<thead>
<tr>
<th>Figures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Figure 1.1 - The initial VPK goal illustration</td>
<td>3</td>
</tr>
<tr>
<td>2.</td>
<td>Figure 3.1 - Off-axis perspective projection matrix</td>
<td>15</td>
</tr>
<tr>
<td>3.</td>
<td>Figure 4.1 - High-level overview of the VPK software architecture</td>
<td>16</td>
</tr>
<tr>
<td>4.</td>
<td>Figure 4.2 - vpkTracker</td>
<td>17</td>
</tr>
<tr>
<td>5.</td>
<td>Figure 4.3 - Blender Game Loop VPK execution</td>
<td>20</td>
</tr>
<tr>
<td>6.</td>
<td>Figure 5.1 - &quot;StoreMart&quot; demo viewed from different angles</td>
<td>23</td>
</tr>
<tr>
<td>7.</td>
<td>Figure 5.2 - Movie poster demo viewed from different angles</td>
<td>23</td>
</tr>
<tr>
<td>8.</td>
<td>Figure 5.3 - Viewer in front of VPK</td>
<td>24</td>
</tr>
</tbody>
</table>
Chapter 1

INTRODUCTION

Large flat-screen displays have become a common sight in many stores, malls, and venues across the country. The main use for these displays in commercial locations is as an advertising or informational medium. Advertisers and business owners have widely embraced this new mode of communication. Worldwide acceptance of this new technology has created a multi-billion dollar industry for hardware and software makers in the Digital Signage market. InfoComm [1], International Sign Expo [2], and the Digital Signage Expo (DES) [3] are large professional technology expositions dedicated solely to the innovations of this form of advertising.

The technological advances in flat-screen technology have led to a decrease in the cost of production and increases in quality, size, weight, and lifetime. When these large displays were less common, their rarity was enough to draw in views from potential customers but has decreased as they have become more commonplace. Now in order to grab the attention of a passer-by these advertisements must rely on interesting content that engages a viewer. Up until recently, kiosks used for this purpose have been largely static with either video or rotating slide shows. New innovations have attempted to create dynamic displays using various methods including touch screens, computer vision, or audio interaction reaching out and pulling users into the content. This level of interaction is the dream of advertisers, as every additional moment of attention an ad can hold will translate into increased sales for their product.

In a completely different technology market that greatly depends on interactivity, current and next-gen video game systems have revolutionized the user experience by pulling the gamer in with unique control schemes. The Microsoft Kinect™, the Sony Playstation® Eye, and the
Nintendo Wii™ remote have given consoles the ability to “see” the gamer and integrate body movement into the game. Pushing the technology even further, engineers have experimented with these control systems providing interesting results. In a popular YouTube video demonstration [4], Software Engineer Johnny Lee uses the infrared video sensor of the Wii™ remote to track a viewer's relative location to the screen and adjust the rendered image to simulate a virtual window for a video game interface. The simulated 3D effect is impressive even while showing rudimentary scenes. By taking advantage of the console's computing power, high-speed data communication, and graphical performance the software demo creates an immersive and natural illusion of depth. The illusion is achieved by converting the viewer's estimated head location into 3D camera coordinates, allowing the viewpoint in the 3D world to be matched with the relative real world location of the viewer's eye. Translations in all three dimensions are mapped in real-time giving a sense of peering through a window that crosses between real and virtual worlds. This type of simulated 3D has been called "Fish-Bowl 3D", "Virtual Window 3D", "Head Coupled 3D", "Viewer Perspective 3D", as well as other names. Many projects can be found using the same paradigm to create interactive demonstrations, virtual window home installations [5], digital art pieces [6], video games [7], and more.

This project's aim is to create a software solution for a Viewer Perspective Kiosk (VPK) displaying informational or advertising content that could be set up in a store-front by integrating a large flat-screen display, webcam, and computer. Figure 1.1 shows the initial illustration describing the end goal for this project. Previous projects using this head tracking 3D approach have only scratched the surface of the potential applications. One of the most advantageous markets to use this technology is for advertising. The main goal of an advertisement is to catch the eye of an individual and keep their attention while product information is presented. When a
viewer has been pulled into the 3D effect, their attention is directed at the advertising content on the screen. In addition, viewers that are not the one “in” the effect are also drawn in as the content on the screen is skewed in an odd way. These secondary viewers will want to experience the effect as well and wait for the next opportunity. If successful, an installation with this mode of interactivity would be attractive as an advertisement and effective as an informational display.

Figure 1.1 – The initial VPK goal illustration showing an advertisement for “StoreMart”. The figure displays the difference in viewing angles with the off-axis rendered image.
For this project, software was completed that achieves the "virtual window" effect utilizing computer vision libraries wrapped and integrated into a multi-threaded face tracker. The data output from the tracker is passed to a 3D game engine that uses the face location to adjust the 3D camera. An off-axis projection is used to skew the rendered content relative to the viewer’s angle of view. Two demonstration scenes were created which consists of a 3D scene with rotating advertisement content for a fictional retail outlet named "StoreMart" and a digital movie poster.
Chapter 2

RELATED WORK

Viewer Perspective 3D is not a new technology but has been the topic of an increasing number of research projects in recent years. Each research project has implementation variations that differentiate them from each other. The VPK has been inspired by many of these other projects but differs in software technology used and target use. The following sections will provide a brief survey of some of the projects that inspired the VPK and some background about the technology used in this project.

There are a number of research project publications and video demonstrations that implement viewer perspective 3D using various hardware and software configurations. The following is a high-level comparison and contrast between the VPK approach and each other project.

2.1 Head Tracking for Desktop VR Displays using the Wii™ remote - Johnny Lee [4]

While not the first implementation of viewer perspective 3D this may be one of the most well-known. In 2007, Johnny Lee posted a video demonstration of his software that used the infrared camera embedded in the Wii™ Remote for head tracking and a custom OpenGL program. This single video has been viewed over 9 million times and has inspired many similar projects, including the VPK. In the video, Johnny gives a very good explanation and illustration of how the head tracking and 3D graphics programs work together to create the effect. A simple scene composed of a segmented corridor and floating targets is displayed on a large flat-screen television while the recording camera moves from side to side. The floating targets appear to
hover in front of the screen and the walls of the corridor appear to extend behind the screen in a very compelling illusion. The low latency camera and communication of the Wii™ remote allowed the demonstration to perform with a highly responsive and natural feel. The approach of this project differs with the VPK in both hardware and software. Head tracking is done using the Wii™ infrared light tracking system that tracks the single points of light from LEDs attached to glasses worn by the user. The VPK uses a standard webcam and computer vision libraries to identify and track faces from RGB video in an attempt to create a cheaper and more easily accessible solution.

2.2 3D Display Simulation using Head Tracking with Microsoft Kinect™ - Manfred Zabarauskas

This project uses the RGBD (Red/Green/Blue/Depth) sensor of Microsoft's Kinect™ to track a single user's head for a 3D block puzzle game. A top down view of the game-board utilizes viewer perspective 3D to create a sense of depth as each block moves into the screen. The Kinect™ sensor and software package is designed for real-time body part tracking and provides accurate high-speed x/y/z location data. Mr. Zabarauskas also uses open-source computer vision libraries to refine the head tracking even further in his implementation. In contrast, the VPK uses a standard webcam and computer vision libraries to assemble a cheaper solution. In addition, the 3D block puzzle game serves as only a proof of concept while the VPK aims to bring this technology to a new application as a kiosk.
2.3 Magic Mirror Display Technology - UNIQLO & Sharp Electronics [8]

At the 2013 Digital Signage Expo (DSE), Sharp Electronics had on display their Magic Mirror interactive virtual dressing room. Using a large flat-screen display with an attached camera and computer the mirror can overlay 3D clothing models on the video image of the user. To achieve this innovative shopping tool, the Magic Mirror uses computer vision software to identify the users pose and location that is then integrated with custom digital kiosk software for clothing selection and overlay. The computer vision libraries and video sensor used in this demonstration are unknown. The main difference between this retail kiosk and the VPK is the lack of simulated 3D using head tracking.

2.4 4K Glasses-Free 3D Display – Phillips [9]

For 3D digital signage one of the most cutting edge pieces of technology is Phillip's 4K lenticular 3D display. With ultra-high resolution 4x that of normal HD televisions and glasses-less 3D, these displays are eye catching, immersive, and expensive. The cost of each display is prohibitive for normal retail installations. In comparison to the VPK, this is purely a hardware solution for a 3D kiosk while the VPK is a software solution used in combination with relatively low cost hardware. The type of 3D effect is also different. Lenticular displays achieve a 3D effect by showing different images to the left and right eye simulating depth by an effect called "stereopsis" [7]. The VPK simulates depth by the movement of objects in the scene called "motion parallax" [7] and a shifting off-axis perspective. These terms will be discussed in more detail under a later section.
The software components needed to achieve the 3D virtual window effect can be simplified into two pieces - a face tracking engine and a 3d Graphics engine.

3.1 Face Tracking

The first and most challenging piece needed in a system that uses the viewer perspective approach is a reliable, accurate, and fast way of finding the viewer's face within the vision of the webcam. Furthermore, the system must be able to track that face as it moves through the field of view. Many complex algorithms have been developed for facial detection, tracking, and recognition, however most are not fast enough for a real-time system. Two related computer vision libraries were explored for this face tracking solution.

*OpenCV (Open Computer Vision)*

Originally published by Intel in 1999, OpenCV is a collection of computer vision libraries that has been released under the open source BSD license [10]. With the purpose of encouraging innovation and the application of computer vision in software, it was designed to be used as a basic vision infrastructure for applications to be built on top of. It is now maintained by Itseez [11] and has been used by many large companies for computer vision research, vision software infrastructure, and commercial vision-based software products [10]. The toolset is made up of Machine Learning, Artificial Intelligence, and statistical algorithms as well as utility libraries for
image or video management. The open source nature of OpenCV and its vast online support made it an ideal candidate for the VPK.

OpenCV has many object recognition and tracking algorithms that can be used to find and follow faces. The first step in the Face Tracking process is face detection which can be accomplished using the built in Haar Feature-based Cascading Classifiers [12].

**Haar Feature-based Cascading Classifiers**

One of the most widely used object detection algorithms in computer vision is the Viola-Jones object detection framework [13]. This framework utilized simple dark and light rectangular key images, called "Haar-like features", arranged in pyramid-ing scales to analyze regions of an image for the target object. These pyramids of key images are combined into decision-tree Classifiers that are pre-trained on input image sets of the target object and saved for later use. At each scale, the input image pixel information within a given rectangle is generalized to dark and light values then compared to the key images. If the generalized pixel information matches closely enough at that scale, the algorithm moves to the next lower stage of classifiers until a satisfactory result is found. OpenCV's implementation has some improvements on this algorithm including the use of additional "Haar-like feature" key images and "boosting" of classifiers by having complex sets of classifiers at each stage [12]. For the VPK OpenCV's included Face Detection cascading classifiers were used for the first iteration of the Face Tracker.

Once the face has been detected, tracking of that individual’s face is the second step needed to reduce jumping between multiple faces and for smoother movement. One of the best algorithms included in OpenCV for object tracking is the CamShift (Continuously Adaptive Meanshift) algorithm.
CamShift

Camshift [14] is an extension of the Meanshift [14] algorithm that is applied to video input or a sequence of images. Meanshift can be explained simply as shifting a target area of an image in a direction to maximize the number of pixels of a target value that are contained in the given target window size. Given an initial window size and position the centroid of pixels of a target value is calculated. The window is then shifted to be centered on the calculated pixel density centroid. This process is repeated until the maximum density of pixels is contained within the window. By filtering an image to only contain pixels of flesh-tone color, a pixel value image can be analyzed by the Meanshift algorithm to find the largest pixel density of flesh-tone that would fit within a given window size.

To accommodate video that would contain a moving target, the Camshift algorithm continuously adapts the result using Meanshift and attempts to fit a rotation for the target window to the maximum pixel value density. OpenCV’s implementation first uses Meanshift to find an initial guess, then increases the size of the target window and attempts to fit an oval shaped window with rotation to find a face within the scene. It continuously loops through this process for video input in order to track movement from one frame to the next. The previous frames window location is used as the starting point as the target face could not have moved far between frames.

FaceAPI

While OpenCV is an extensive collection of computer vision libraries, FaceAPI [15] is a subset of the computer vision libraries built upon OpenCV focused on Face Detection and Tracking. Seeing Machines© [16] developed FaceAPI and has released it to be used with no
charge in non-commercial use. They have combined OpenCV face tracking features together and added other algorithms specifically designed to track faces and facial features. FaceAPI's singular focus on face tracking, and its pre-packaged combination of many useful Computer Vision algorithms made it an ideal solution for the VPK.

The key features needed for this application are the same features as described in the previous section including Haar Feature-based Cascading Classifiers, and Camshift. In addition to these key features, the library uses many other computer vision and image processing algorithms to refine and simplify the input for analysis.

3.2 3D Graphics Game Engine

To create an engaging virtual world on the other side of the VPK screen a 3D graphics engine is needed to render the world in real-time. Gaming graphics have consistently pushed the boundaries of graphics computation optimization, and visual effects to create realistic 3D worlds with minimal latency. The VPK can leverage these features in open source game engines to create impressive visuals that can be rendered in relation to the viewer's perspective.

*OGRE3D and OpenSceneGraph*

When choosing an engine for the VPK the first option was one of the most widely used open source game engines, OGRE3D (Object-Oriented Graphics Rendering Engine)\textsuperscript{17}. Since 1999 the OGRE3D team has improved and maintained this C++ based rendering library. The engine supports both OpenGL and DirectX and can be used on multiple platforms. For rendering optimization another open source library, OpenSceneGraph [18], can be used in conjunction with the OGRE rendering engine to create a fast and rich game world.
By using these libraries the VPK can take advantage of the decades of optimization and innovation the gaming industry has developed to create such impressive graphics. Some of the key features the VPK can take advantage of are:

- **Scenegraph Integration** - A scenegraph is a construct used in game design which manages all objects in a game world with built in functions for quick traversal, minimizes render calls, and supports many optimizations used in current games.

- **Culling** - Culling is the process of excluding objects from the list of things to render and by intelligently managing this list the performance of a rendering engine can be greatly increased.

- **Level of Detail** - LOD is a feature that simplifies objects that are distant from the scene camera and therefore not needing the full level of detail. These simplified game assets can be rendered much faster.

- **Lighting/Shadows** - Adding to the realism of a rendered scene, real-time lighting and shadow rendering are much needed features of OGRE.

- **Shaders** - Stunning effects such as water, fire, normal maps, smooth shading, and many more are all enabled by using GPU shaders.

- **Texturing** - Texturing is one of the most basic features of a rendering engine that adds to the realistic appearance of objects within the scene. Easy management and mapping of textures to objects greatly reduces the complexity of implementing a VPK solution.

- **Model Importing** - Importing of complex 3D objects made in modeling software such as Blender 3D, Maya, and SketchUp allows for quick content creation and integration into the VPK virtual world.
Another alternative game engine explored was the open source 3D modeling and game engine Blender [19]. Initially developed in 1995 as a commercial product, in 2002 Blender was released under the GNU General Public License and is free to use [20]. With a large community of active 3D modelers, programmers, and game developers who contribute to the project Blender has become a very mature 3D modeling tool and game engine on par with expensive professional 3D software packages. Many advanced features have been implemented allowing high quality computer graphics to be created for use by amateurs to professionals. The large user base and developer community has produced very good tutorials and documentation making learning the software easy and quick. An integrated scripting engine using Python allows custom features to be added to the application as well as game mechanics. With the addition of a dedicated game engine, a whole new community of game design hobbyist became avid users and in 2008 release the first blender community 3D game, *Yo Frankie* [20].

Because of the extensive content creation tools and native support for advanced game graphics, Blender was an ideal solution for quickly creating the VPK prototype completed in this project. The key feature useful to the VPK are the same as listed under the OGRE3D engine with the addition of these:

- **Game Engine**
  - Animation in the game engine is made easy by the tight cohesion of the Blender 3D modeling/animation tools and the game engine.
  - Physics simulation libraries are integrated into the game engine and make it easy for adding content into the VPK virtual world with physics based
properties. The realism of the physics simulation helps create a compelling scene.

- **Content Creation** - Blender's first purpose is a to be an easy to use and feature rich 3D content creation tool. Having both the Game Engine and Content Creation features removes the need for other expensive modeling software.

- **Python Scripting** - The integrated Python Scripting support allows game creators even more control over game assets and mechanics. For the VPK it also allows use of Python's dll importing feature.

**Off-axis Perspective**

In order to achieve a correct image based on the viewer's perspective the 3D scene must be rendered with a non-standard projection matrix. Figure 2.1 illustrates the difference between a standard perspective projection and an off-axis perspective projection. The standard projection matrix used to calculate and draw objects with perspective in a virtual world is based on a viewpoint directly perpendicular to the viewing plane. The image rendered to the display looks correct as far as perspective making objects in the distance smaller or larger. However, as the viewer's head moves from an angle that is not along the perpendicular of the viewing plane the image looks flat, a 2D mapping of a 3D world. As mentioned previously the VPK uses an effect of human vision known as parallax in conjunction with a shifting off-axis perspective to create an illusion of depth. Parallax is described as the "difference in the apparent position of an object viewed along two different lines of sight" [7]. In the VPK, depth is implied by adjusting the image to match the expected parallax of the camera and world based on the Viewer's head location.
In a standard computer graphics rendered scene with a static camera, there would be no parallax as there is only one line of sight. Because the VPK's camera is actively moving to match the movement of the viewer the parallax is implied by the camera motion. The shifting off-axis perspective of the VPK causes the 2D mapping of the 3D world to be skewed in such a way as to match what the Viewer's eye would expect to see if they were looking through a window to objects on the other side despite the non-perpendicular location of their line of sight.

Figure 3.1 – An explanation about how the off-axis perspective projection matrix is calculated. [21]
Chapter 4
SYSTEM ARCHITECTURE

As with most software projects the architecture of the system changed over the life cycle of the project. Adjustments were made to keep progress moving forward. This section will give an overview of the final architecture and describe some of the initial components that changed and explain why. The VPK implementation can be broken into two main components, the face tracking engine (vpkTracker) and the 3D game engine (vpkGame). Figure 4.1 shows a high level overview of the system.

Figure 4.1- High-level overview of the VPK software architecture. The two main modules are the FaceTracker and the Game Engine.
4.1 Face Tracking Engine (vpkTracker)

At the core of the Face tracking engine are the FaceAPI libraries which handle the actual detection and tracking. A wrapper class was implemented to handle interfacing with the game engine, thread management, and location encoding. The vpkTracker C++ class was implemented by segmenting these 3 aspects into an interface layer, the main class, and utility functions. In the end a dll is compiled to be used in the 3D game engine.

Figure 4.2 – The vpkTracker wraps the face detection and tracking algorithms of OpenCV and FaceAPI.
The Interface layer defines the library calls available for the Face Tracking engine. The interface is simple with an init() function which takes a shared memory location, and a shutdown() function that will tell the main class to stop the tracking thread, cleanup, and shutdown. Future plans for additional bi-directional communication were made to allow for updating of face tracker parameters.

The main class (Tracker) was initially designed to use OpenCV directly calling on Haar Feature-based Cascading Classifiers to detect faces. The first version of the main class accomplished this but the results were too inconsistent causing visual problems. Quickly the project shifted and the FaceAPI was integrated for detection and tracking. Leveraging this library allowed for a simpler implementation and improved results. Vision algorithms such as Haar Feature-based Cascading Classifiers, Camshift, Background subtraction, etc. were already combined into a simple API and optimized. The Tracker class handles initialization and shutdown of the smAPI (FaceAPI), webcam, and worker threads. A callback is registered with the smEngine that will receive all head pose data. The callback encodes and saves the X,Y,Z location returned by the FaceAPI into a single bitstring. This data is then copied into the shared memory location with appropriate shared memory protection. In the final deployment, the Game Engine will allocate and pass a pointer into the vpkTracker where the face location will be stored. The Game Engine will be a read only user of the shared memory while the vpkTracker will be a write only user. To avoid any concurrent access issues such as a miss-read, OS Atomic write functions were used when storing data in the shared memory. On shutdown the Tracker will close and deallocate the smEngine, smCamera, and any other resources allocated.

The utility function encodeLocation() is important as it defines the algorithm used to map x,y,z locations into a single 32 bit number. Since the VPK is multi-threaded and using shared
memory, the coordinates are saved into a variable with an Atomic write. The bitstring is broken into 3 pieces to store the x,y,z values with 10 bits each. To map the float value $Z_t$ into the bitstring a ratio is set up by storing the value $= Z_t \times 2^{10}$ into bits 9-0. By mapping the calculated face location values in this way, some accuracy is lost but is acceptable in this situation. A similar decoding inverse function in the Game Engine converts the values back to floating point.

4.2 3D Game Engine (vpkGame)

As described in the Background section, the Blender Game Engine is a component of the larger software application Blender 3D. Custom functionality and advanced game interaction can be implemented using Blender's integrated Python scripting engine. A vpkEngine which starts the Face Tracker and manages the Camera adjustments was implemented using python scripts. To present an interesting demo the Game World was populated with objects and images to create a few fake advertisements and an Options gui were added as well.
The *vpkEngine* within Blender game is initialized during the startup of the game. In initialization the engine defines constants and global variable, loads the vpkTracker dll, then creates the shared memory pointer that will be passed. Lastly it calls the initialization function of the dll and passes along the pointer. After the initialization phase the Face Tracker thread will be running, searching for an initial face and storing the results in the shared variable while the Game Engine begins its game loop. During the game loop the vpkEngine sets the camera location based on the currently stored face location and adjusts the camera perspective matrix to the calculated

---

**Figure 4.3 – The Blender game engine loop with a sequence of events during the VPK execution.**
off-axis perspective matrix. The Game World is then rendered normally with the new camera parameters.

An example advertisement was made that resembled the initial project goal figure showing a "StoreMart" ad (Figure 5.1). To accomplish this the world was populated with many objects and animations. These had to be created and animated using the Blender 3D modeling tools. A rotating animation with the store "welcome" message and multiple product advertisements was added to illustrate the intended commercial use. While not fully functional, an Options Gui menu was created and scripted to allow adjustment of rendering parameters.
Together the vpkTracker and vpkGame succeed in creating the illusion of a virtual window with depth that can be used for an advertising or informational medium. One initial advertisement was created for a fictitious retail store "StoreMart" that was successfully demonstrated. A second movie poster scene was created inspired by content displayed on a digital kiosk for RedBox® movie rentals. A movie poster was split into multiple depth planes and setup in the VPK game world. The effect was natural and impressive. Figures 5.1 – 5.3 show photos of the results with the two VPK demos.
Figure 5.1 – StoreMart Results. Photos taken of the VPK with the StoreMart Demo at various angles to match the original goal illustration. See Figure 1.1. The bottom photos were taken from a secondary viewpoint.

Figure 5.2 - Movie Poster Results. Photos taken of the VPK with the Movie Poster Demo. The Great Gatsby [22] movie poster property of Warner Bros ®.
Performance:

The VPK renders the demo ads, fullscreen, at 60 frames per second consistently with or without face tracking enabled, full. The computer that was running the VPK has the following specifications:

- CPU: Intel® Core™ i7-2760QM CPU @ 2.4GHz
- RAM: 8.0 GB
- OS: Windows 7 64bit
- Graphics Card: NVIDIA Quadro 1000M
- Display: 24.1" Asus Widescreen 1920x1200

The FaceAPI performance metrics are as follows [15]:

- Fully automatic commencement of tracking, with automatic reacquisition if tracking is lost;
- Works with any webcam;
- Works with PointGrey high-speed firewire cameras;
- Advanced face-detection function, with 3 levels of feature refinement;
- Can track up to +/- 90 degrees of head rotation;
- Can track with as few as 50 pixels across the face;
- Large tracking volume (as a guide, a 640×480 camera provides tracking volume of approximately 100 x 60 x 100 cm);
- Robust to occlusions, fast movements, large head rotations, lighting, facial deformation, skin color, beards, glasses etc;
- Full control over all tracking parameters;
- Works in visible and IR lighting;
- Face texture information is provided, along with a number of supporting features for simple integration into your software development, including comprehensive documentation.
Many improvements are planned in order to make the VPK a more robust and viable solution for in-store kiosks. Easy Content Creation and Improvements in performance are two of the first areas that need further development.

**Easy Content Creation**

In able to deploy a full VPK solution in a public venue an easy way for a non-programmer to edit the content is vital. At present, content must be made using the Blender 3D modeling tools. This is not an ideal solution as Blender, while powerful, has a steep learning curve. A content creation tool that would allow for easy importing of images in rotating advertisements, selection between a few template background scenes, and editing of text would be an ideal solution. For the Movie Poster style kiosk a tool that would give the ability to import image layers that would be stacked at configurable Z depths would allow easy content creation and updating.

**Performance Improvement**

During the implementation phase of this project the level of optimization using the FaceAPI was limited to mostly the default configuration. Further investigation into the configurable options of the FaceAPI engine could increase the accuracy of the VPK. The engine configuration could be optimized for this specific application.
The shared memory scheme used to pass the face location is not the typical mode of Inter-Process Communication (IPC) or inter class communication. A more traditional approach would be to use sockets and UDP communication to share data between processes. Another more straightforward strategy would be to add thread-safe accessors and setters to the vpkTracker interface. A comparison of these approaches may reveal advantages to one of them over the others. The Microsoft Kinect™ or other RGBD sensor could replace the FaceAPI. These sensors are designed for body part tracking by combining hardware, low level drivers, and software libraries for this purpose. The accuracy and speed of these sensors would provide a great performance improvement for estimating the viewer's head location.
Chapter 7
CONCLUSIONS

Over the course of this project there were many successes and failures. With a focus on the final real-world application, development during this project needed to shift numerous times between available libraries and algorithms. I first tried using OpenCV for head tracking and OGRE3D for the game engine but adjusted after having difficulty with integration and performance. FaceAPI and the Blender Game Engine emerged as the best options for this project and worked very well in the final demo. A Face Tracking dll was created by wrapping the FaceAPI with an multi-threaded engine. This engine was integrated into a custom Blender Game Engine that adjusted the camera perspective and position based on the Face Tracker’s output. The final demo achieved the desired effect with great success. The simulated 3D was very natural and gave the feeling of looking through a virtual window. As an informational or advertising kiosk it would be both eye-catching an engaging.

The goal of this project was to create a new application of an existing technology. Viewer perspective 3D is not necessarily a new technology as it has been used in various research projects, however there are not many applications for the technology beyond research. Dropping technology costs have enabled more hobbyists to experiment with this type of 3D content. With the VPK framework implemented through this project, further development will continue toward creating a deployable commercial system. An aggressive push toward innovation in digital advertising could give the VPK a niche in the multi-billion dollar Digital Signage market. Increased exposure through marketable applications such as the VPK will open the door for even more innovative uses of Viewer Perspective 3D. Perhaps in a short time digital advertisements in
some of the largest stores will grab the attention of anyone nearby to engage and interact with the illusion of a virtual window.
APPENDIX

*FaceTracker dll Source Code:*

File: Tracker.h

```c
#ifndef TRACKER_H
#define TRACKER_H

#include <windows.h>
#include <tchar.h>
#include <strsafe.h>
#include <stdio.h>
#include <random>
#include "LocationEncode.h"
#include "sm_api.h"

class Tracker { 
private:
    //member variables
    SHARED_VAR* mSharedVar;
    HANDLE mThreadHandle;
    DWORD mThreadId;
    //if we are using the FaceAPI
    smEngineHandle engine_handle;
    //functions
    smCameraHandle createFirstCamera();

class Tracker { 
public:
    Tracker (SHARED_VAR* sharedVar);
    ~Tracker ();
    int init();
    static void STDCALL receiveHeadPose(void *,smEngineHeadPoseData head_pose,
                    smCameraVideoFrame video_frame);
    void saveHeadLocation(smEngineHeadPoseData head_pose);
};

#endif //TRACKER_H
```
// Face API - Some utilities for error handling, printing and console key-press interpreting etc.
#include "utils.h"
#include "lock.h"
#include "mutex.h"
#include "sm_api.h"

#define USE_FACE_API

#ifndef USE_FACE_API
  bool runthread = true;
  // Worker method for thread
  DWORD WINAPI runMethod( LPVOID lpParam ) {
    SHARED_VAR* sharedVar = (SHARED_VAR*) lpParam;
    Detector* myDetector = new Detector();
    myDetector->init();

    clock_t t;
    int avgTicks = 0;
    while(runthread) {
      t = clock();
      // try the camshift first
      if(!myDetector->detectWithCascade(sharedVar)) {
        printf("dll - no face found with Cascade\n");
      }
      t = clock() - t;
      avgTicks = (avgTicks + t)/2;
    }
    float avgTime = ((float)avgTicks)/CLOCKS_PER_SEC;
    printf("dll - thread exiting avgTime=%f secs\n", avgTime);
    delete myDetector;
  }
#else // #ifdef USE_FACE_API
  void * gSelfPt;
#endif // #ifdef USE_FACE_API
return 0;
}
#endif //ifdef USE_FACE_API

Tracker::Tracker(SHARED_VAR* sharedVar) {
    mSharedVar = sharedVar;
}

int Tracker::init() {
#ifdef USE_FACE_API
    try {
        // Initialize the API
        THROW_ON_ERROR(smAPIInit());

        // Register the WDM category of cameras
        THROW_ON_ERROR(smCameraRegisterType(SM_API_CAMERA_TYPE_WDM));

        engine_handle = 0;

        // Create a new Head-Tracker engine that uses the camera
        THROW_ON_ERROR(smEngineCreate(SM_API_ENGINE_LATEST_HEAD_TRACKER,&engine_handle));

        // Print out a list of connected cameras, and choose the first camera on the system
        //camera_handle = createFirstCamera();

        // Create a new Head-Tracker engine that uses the camera
        //THROW_ON_ERROR(smEngineCreateWithCamera(SM_API_ENGINE_LATEST_HEAD_TRACKER,camera_handle,&engine_handle));

        // save off a pointer to self for non-static callback
        gSelfPt = (void*)this;

        // Register Headpose Callback
        THROW_ON_ERROR(smHTRegisterHeadPoseCallback(engine_handle,0,&Tracker::receiveHeadPose));

        // Start tracking
        THROW_ON_ERROR(smEngineStart(engine_handle));
    }
    catch (exception &e)
{  
    smAPIQuit();  
    return -1;  
}  
return 0;

#else  
    mThreadHandle = CreateThread(NULL, 0, runMethod, (LPVOID)mSharedVar, 0, &mThreadId);  
    if (mThreadHandle == NULL)  
    {  
        printf("dll - Error creating thread. Exiting.");  
        return -1;  
    }  
#endif  
return 0;  

Tracker::~Tracker() {  
    printf("dll shutdown\n");  

#ifdef USE_FACE_API  
    // Destroy engine  
    THROW_ON_ERROR(smEngineDestroy(&engine_handle));  
    smAPIQuit();  
#else  
    // tear down thread  
    runthread = false;  
    // Close all thread handles and free memory allocations.  
    CloseHandle(mThreadHandle);  
#endif  
    mSharedVar = 0;  
}

// Create the first available camera detected on the system, and return its handle  
smCameraHandle Tracker::createFirstCamera()  
{  
    // Detect cameras  
    smCameraInfoList info_list;  
    THROW_ON_ERROR(smCameraCreateInfoList(&info_list));  
    if (info_list.num_cameras == 0)
throw runtime_error("No cameras were detected");
}
else
{
/* printf("The followings cameras were detected: ");
for (int i=0; i<info_list.num_cameras; ++i)
{
  char buf[1024];
  cout << "   " << i << " . Type: " << info_list.info[i].type;
  THROW_ON_ERROR(smStringWriteBuffer(info_list.info[i].model,buf,1024));
  cout << " Model: " << string(buf);
  cout << " Instance: " << info_list.info[i].instance_index << endl;
  // Print all the possible formats for the camera
  for (int j=0; j<info_list.info[i].num_formats; j++)
  {
    smCameraVideoFormat video_format = info_list.info[i].formats[j];
    cout << "    - Format: ";
    cout << " res (" << video_format.res.w << "," << video_format.res.h << ")";
    cout << " image code " << video_format.format;
    cout << " framerate " << video_format.framerate << "(hz)";
    cout << " upside-down? " << (video_format.is_upside_down ? "y":"n") << endl;
  }
}*/
}

// Create the first camera detected on the system
smCameraHandle camera_handle = 0;
THROW_ON_ERROR(smCameraCreate(&info_list.info[0], // Use first camera
  0, // Use default settings for lens
  &camera_handle));

// Destroy the info list
smCameraDestroyInfoList(&info_list);

return camera_handle;
}

void Tracker::saveHeadLocation(smEngineHeadPoseData head_pose)
{
  // Need to transform the coordinates from -1.0 - 1.0 to 0.0 - 1.0
  encodeLocation(((head_pose.head_pos.x+1.0f)/2.0f),((head_pose.head_pos.y+1.0f)/2.0f),
  head_pose.head_pos.z, mSharedVar);
  //printf("dll - saveHeadLocation : (%f, %f, %f)n",
  head_pose.head_pos.x,head_pose.head_pos.y,head_pose.head_pos.z);
}
// Callback function for head-poses
void STDCALL Tracker::receiveHeadPose(void *, smEngineHeadPoseData head_pose, 
smCameraVideoFrame video_frame)
{
    sm::faceapi::samplecode::Lock lock(sm::faceapi::samplecode::g_mutex);
    Tracker* self = (Tracker*) gSelfPt;
    self->saveHeadLocation(head_pose);
}
File: Interface.cpp

#include <stdio.h>
#include <string>
#include "tracker.h"

#define USE_FACE_API

Tracker * myTracker;

extern "C"
{
    extern __declspec(dllexport) void _shutdown();
    extern __declspec(dllexport) void _init(SHARED_VAR * sharedVar) {
        printf("dll - Initialization of shared objects sharedVar(%p)=%d\n", sharedVar,
        *sharedVar);
        myTracker = new Tracker(sharedVar);
        if(myTracker->init() < 0)
            _shutdown();
    }
    extern __declspec(dllexport) void _shutdown() {
        printf("dll - shutting down\n");
        delete myTracker;
    }
}
File: Detector.h

#ifndef DETECTOR_H
#define DETECTOR_H

#include "opencv2/objdetect/objdetect.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include "tracker.h"
#include "Parameters.h"
#include "LocationEncode.h"

using namespace std;
using namespace cv;

class Detector
{
  public:
    Detector();
    ~Detector();
    void setParameters(Parameters* params);
    int init();
    bool detectWithCascade(SHARED_VAR* sharedVar);
  private:
    CascadeClassifier face_cascade;
    VideoCapture capture;
    Mat frame;
    Mat frame_gray;
    std::vector<Rect> faces; // TODO: shrink the size to 1
    void encodeLoc(cv::Point* loc, SHARED_VAR* sharedVar);
    Parameters* mParameters;
};

#endif //DETECTOR_H
File: Detector.cpp

#include "Detector.h"
#include "LocationEncode.h"

#include <iostream>
#include <stdio.h>

/** Global variables */
String face_cascade_name = "C:\Users\gcantrell\Desktop\VPK2\OpenCV\opencv\data\haarcascades\haarcascade_frontalface_alt.xml";

Detector::Detector() {

}

Detector::~Detector() {
    //cvReleaseHaarClassifierCascade( &(this->face_cascade));
    capture.release();
    printf("dll - should have released capture\n");
    delete mParameters;
}

void Detector::setParameters(Parameters* params) {
    mParameters = params;
    init();
}

int Detector::init() {
    // 1. Load the cascades
    if (!face_cascade.load( face_cascade_name)) {
        printf("--(!)Error loading face cascade '%s'\n", face_cascade_name);
        return -1;
    }

    //open the video capture device
    if (!capture.isOpened())
        capture.open(-1);
    if (!capture.isOpened()) {
        printf("--(!)Error opening video capture\n");
        return -1;
    }
}
bool Detector::detectWithCascade(SHARED_VAR* sharedVar) {
    if(capture.isOpened() && capture.read(frame)) {
        if( frame.empty()) {
            printf("unable to capture frame.");
            return false;
        }

        cvtColor( frame, frame_gray, COLOR_BGR2GRAY);
        equalizeHist( frame_gray, frame_gray);

        //-- Detect single largest face
        face_cascade.detectMultiScale( frame_gray, faces, 1.1, 2,
0|CASCADE_FIND_BIGGEST_OBJECT, Size(30, 30) );

        //TODO: represent Z axis somehow, Headsize(must be smooth) or Eye width ?
        Maybe Camshift would be better for estimating Z
        Rect face = faces[0];

        Point loc(faces[0].x + faces[0].width/2, faces[0].y + faces[0].height/2);

        //encode the location into the shared variable
        encodeLoc(&loc, sharedVar);
        return true;
    } else {
        printf("unable to access capture device\n");
        return false;
    }
}

void Detector::encodeLoc(Point* loc, SHARED_VAR* sharedVar) {

    //This float math could be slow, maybe think of a new way
    //at least save the capture size into parameters so we don't call a function all the time
    float xPercent = (float)loc->x/(float)capture.get(CV_CAP_PROP_FRAME_WIDTH);
    float yPercent = (float)loc->y/(float)capture.get(CV_CAP_PROP_FRAME_HEIGHT);

    encodeLocation(xPercent, yPercent, 0.5f, sharedVar);
}
File: Parameters.h

#pragma once
#ifndef PARAMETERS_H
#define PARAMETERS_H

#include <string.h>

class Parameters {
public:
    Parameters();
    ~Parameters();
    void setFaceCascadePath(std::string filePath) {this->mFaceCascadePath = filePath;}
    std::string getFaceCascadePath() const { return mFaceCascadePath;}

private:
    std::string mFaceCascadePath;
};

#endif //PARAMETERS_H
#ifndef LOCATIONENCODE_H
#define LOCATIONENCODE_H

#include <stdint.h>
#include <stdio.h>

typedef volatile uint32_t SHARED_VAR;
#define BITS_FOR_X 10
#define BITS_FOR_Y 10
#define BITS_FOR_Z 10

#define MAX_X ((1 << BITS_FOR_X)-1)
#define MAX_Y ((1 << BITS_FOR_Y)-1)
#define MAX_Z ((1 << BITS_FOR_Z)-1)

//Encode the location into a single variable
void encodeLocation(float xLocPercent, float yLocPercent, float zLocPercent, SHARED_VAR * sharedVar);

__inline SHARED_VAR Round(float a);

#endif //LOCATIONENCODE_H
// Encode the location into a single variable
// TODO: this float math could be slow, think of a faster way. It could be a minor slowdown compared
// to finding the face.

float zLocMax = 10.0f;

void encodeLocation(float xLocPercent, float yLocPercent, float zLocValue, SHARED_VAR* sharedVar) {
    float x = (float)(MAX_X) * xLocPercent;
    float y = (float)(MAX_Y) * yLocPercent;
    // TODO replace the zLocMax with a configurable value
    float z = (float)(MAX_Z) * ((zLocValue > zLocMax)? zLocMax: (zLocValue / zLocMax));

    SHARED_VAR temp = 0;
    temp = Round(x);
    temp = temp << BITS_FOR_Y;
    temp += Round(y);
    temp = temp << BITS_FOR_Z;
    temp += Round(z);

    printf("dll - LocationEncode : Loc(%f, %f, %f) = (%f, %f, %f) = %#x\n", xLocPercent, yLocPercent, zLocValue, x, y, z, temp);

    // Atomic write to the shared variable
    InterlockedExchange(sharedVar, temp);
}

// Taken from http://stereopsis.com/FPU.html
// Fast float to int conversion
__inline SHARED_VAR Round(float a) {
    SHARED_VAR retval;

    __asm fld a
    __asm fistp retval

    return retval;
}
Blender 3D python scripts

File: VPKengine.py

from ctypes import *
import sys
import GameLogic
import bge
import bpy
import mathutils
import bgl

BITS_FOR_X = 10
BITS_FOR_Y = 10
BITS_FOR_Z = 10

MAX_X = ((1 << BITS_FOR_X)-1)
MAX_Y = ((1 << BITS_FOR_Y)-1)
MAX_Z = ((1 << BITS_FOR_Z)-1)

BITMASK_FOR_X = MAX_X << (BITS_FOR_Y + BITS_FOR_Z)
BITMASK_FOR_Y = MAX_Y << BITS_FOR_Z
BITMASK_FOR_Z = MAX_Z

sharedVar = c_int(0)
dll = cdll.LoadLibrary("C:\\Users\\gcantrell\\Desktop\\VPK2\\BlenderGame\\vpkTracker")

scene = bge.logic.getCurrentScene()
cam = scene.active_camera

xPos = 0.0
yPos = 0.0
zPos = 0.0

ORIGIN_X = 0.0
ORIGIN_Y = 0.0
ORIGIN_Z = 1.0

RANGE_X = 0.5
RANGE_Y = 0.5

HALF_RANGE_X = RANGE_X/2
HALF_RANGE_Y = RANGE_Y/2
#1 real meter is 0.915 faceAPI units
Z_SCALER = 0.915
loc1 = [0.0,0.0,0.0,0.0]
loc2 = [0.0,0.0,0.0,0.0]
firstLoc = False

def init():
    #testCalcCameraMatrix()
    dll._init(byref(sharedVar))

def shutdown():
    dll._shutdown()
    GameLogic.endGame()

def setCameraLocation():
    val = sharedVar.value
    x = (BITMASK_FOR_X & val) >> (BITS_FOR_Y + BITS_FOR_Z)
    y = (BITMASK_FOR_Y & val) >> (BITS_FOR_Z)
    z = BITMASK_FOR_Z & val

    xPerc = 0.0
    yPerc = 0.0
    zPerc = 0.0

    if x != 0:
        xPerc = x/MAX_X
    if y != 0:
        yPerc = y/MAX_Y
    if z != 0:
        zPerc = z/MAX_Z

    xPos = ORIGIN_X + (xPerc*(RANGE_X)) - HALF_RANGE_X
    yPos = ORIGIN_Y + (yPerc*(RANGE_Y)) - HALF_RANGE_Y
    zPos = ORIGIN_Z + (zPerc*(1/Z_SCALAR))

    pos = avg3Locs(xPos, yPos, zPos)
    cam.worldPosition = [pos[0], -pos[2], pos[1]]
    updateCameraMatrix()

def avg3Locs(xPos, yPos, zPos):
    if firstLoc == True:
        firstLoc = False
        x = xPos
        y = yPos
        z = zPos
        loc1 = [xPos, yPos, zPos]
        loc2 = [xPos, yPos, zPos]
    else:
        x = (xPos + loc1[0] + loc2[0])/3
y = (yPos + loc1[1] + loc2[1])/3
x = (zPos + loc1[2] + loc2[2])/3
loc1 = [loc2[0], loc2[1], loc2[2]]
loc2 = [xPos, yPos, zPos]

return [x, y, z]

def updateCameraMatrix():
    screenObj = bpy.data.objects["Screen"]
    screen = bpy.data.meshes["Screen"]

    if(screen != None):
        pa = mathutils.Vector(screenObj.matrix_world * screen.vertices[1].co)
        pb = mathutils.Vector(screenObj.matrix_world * screen.vertices[0].co)
        pc = mathutils.Vector(screenObj.matrix_world * screen.vertices[3].co)
        pe = cam.worldPosition
        n = cam.near
        f = cam.far

        invModelViewMatrix = cam.modelview_matrix.copy()
        invModelViewMatrix.invert()
        cam.projection_matrix = calcCameraMatrix(pa, pb, pc, pe, n, f) * invModelViewMatrix

def testCalcCameraMatrix():
    pa = mathutils.Vector([-1.0, 2.0, 0.0])
    pb = mathutils.Vector([1.0, 2.0, 0.0])
    pc = mathutils.Vector([-1.0, 2.0, 2.0])
    pe = mathutils.Vector([0.0, 0.0, 1.0])
    n = 1.0
    f = 100.0
    print("cam.lt")
    print(cam.localTransform)
    print("cam.wt")
    print(cam.worldTransform)
    print("cam.mvm")
    print(cam.modelview_matrix)
    print("cam.prm")
    print(cam.projection_matrix)
    cam.projection_matrix = mathutils.Matrix.Identity(4)
    cam.worldPosition = pe
    matrix = calcCameraMatrix(pa, pb, pc, pe, n, f)
    invModelViewMatrix = cam.modelview_matrix.copy()
    invModelViewMatrix.invert()
    cam.projection_matrix = matrix * invModelViewMatrix
    print("matrix", matrix)
    print("cam", cam.worldPosition)
    print("dir", cam.getScreenVect(0.5, 0.5))
print("add", (cam.worldPosition + cam.getScreenVect(0.5,0.5)))
print("add00", (cam.worldPosition + cam.getScreenVect(0.0,0.0)))
print("add01", (cam.worldPosition + cam.getScreenVect(0.0,1.0)))
print("add10", (cam.worldPosition + cam.getScreenVect(1.0,0.0)))
print("add11", (cam.worldPosition + cam.getScreenVect(1.0,1.0)))

def calcCameraMatrix(pa, pb, pc, pe, n, f):
    #print("pa",pa)
    #print("pb",pb)
    #print("pc",pc)
    #print("pe",pe)
    #print("n",n)
    #print("f",f)
    #Compute an orthonormal basis for the screen
    vr = pb - pa
    vu = pc - pa
    #print("vu = pb - pa = ", vu, (pc - pa))
    vr.normalize()
    vu.normalize()
    vn = vr.cross(vu)
    vn.normalize()
    #print("vr = pb - pa",vr)
    #print("vu = pc - pa",vu)
    #print("vn = vr X vu",vn)
    #Compute the screen corner vectors
    va = pa - pe
    vb = pb - pe
    vc = pc - pe
    #print("va = pa - pe",va)
    #print("vb = pb - pe",vb)
    #print("vc = pc - pe",vc)
    #Find the distance from the eye to the screen plane
    d = -(va.dot(vn))
    #print("d = - va . vn = ",d)
    #Find the extent of the perpendicular projection
    l = vr.dot(va) * (n / d)
    r = vr.dot(vb) * (n / d)
    b = vu.dot(va) * (n / d)
    t = vu.dot(vc) * (n / d)
    #print("l =", l ," r =", r ," b =", b ," t =",t)
# Projection Matrix

\[ p = \text{mathutils.Matrix}() \]
\[ p[0][0] = \frac{(2.0 \times n)}{(r - l)} \]
\[ p[0][1] = 0.0 \]
\[ p[0][2] = \frac{(r + l)}{(r - l)} \]
\[ p[0][3] = 0.0 \]

\[ p[1][0] = 0.0 \]
\[ p[1][1] = \frac{(2.0 \times n)}{(t - b)} \]
\[ p[1][2] = \frac{(t + b)}{(t - b)} \]
\[ p[1][3] = 0.0 \]

\[ p[2][0] = 0.0 \]
\[ p[2][1] = 0.0 \]
\[ p[2][2] = \frac{-(f + n)}{(f - n)} \]
\[ p[2][3] = \frac{-2.0 \times f \times n}{(f - n)} \]

\[ p[3][0] = 0.0 \]
\[ p[3][1] = 0.0 \]
\[ p[3][2] = -1.0 \]
\[ p[3][3] = 0.0 \]

# Rotation Matrix

\[ rm = \text{mathutils.Matrix}() \]
\[ rm[0][0] = vr.x \]
\[ rm[0][1] = vr.y \]
\[ rm[0][2] = vr.z \]
\[ rm[0][3] = 0.0 \]

\[ rm[1][0] = vu.x \]
\[ rm[1][1] = vu.y \]
\[ rm[1][2] = vu.z \]
\[ rm[1][3] = 0.0 \]

\[ rm[2][0] = vn.x \]
\[ rm[2][1] = vn.y \]
\[ rm[2][2] = vn.z \]
\[ rm[2][3] = 0.0 \]

\[ rm[3][0] = 0.0 \]
\[ rm[3][1] = 0.0 \]
\[ rm[3][2] = 0.0 \]
\[ rm[3][3] = 1.0 \]

# Translation Matrix

\[ tm = \text{mathutils.Matrix}() \]
\[ tm[0][0] = 1.0 \]
tm[0][1] = 0.0
tm[0][2] = 0.0
tm[0][3] = -pe.x

tm[1][0] = 0.0
tm[1][1] = 1.0
tm[1][2] = 0.0
tm[1][3] = -pe.y

tm[2][0] = 0.0
tm[2][1] = 0.0
tm[2][2] = 1.0
tm[2][3] = -pe.z

tm[3][0] = 0.0
tm[3][1] = 0.0
tm[3][2] = 0.0
tm[3][3] = 1.0

#Set Matrices
return p * rm * tm

def calcCameraMatrixGL(pa, pb, pc, pe, n, f):
    #Compute an orthonormal basis for the screen
    vr = pb - pa
    vu = pc - pa

    vr.normalize()
    vu.normalize()
    vn = vr.cross(vu)
    vn.normalize()

    print("vr = pb - pa",vr)
    print("vu = pc - pa",vu)
    print("vn = vr X vu",vn)

    #Compute the screen corner vectors
    va = pa - pe
    vb = pb - pe
    vc = pc - pe

    print("va = pa - pe",va)
    print("vb = pb - pe",vb)
    print("vc = pc - pe",vc)
# Find the distance from the eye to the screen plane
\[ d = -\langle \mathbf{v}_a, \mathbf{v}_n \rangle \]
```
print("d = - \mathbf{v}_a \cdot \mathbf{v}_n =", d)
```

# Find the extent of the perpendicular projection
\[ l = \langle \mathbf{v}_r, \mathbf{v}_a \rangle \cdot \left( \frac{\mathbf{n}}{d} \right) \]
\[ r = \langle \mathbf{v}_r, \mathbf{v}_b \rangle \cdot \left( \frac{\mathbf{n}}{d} \right) \]
\[ b = \langle \mathbf{v}_u, \mathbf{v}_a \rangle \cdot \left( \frac{\mathbf{n}}{d} \right) \]
\[ t = \langle \mathbf{v}_u, \mathbf{v}_c \rangle \cdot \left( \frac{\mathbf{n}}{d} \right) \]
```
print("l =", l, " r =", r, " b =", b, " t =", t)
```

# Save off GL state
```
matrixMode = bgl.Buffer(bgl.GL_INT, 1)
bgl.glGetIntegerv(bgl.GL_MATRIX_MODE, matrixMode)
oldMatrix = bgl.Buffer(bgl.GL_FLOAT, 16)
bgl.glGetFloatv(bgl.GL_PROJECTION_MATRIX, oldMatrix)
```

# Load the perpendicular projection
```
bgl.glMatrixMode(bgl.GL_PROJECTION)
bgl.glLoadIdentity()
bgl.glFrustum(l, r, b, t, n, f)
```

# Rotate the projection to be non-perpendicular
```
matrix = bgl.Buffer(bgl.GL_FLOAT, 16)
matrix[0] = \mathbf{v}_r[0]
matrix[1] = \mathbf{v}_u[0]
matrix[2] = \mathbf{v}_n[0]
matrix[3] = 0.0
```
```
matrix[4] = \mathbf{v}_r[1]
matrix[5] = \mathbf{v}_u[1]
matrix[6] = \mathbf{v}_n[1]
matrix[7] = 0.0
```
```
matrix[8] = \mathbf{v}_r[2]
matrix[9] = \mathbf{v}_u[2]
matrix[10] = \mathbf{v}_n[2]
matrix[11] = 0.0
```
```
matrix[12] = 0.0
matrix[13] = 0.0
matrix[14] = 0.0
matrix[15] = 1.0
```
```
bgl.glMultMatrixf(matrix)
newMatrix = bgl.Buffer(bgl.GL_FLOAT, 16)
bgl.glGetFloatv(bgl.GL_PROJECTION_MATRIX, newMatrix)
```
# restore the GL state
bgl.glLoadMatrixf(oldMatrix)
bgl.glMatrixMode(matrixMode[0])

retMatrix = mathutils.Matrix()
retMatrix[0][0] = newMatrix[0]
retMatrix[0][1] = newMatrix[1]
retMatrix[0][2] = newMatrix[2]
retMatrix[0][3] = newMatrix[3]

retMatrix[1][0] = newMatrix[4]
retMatrix[1][1] = newMatrix[5]

retMatrix[2][0] = newMatrix[8]
retMatrix[2][1] = newMatrix[9]

retMatrix[3][0] = newMatrix[12]
retMatrix[3][2] = newMatrix[14]

return retMatrix
import bgui
import bge

class BlenderUISystem(bgui.System):
    """
    A BGUI System setup to work with Blender
    BGUI library developed by Moguri
    https://code.google.com/p/bgui/
    """
    def __init__(self):
        # Initialize the system (replace the None with a path to a theme if you have one)
        bgui.System.__init__(self, 'bguiTheme')

        # Use a frame to store all of our widgets
        self.frame = bgui.Frame(self, 'Options', border=1, aspect=None, size=[0.25,0.27], pos=[0.375,0.02])
        self.frame.colors = [(1.0, 1.0, 1.0, 0.5) for i in range(4)]

        # Add widgets here and attach them to the frame
        lbl_color = (0.0, 0.0, 0.0, 1.0)
        txt_color = (0.2, 0.2, 0.2, 1.0)
        # Options label
        self.lbl = bgui.Label(self.frame, 'lblOptions', text='Options', pt_size=22, color=lbl_color, pos=[0.02, 0.92], options=bgui.BGUI_DEFAULT)
        y_temp = 0.11
        y_val = 0.83
        # Screen dimensions section label
        self.lbl = bgui.Label(self.frame, 'lblScreen', text='Screen dimensions(inches)', pt_size=20, color=lbl_color, pos=[0.025, y_val], options=bgui.BGUI_DEFAULT)
        y_val -= y_temp
        # Screen width label
        self.lbl = bgui.Label(self.frame, 'lblScreenW', text='Width', pt_size=20, color=lbl_color, pos=[0.025, y_val], options=bgui.BGUI_DEFAULT)
        # Screen width text_input
        self.txtScreenW = bgui.TextInput(self.frame, 'txtScreenW', text="000.000", pt_size=20, color=txt_color, size=[0.75,0.091], pos=[0.22, y_val-0.02], input_options = bgui.BGUI_INPUT_SELECT_ALL,
options = bgui.BGUI_DEFAULT)
self.txtScreenW.activate()
self.txtScreenW.on_enter_key = self.on_input_enter
y_val -= y_temp

# Screen height label
self.lbl = bgui.Label(self.frame, 'lblScreenH', text='Height',
pt_size=20, color=lbl_color, pos=[0.025, y_val],
options=bgui.BGUI_DEFAULT)

# Screen height text_input
self.txtScreenH = bgui.TextInput(self.frame, 'txtScreenH', text="000.000", pt_size=20,
color=txt_color,
size=[0.75,0.091], pos=[0.22, y_val-0.02], input_options =
bgui.BGUI_INPUT_SELECT_ALL,
options = bgui.BGUI_DEFAULT)
#self.txtScreenH.activate()
y_val -= y_temp

# Viewer Range section label
y_val -= 0.02
self.lbl = bgui.Label(self.frame, 'lblViewerRange', text='Viewer range(+- from origin)',
pt_size=20, color=lbl_color, pos=[0.025, y_val],
options=bgui.BGUI_DEFAULT)
y_val -= y_temp

# Viewer Range X label
self.lbl = bgui.Label(self.frame, 'lblViewerRangeX', text='X',
pt_size=20, color=lbl_color, pos=[0.025, y_val],
options=bgui.BGUI_DEFAULT)

# Viewer Range X text_input
self.txtViewerX = bgui.TextInput(self.frame, 'txtViewerX', text="000.000", pt_size=20,
color=txt_color,
size=[0.75,0.091], pos=[0.22, y_val-0.02], input_options =
bgui.BGUI_INPUT_SELECT_ALL,
options = bgui.BGUI_DEFAULT)
#self.txtViewerX.activate()
y_val -= y_temp

# Viewer Range Y label
self.lbl = bgui.Label(self.frame, 'lblViewerRangeY', text='Y',
pt_size=20, color=lbl_color, pos=[0.025, y_val],
options=bgui.BGUI_DEFAULT)

# Viewer Range Y text_input
self.txtViewerY = bgui.TextInput(self.frame, 'txtViewerY', text="000.000", pt_size=20, color=txt_color,
    size=[0.75, 0.091], pos=[0.22, y_val-0.02], input_options =
    bgui.BGUI_INPUT_SELECT_ALL,
    options = bgui.BGUI_DEFAULT)
#self.txtViewerY.activate()
y_val -= y_temp

# Viewer Range Z label
self.lbl = bgui.Label(self.frame, 'lblViewerRangeZ', text='Z',
    pt_size=20, color=lbl_color, pos=[0.025, y_val],
    options=bgui.BGUI_DEFAULT)

# Viewer Range Z text_input
self.txtViewerZ = bgui.TextInput(self.frame, 'txtViewerZ', text="000.000", pt_size=20, color=txt_color,
    size=[0.75, 0.091], pos=[0.22, y_val-0.02], input_options =
    bgui.BGUI_INPUT_SELECT_ALL,
    options = bgui.BGUI_DEFAULT)
#self.txtViewerZ.activate()
y_val -= y_temp

# A FrameButton widget
self.btn = bgui.FrameButton(self.frame, 'button', text='OK', size=[0.17, 0.10], pos=[0.8, 0.011],
    options=bgui.BGUI_DEFAULT)

    self.btn.on_click = self.button_click

# Create a keymap for keyboard input
self.keymap = {getattr(bge.events, val): getattr(bgui, val) for val in dir(bge.events) if
    val.endswith('KEY') or val.startswith('PAD')}

def on_input_enter(self, widget):
    print("Input_enter")

def button_click(self, widget):
    print("OK Button Clicked")

def main(self):
    """A high-level method to be run every frame.
    This handles things like sending mouse and keyboard events to BGUIL""

    # Handle the mouse
    mouse = bge.logic.mouse
    mouse_events = mouse.events
# Get the position
pos = list(mouse.position)
pos[0] *= bge.render.getWindowWidth()
pos[1] = bge.render.getWindowHeight() - (bge.render.getWindowHeight() * pos[1])

## Get the mouse state
if mouse_events[bge.events.LEFTMOUSE] ==
bge.logic.KX_INPUT_JUST_ACTIVATED:
    mouse_state = bgui.BGUI_MOUSE_CLICK
elif mouse_events[bge.events.LEFTMOUSE] ==
bge.logic.KX_INPUT_JUST_RELEASED:
    mouse_state = bgui.BGUI_MOUSE_RELEASE
elif mouse_events[bge.events.LEFTMOUSE] == bge.logic.KX_INPUT_ACTIVE:
    mouse_state = bgui.BGUI_MOUSE_ACTIVE
else:
    mouse_state = bgui.BGUI_MOUSE_NONE

# Send the position and state to BGUI
self.update_mouse(pos, mouse_state)

# Handle the keyboard
keyboard = bge.logic.keyboard

## Get the keys
key_events = keyboard.events
is_shifted = key_events[bge.events.LEFTSHIFTKEY] ==
bge.logic.KX_INPUT_ACTIVE or 
    key_events[bge.events.RIGHTSHIFTKEY] ==
bge.logic.KX_INPUT_ACTIVE

# Update keys for BGUI
for key, state in keyboard.events.items():
    if state == bge.logic.KX_INPUT_JUST_ACTIVATED:
        self.update_keyboard(self.keymap[key], is_shifted)

# Now setup the scene callback so we can draw
bge.logic.getCurrentScene().post_draw = [self.render]

def main(cont):
    """main() method to attach to a module controller in the BGE""

    own = cont.owner
    mouse = bge.logic.mouse

    if 'sys' not in own:
        # Create our system and show the mouse
        own['sys'] = BlenderUISystem()
mouse.visible = True
else:
    # The system's been created, so just run it
    own[‘sys’].main()
REFERENCES

  <http://www.infocomm.org/cps/rde/xchg/infocomm/hs.xsl/index.htm>

  <http://www.signexpo.org/>

  <http://www.digitalsignageexpo.net/>


[12] Haar Feature-based Cascade Classifier for Object Detection" *Cascade Classification — OpenCV 2.4.7.0 documentation*. OpenCV. 12 Nov. 2013
  <http://docs.opencv.org/modules/objdetect/doc/cascade_classification.html>


