VIRTUAL REALITY SIMULATION USING STEREOSCOPIC VISION AND MOTION TRACKING

A graduate project submitted in partial fulfillment of the requirements

For the degree of Master of Science in Computer Science

By

Stephen Burns
The graduate project of Stephen Burns is approved:

_________________________                                __________
Professor G. Michael Barnes                                Date

_________________________                                __________
Professor Robert Mcilhenny                                  Date

_________________________                                __________
Professor Rick Covington                                    Date

Signature Page
# Table of Contents

Introduction 1

Statement of Purpose 1

Project Game Description 4

Project Details 10

Technology Architecture and Program Class Structure 10

Ogre Graphics Engine 16

Creating the Scene 17

Stereoscopic Vision with Nvidia 3D Vision 22

Motion tracking with OpenCV 25

A. Setting up OpenCV 25

B. Calculating Target Colors for the Room 26

C. Finding Target Locations 27

D. Displaying OpenCV Window 28

E. Calculating Head Movements 30

F. Smoothing 32

Game Cycle 33

A. Keyboard Input to Control Avatar 34

B. Moving the Avatar 35

C. Checking Collisions 36

D. Updating Motion Tracking 37

E. Applying the Viewing Rotations 40

F. Updating Display Information 41

G. Removing the Viewing Rotations 42

Analysis and Conclusion 44
Tables and Figures

Figures
Figure 1: Anaglyph Glasses 2
Figure 2: Image of Game 5
Figure 3: Image of Turning in Game 6
Figure 4: Image of Nvidia Glasses with color Post-Its 7
Figure 5: Technology Diagram 10
Figure 6: Class File Relationship Diagram 11
Figure 7: Nvidia 3D Vision Glasses and Wireless Emitter 22
Figure 8: Image of Separation and Convergence 23
Figure 9: OpenCV Window Showing Color Locations. 30
Figure 10: Game update cycle 34

Tables
Table 1: Input Commands 9
Table 2: Baseapplication.cpp Methods 13
Table 3: Main.cpp Methods 14
Table 4: Map.cpp Methods 15
Table 5: GameObject.cpp Methods 15
Table 6: MotionTracking.cpp Methods 16
Table 7: Scene Map 21
Table 8: Stereoscopic Keyboard Commands 24
ABSTRACT

VIRTUAL REALITY SIMULATION USING STEREOSCOPIC VISION AND MOTION TRACKING

By

Stephen Burns

Master of Science in Computer Science

The graduate project is a three dimensional virtual reality game which uses user motion tracking for input and stereoscopic vision for display. The game created is a searching game where keyboard and user head movements are used for input. The stereoscopic glasses that the user wears are tracked to calculate head movement. The user can choose between controlling the avatar's movement with head movements or controlling the first person view with head movements. The graphics are rendered using the Ogre Graphics Engine. The motion tracking is calculated using the OpenCV library with a web-camera. Nvidia 3D Vision for stereoscopic vision was used with the project. This paper will discuss the motion tracking, stereoscopic display, and the Ogre Engine used in the project. The design and implementation of the project is described and evaluated. Lastly, possible extensions to the project are presented.
**Introduction**

**Statement of Purpose**

Motion tracking and stereoscopic vision have become increasingly popular in video games and entertainment. The three most modern home video game consoles (XBOX 360, PlayStation 3, and Nintendo Wii) all have video games which use motion tracking for input. The motion tracking in these video game consoles have given players a much more interactive experience than standard controls such as a controller, keyboard, or mouse. Although stereoscopic vision has been around for a long time, it has become much more popular in recent years, especially in movies. Stereoscopic televisions have become popular, and many movies are shown in stereo in movie theaters. Gamers can also play console and PC games with stereoscopic vision with the correct TV or monitor and hardware.

In stereoscopic viewing, each eye is presented a slightly different image to create the illusion of 3D. If done correctly, sections of the image appear to be closer to the person than the screen, and sections look further away than the screen as if the user is looking into a window.

Stereo viewing can be obtained in a variety of ways. Three popular stereo techniques are using anaglyph images, polarization, and shutter vision. In anaglyph images, each eye is presented a different image in different colors. The user wears color coded anaglyph glasses. There are various color combinations that can be used, the most popular being Red/Blue. The image contains two different color filtered images. When viewing the image through the glasses, one eye would see the red portion, the other would see the blue portion. The two image portions
represent pictures of the scene from the user's left eye and the other from the right eye. Anaglyph is a popular technique for movie posters and books, and can also be used on computer screens and projectors. [2, 11]

![Anaglyph Glasses Image](image)

Figure 1: Anaglyph Glasses. Image from reference [11]

Polarization is used most commonly in 3D televisions and movie theaters for stereoscopic vision. With polarization, each eye views full color images. On polarized glasses, each eye has separate polarization filters, and two images are displayed on the screen by using synchronized projectors. Each eye's view is shown and seen through a lens with the eye-specific polarization. [5]

The third stereoscopic technique is shutter vision. Nvidia 3D Vision uses shutter vision technology. In this technique, the screen shows one image at a time for a specific eye, and switches between the two images at a fast rate (Nvidia 3D Vision displays 120 frames per second and alternates between eye specific images every frame). The user wears glasses which dims the view for a specific eye since the closed lens presents an opaque color when the screen is displaying an image for the other eye. Thus, each eye views the correct eye-specific image. The 3D glasses synchronize with the images on the screen using a wireless emitter. Because the
dimming happens very quickly, the user does not notice the opaque flashing and the vision looks as if each eye is viewing the screen at the same time. This technique with the Nvidia 3D Vision glasses and emitter was used in my project. [9]

In motion tracking, object locations are tracked using devices such as a camera or other sensor. Motion tracking in video games has become increasingly popular, especially video game consoles. All three of the major video game consoles (Xbox 360, Playstation 3, and Nintendo Wii) use motion tracking. Nintendo Wii games are all based on motion tracking, and the other two consoles sell separate motion tracking hardware and games. The Xbox Kinect uses an infrared camera to calculate full body motion tracking for specific games on the Xbox 360. The camera calculates a 3-dimensional image of objects and is able to distinguish humans in the images [7, 8]. For more information about the Xbox Kinect, see reference [7].

The TrackIR can be used for motion tracking in computer games. The TrackIR camera is an infrared camera used to track head position and orientation. Computer games can use the TrackIR camera for motion tracking input control for head rotations and movement. The TrackIR camera is placed on top of the monitor, and the user places a hat device on his head. The TrackIR can be used for many driving and flight games. For more information about TrackIR and a list of games that can be used with see, see reference [6].

Combining motion tracking and stereoscopic vision gives an immersive and interactive experience in a 3D simulation. My graduate project is a simple searching 3D game using stereoscopic vision and motion tracking for input. I am using the Ogre Graphics Engine, NVidia
3D vision for stereoscopic vision, and the OpenCV library for computer vision motion tracking with a web-camera. Next I will describe the game project and how these three libraries were used.

**Project Game Description**

The project created is a searching game ran in stereoscopic 3D where the user’s head movements are tracked for input, and additional input is taken from the keyboard. Blue and purple post-ITs are placed on the sides of the NVidia Shutter Vision glasses. The post-ITs are the tracking targets that are used to calculate user head-movement. The user controls the avatar through a 3D environment to search for models. The user collects the models by running into the models. When the user finds all three models, the game is finished. There are walls placed throughout the environment that block the user from immediately seeing the models to create a maze, which increases the difficulty of the game.
Figure 2: Image of Game. The barrel is a target model, and the cube models are the walls.

Display information is turned on.

The avatar movement in the game was created to have a skating or bicycle feel. To accomplish this, the avatar continuously moves forward or backward when commanded to by pressing up or down arrow keys. The user can increase or decrease speed by pressing the up and down arrow keys, and the new speed is maintained. When turning with the left and right arrow keys or by leaning your face to the left or right, the avatar tilts into the turn (rolls or rotate about the Z coordinate axis). The avatar can jump by pressing space-bar key or tilting your face up. In a jump, the avatar lifts off the ground and reaches a maximum height, then returns to the ground.
While in the air, the avatar cannot turn (although the user can rotate the viewing camera), and keeps moving forward while in air. If the avatar collides with a wall, the avatar’s movement is stopped. After the avatar collides with a wall, for the avatar to move the user must change direction so that the forward movement does not cause a collision. The avatar must rotate and move forward with a different direction or move backwards to get away from the wall.

Figure 3: Image of turning in game. Display information is turned off.

The program reads from the web camera and locates the starting locations of the tracking targets. The web camera is read throughout the game, and head movements are calculated by comparing the tracking target locations with the starting locations.
Figure 4: Image of Nvidia glasses with color Post-Its.

The user can choose from a few control options for the motion tracking and keyboard input. The first form of control is where the avatar movement controls are entirely handled by the keyboard input (arrow keys for movement, and space-bar for jumping) includes turning, accelerating, decelerating, and jumping. The user’s head movements are tracked and change the viewing camera of the avatar, but not its movement. If the user moves his head to the left or to the right, the in-game camera turns the same direction.

In the second form of control, the head movements control the turning and jumping of the avatar, where left and right head movements control turning, and tilting your face up jumps the avatar). Acceleration in this control mode is handled by the keyboard by pressing up and down arrow keys. In this form of control, the amount that the user moves his head affects the turning speed of the avatar. The greater distance the user moves his head, the greater the viewing camera rolls (rotates about the Z axis) and the turning is greater on the Y axis. Rolling the camera when
turning temporarily changes the viewing camera for a leaning effect, but rotating about the Y-axis permanently changes the avatar’s orientation and the avatar is moved in this new direction.

The Nvidia 3D Vision requires specific hardware on the computer running the game. The Nvidia 3D drivers will work in 3D if a game renders in Direct3D 9 or later graphics API. The Ogre graphics engine for my game uses Direct3D9 to render, although Ogre can use other versions of Direct3D and OpenGL to render. The 3D stereoscopic settings can be modified by the user through the keyboard for stereo depth and separation. The game controls this by using functions from the NVAPI which gives programmers control over the NVidia graphics card [1]. Below is a table of input commands using the keyboard and head movements.

<table>
<thead>
<tr>
<th>Up Arrow Key (press)</th>
<th>Accelerate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Down Arrow Key (press)</td>
<td>Decelerate</td>
</tr>
<tr>
<td>Left Arrow Key (hold down)</td>
<td>Turn left</td>
</tr>
<tr>
<td>Right Arrow Key (hold down)</td>
<td>Turn right</td>
</tr>
<tr>
<td>Space bar key (press)</td>
<td>The avatar will jump.</td>
</tr>
<tr>
<td>F1 (press)</td>
<td>Show guess locations – Changes CV window to showing location of estimated target point to all pixels that are captured</td>
</tr>
<tr>
<td>F2 (press)</td>
<td>Change Motion Tracking Control Type</td>
</tr>
<tr>
<td>F3 (press)</td>
<td>Decrease stereoscopic separation</td>
</tr>
<tr>
<td>F4 (press)</td>
<td>Increase stereoscopic separation</td>
</tr>
<tr>
<td>F5 (press)</td>
<td>Decrease stereoscopic convergence</td>
</tr>
<tr>
<td>F6 (press)</td>
<td>Increase stereoscopic convergence</td>
</tr>
<tr>
<td>Input</td>
<td>Command</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>F7 (press)</td>
<td>Decrease motion tracking smoothing smoothing</td>
</tr>
<tr>
<td>F8 (press)</td>
<td>Increase motion tracking smoothing smoothing</td>
</tr>
<tr>
<td>F9 (press)</td>
<td>Decrease motion tracking update frequency.</td>
</tr>
<tr>
<td>F10 (press)</td>
<td>Increase motion tracking update frequency.</td>
</tr>
<tr>
<td>R (press)</td>
<td>Set and reset motion tracking target center. When R is pressed, the location of the targets will be calculated. This location will be used as the new center.</td>
</tr>
<tr>
<td>S (press)</td>
<td>Toggle Display information on and off.</td>
</tr>
<tr>
<td>Move head left</td>
<td>If in control mode 1, the game camera will rotate left while the avatar keeps moving the same direction.</td>
</tr>
<tr>
<td></td>
<td>If in control mode 2, the avatar will turn left. This will change the direction of movement.</td>
</tr>
<tr>
<td>Move head right</td>
<td>If in control mode 1, the game camera will rotate right while the avatar keeps moving the same direction.</td>
</tr>
<tr>
<td></td>
<td>If in control mode 2, the avatar will turn right. This will change the direction of movement.</td>
</tr>
<tr>
<td>Move head up</td>
<td>If in control mode 1, the first-person camera will up while the avatar keeps moving the same direction.</td>
</tr>
<tr>
<td></td>
<td>If in control mode 2, the avatar will jump.</td>
</tr>
<tr>
<td>Escape key (press)</td>
<td>Exit game.</td>
</tr>
</tbody>
</table>

Table 1: Input Commands
Project Details

We previously discussed how the game plays, we will now discuss the details about the project's implementation.

Technology Architecture and Program Class Structure

Below is an image showing the technology used in the project.

![Technology Architecture Diagram](image)

*Figure 5: Technology diagram*

C++ language is used to program the three libraries which are used in this project: NVAPI, OpenCV, and Ogre Graphics Engine. Ogre is a scene-graph based graphics engine used in this project. It can render with various versions of DirectX and OpenGL, which are low-level 3D graphics APIs. This project renders with DirectX. DirectX is used by Nvidia 3D Vision. NVidia's NVAPI is a software development kit which allows the programmer to access the Nvidia GPU and drivers [12]. This project uses functions from the NVAPI to modify the stereoscopic settings in the Nvidia 3D Vision. OpenCV is an open-source computer vision...
library. It was used for the motion tracking in this project. OpenCV libraries were used to capturing images with a web-camera and locating the user's head positions in the images.

The project code is separated into several C++ classes and files. Below is a relationship diagram of the classes.

![Class relationship diagram](image)

*Figure 6: Class relationship diagram.*

BaseApplication.cpp contains the code for the Ogre initializations such as the window, resources, camera, and viewports. Main.cpp contains the majority of the code for the game logic. This class contains code to create the scene and the game update cycles such as input processing, movement, collisions, and modifies the first-person camera. Main.cpp is a subclass of BaseApplication.cpp and updates properties from the BaseApplication.cpp during game cycles such as the camera. The methods in BaseApplication.cpp are shown in Table 2, and Main.cpp methods are shown in Table 3.
Main.cpp creates an instance of Map and MotionTracking. Map.cpp contains a 2D array map representing the game scene which Main.cpp uses to build the actual scene with the Ogre Graphics Engine. Methods in this class create and modify the 2D map. Map.cpp creates an instance of MotionTracking.cpp, to handle computer vision and motion tracking calculations. Methods in this class include calculating color scales, locating the users head position, and calculating motion tracking using the head positions. Main.cpp creates instances of GameObject.cpp, one for each of the walls and target models. This class contains pointers that reference the location, orientation, and texture information of the models. This class also contains information about whether the model is alive or dead, which are used for the target modes (they are dead when the avatar collides with them. The methods in Map.cpp are shown in Table 4, GameObject.cpp methods are shown in Table 5, and MotionTracking.cpp methods are shown in Table 6.
### Table 2: `BaseApplication.cpp` Methods

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>void chooseSceneManager(void)</td>
</tr>
<tr>
<td>bool configure(void)</td>
</tr>
<tr>
<td>void createFrameListener(void)</td>
</tr>
<tr>
<td>void createCamera(void)</td>
</tr>
<tr>
<td>void createViewports(void)</td>
</tr>
<tr>
<td>void go(void)</td>
</tr>
<tr>
<td>void loadResources(void)</td>
</tr>
<tr>
<td>bool setup(void)</td>
</tr>
<tr>
<td>void setupResources(void)</td>
</tr>
<tr>
<td>void windowClosed(Ogre::RenderWindow*)</td>
</tr>
<tr>
<td>void windowResized(Ogre::RenderWindow*)</td>
</tr>
<tr>
<td>Function</td>
</tr>
<tr>
<td>----------------------------------------------</td>
</tr>
<tr>
<td><code>applyViewRotations(void)</code></td>
</tr>
<tr>
<td><code>createCamera(void)</code></td>
</tr>
<tr>
<td><code>checkCollisions(void)</code></td>
</tr>
<tr>
<td><code>createScene(void)</code></td>
</tr>
<tr>
<td><code>frameRenderingQueued(const Ogre::FrameEvent &amp; evt)</code></td>
</tr>
<tr>
<td><code>keyPressed(const OIS::KeyEvent &amp;)</code></td>
</tr>
<tr>
<td><code>movementControlCamera(void)</code></td>
</tr>
<tr>
<td><code>processKeyboard(void)</code></td>
</tr>
<tr>
<td><code>resetViewRotations(void)</code></td>
</tr>
<tr>
<td><code>setupCV(void)</code></td>
</tr>
<tr>
<td><code>setupNV(void)</code></td>
</tr>
<tr>
<td><code>updateDisplayInfo(void)</code></td>
</tr>
<tr>
<td><code>updateGame(void)</code></td>
</tr>
<tr>
<td><code>updateMovement(void)</code></td>
</tr>
</tbody>
</table>

*Table 3: Main.cpp methods*
### Map.cpp

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>void createScene(void)</td>
</tr>
<tr>
<td>int getValue(int, int)</td>
</tr>
<tr>
<td>bool setValue(int, int)</td>
</tr>
</tbody>
</table>

*Table 4: Map.cpp Methods*

### GameObject.cpp

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>void setEntity(Ogre::Entity*)</td>
</tr>
<tr>
<td>void setNode(Ogre::SceneNode*)</td>
</tr>
<tr>
<td>Ogre::Entity* getEntity(void)</td>
</tr>
<tr>
<td>Ogre::SceneNode* getNode(void)</td>
</tr>
</tbody>
</table>

*Table 5: GameObject.cpp Methods*
### MotionTracking.cpp

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>calculateColorScales(void)</td>
</tr>
<tr>
<td>computeHeadMovements(void)</td>
</tr>
<tr>
<td>decreaseMotionTrackingFrequency(void)</td>
</tr>
<tr>
<td>findColorLocation(void)</td>
</tr>
<tr>
<td>getAmountMoved(void)</td>
</tr>
<tr>
<td>getHeadPosition(void)</td>
</tr>
<tr>
<td>increaseMotionTrackingFrequency(void)</td>
</tr>
<tr>
<td>setupCV(void)</td>
</tr>
<tr>
<td>update(void)</td>
</tr>
</tbody>
</table>

*Table 6: MotionTracking.cpp Methods*

#### Ogre Graphics Engine

We will now discuss a little about the Ogre Graphics Engine and the basic objects in this library which will be useful for understanding the project implementation. Ogre is an open-source scene-oriented based graphics engine built with C++. The engine can use various graphics APIs including versions of Direct3D and OpenGL, although only Direct3D was used in my project because only Direct3D will work with Nvidia 3D Vision. A full list features with Ogre can be found at reference [4].
Ogre uses a SceneManager object to manage all objects, lights, and cameras in the scene. Entity objects are any objects that can be seen on the scene such as mesh models and textures. SceneNode objects contain information about location and orientation. Entity objects are attached to SceneNodes, and the Entity object is displayed in the location and orientation of the SceneNode. Ogre organizes the nodes in a scene-graph data structure, and all nodes are attached to the SceneManager's root scene node. Ogre uses the right hand coordinate system, where the positive Z axis comes out of the screen.

Creating the Scene

The creation of the game scene is done in its createScene method in Main.cpp. The scene consists of a player camera, lights, a skybox, ground, walls and target models. The skybox is created using the setSkyBox method in Ogre SceneManager class. This creates a box of six images a distance away from the camera with the camera in the center. The skybox is always drawn at a set distance from camera, so when the avatar is moving around, the camera is always in the middle of the box which creates an illusion of sky.

There are various types of lights that can be created in Ogre. There is an ambient light which lights the entire scene equally, a point light which gives light in all directions equally from a point, a spotlight which is similar to a flashlight, and a directional light which is similar to sun or moon light. Ambient light and a directional light is used in the program. The ambient light is set with the following line of code:

mSceneMgr->setAmbientLight(Ogre::ColorValue(.5, .5, .5));
The mSceneMgr variable is the Ogre SceneManager object for the program. The ambient light is set using RGB (red, green, blue) color values ranging from 0 to 1, where all 0s is black and all 1s is white.

The directional light is created with following code:

```cpp
Ogre::Light* directionalLight = mSceneMgr->createLight("directionalLight");
directionalLight->setType(Ogre::Light::LT_DIRECTIONAL);
directionalLight->setDiffuseColour(Ogre::ColourValue(1, 1, 1));
directionalLight->setSpecularColour(Ogre::ColourValue(1, 1, 1));
directionalLight->setDirection(Ogre::Vector3(0, -1, 1));
```

The light is created using the Ogre SceneManager. Once the light is created, the properties of the light are specified such as the type of light, the diffuse and specular colors, and the direction.

Before the models are placed in the scene, an instance of Map is created, the class is from Map.cpp. A Map object contains a two-dimensional array representing a map of the scene. Each element in the array is an integer representing an open space, wall, target model, or camera location. A method in the Map class called `createScene` specifies the positions of the game elements on the two-dimensional array. The walls are placed around the map first randomly, then the locations of the target models and the camera are determined.

As previously discussed, the objects in the game will have an Ogre Entity which contain details about the model such as the mesh, texture, shading, bounding sphere, and other details. The objects will also have an Ogre SceneNode which contain information such as the position, orientation, and scale. A class GameObject is created which has pointers to the Entity and
SceneNode and has a property indicating whether the object is dead or alive (used for the target models, collided models are dead).

After the Map is created, the program loops through the 2D array, and creates a GameObject for each of the objects in the map. The Entity will be set to the correct mesh and texture, and the locations will be set according to their location in the Map. The locations of the models are calculated by their indexes in the map multiplied by the `sizeOfCell` variable. The size of the cell is set to size of the bounding radius of the wall model, which is the largest model used.

In Ogre, an Entity object contains information about each model such as the mesh file and a bounding radius. A SceneNode object contains information about the location, scale, and orientation. When adding a model to the program environment when creating the scene, both a SceneNode and Entity are created, and the Entity is attached to its associated SceneNode. As the program is looping through the map when creating the scene, if the map contains a wall at it's current index location, the Entity is first created with the SceneManager's `createEntity` method with the name of the mesh as an argument. A name must also be given to the entity, so I used “Wall_1” on the first wall then “Wall_2” and so on. A SceneNode is then created from the root scene node, and the entity is attached to the SceneNode. The following code creates the SceneNode and Entity:

```cpp
Ogre::Entity* entTemp = mSceneMgr->createEntity(s.str(), "cube.mesh");
Ogre::SceneNode* tempNode = mSceneMgr->getRootSceneNode()->createChildSceneNode();
tempNode->attachObject(entTemp);
```
s.str() is a C++ stringstream class containing the name of the wall (“Wall_1”, “Wall_2” etc.).

After the SceneNode and Entity are created, the model is moved to its correct location by changing the position of the SceneNode with the following line of code:

\[ x = \text{firstIndex} \times \text{sizeOfCell} - (\text{numberOfCells} \times \text{sizeOfCell} / 2); \]
\[ y = \text{entTemp->getBoundingRadius}, \]
\[ z = \text{secondIndex} \times \text{sizeOfCell} - (\text{numberOfCells} \times \text{sizeOfCell} / 2) \]
\[ \text{tempNode->setPosition}(x, y, z); \]

The variable \text{numberOfCells} is the number of cells in a row, which is equal to the number of cells in a column. The \text{firstIndex} and \text{secondIndex} values are the index values of the specific cell in the 2D map. The y value is set to the radius of the model so that the model stands on top of the ground. The x and z values are set to their map index multiplied by the size of the cell, which had been previously set. The x and z values are subtracted by half of size of the map (the number of cells multiplied by the size of the cell divided by 2) so that the center of the map is at the world coordinates \{0, 0, 0\}. An example of a generated map is shown in Table 7.
Table 7: Game Scene Map

The game uses a first-person perspective, so the in game camera acts as an avatar. The game camera is created with the following lines of code:

```
Ogre::Camera* mCamera = mSceneMgr->createCamera("PlayerCamera");
mCamera->setPosition(Ogre::Vector3(startingCameraX, 10, startingCameraZ));
mCamera->lookAt(Ogre::Vector3(0, 10, 0));
```

The camera is created with the scene manager. The position and lookAt properties are modified with methods from the Camera class given an XYZ coordinate. The number 10 is used as the Y value so that the camera is located and looks above the ground. The startingCameraX and
startingCameraZ values are set the same way as the X and Z values of the wall (using the indexes on the map and the sizeOfCell value).

Stereoscopic Vision using Nvidia 3D Vision

In my project, I used NVidia 3D Vision. Nvidia 3D Vision uses 3D shutter vision technology. In this technology, the user wears liquid crystal shutter glasses, a compatible monitor which has a 120 hertz rate, and a compatible Nvidia graphics card. The monitor flashes the image for the left and right eye, and synchronizes with the glasses which flash an opaque tint for each eye which blocks out the image the specific eye shouldn’t see. The synchronization happens using a wireless emitter. Each eye views 60 hertz, which is fast enough that the opaque tint switching back for forth is unnoticeable for most people [1].

*Figure 7: Nvidia 3D Vision glasses and wireless emitter.*

To create stereoscopic vision, the Nvidia driver duplicates and modifies a render target into eye specific locations based off of the viewing space, the location of the model or image, and the stereoscopic settings for convergence and separation as shown in the image below.
Figure 8: Image of separation and convergence

When an object's distance from the camera is less than the convergence, the object will appear to be closer to the user than the screen. Objects that are at the convergence distance away from the viewing camera will look the same to both eyes. Objects that are further away from the convergence will appear to be further away than the screen. The separation property is the distance between the two eye locations.

To use the Nvidia 3D Vision in the project, the 3D vision hardware had to be connected to the computer with correct drivers installed. The NVAPI was downloaded from the Nvidia website, and the header file and library were included and linked to my project. After the Ogre was initialized and before the main program logic is running, I created a method called `setupNV` which initializes the NVApi to be used for stereo throughout the program. `setupNV` method contains the following function calls:

```
NvAPI_Initialize();
```
The first two function calls initialize NvAPI and enable the automatic stereo vision. 

`NvAPI_Stereo_Enable();`  
`NvAPI_Stereo_CreateHandleFromIUnknown(Ogre::D3D9RenderSystem::getActiveD3D9Device(), &handle);`

The `NvAPI_Stereo_Enable()` function enables the automatic stereo vision. The `NvAPI_Stereo_CreateHandleFromIUnknown()` function creates a StereoHandle object from the active Direct3D device. `D3D9RenderSystem::getActiveD3D9Device()` method obtains the Direct3D 9 device that was initialized in Ogre. By the time this method is called, Ogre has previously initialized the Direct3D 9 window. The StereoHandle variable “handle” is used for stereo function calls for the NvAPI.

The stereo properties “separation” and “convergence” can be modified in the project through the keyboard. The following are the keyboard commands to modify the stereoscopic properties:

<table>
<thead>
<tr>
<th>Key</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3</td>
<td>Decrease Separation</td>
</tr>
<tr>
<td>F4</td>
<td>Increase Separation</td>
</tr>
<tr>
<td>F5</td>
<td>Decrease Convergence</td>
</tr>
<tr>
<td>F6</td>
<td>Increase Convergence</td>
</tr>
</tbody>
</table>

Table 8: Stereoscopic Keyboard Commands

The NVApi functions used to modify the stereoscopic settings are

`NvAPI_Stereo_DecreaseSeparation(StereoHandle handle),`  
`NvAPI_Stereo_IncreaseSeparation(StereoHandle handle),`  
`NvAPI_Stereo_DecreaseConvergence(StereoHandle handle),`  
and
NvAPI_Stereo_IncreaseConvergence(StereoHandle handle). The StereoHandle used as an argument was the handle obtained in setupNV() method from the active Direct3D 9 device.

Motion Tracking using OpenCV

In the project, head movements are tracked using a web-camera and OpenCV library. References [4] and [12] were used to learning OpenCV. Head movement tracking was used in the game for rotating the view camera of the first person avatar, and is also used for controlling the avatar depending on the control settings in the game. To track the head movements, color post-its were placed on the sides of the Nvidia shutter vision glasses. The location of these two colors are found and are used to calculate head movements of the user.

A. Setting up OpenCV

At the start of the project, before the window is initialized and before the main update loop of the program, the MotionTracking object from MotionTracking.cpp is instantiated, and the following code sets up the camera in the constructor:

```
CvCapture capture = cvCaptureFromCAM( 0 );
cvSetCaptureProperty(capture, CV_CAP_PROP_FRAME_WIDTH, 300);
cvSetCaptureProperty(capture, CV_CAP_PROP_FRAME_HEIGHT, 300);
```

The first line of code initializes the web-camera. The following two lines of code sets the width and height properties of the image to 300 pixels. Having too high of a pixel count decreases the
performance of capturing the image during the game, but too low of the pixel count reduces the
image quality which will reduce the accuracy of the motion tracking.

B. Calculating Target Colors for the Room

After the camera capture is setup, the CalculateColorScales method, from MotionTracking.cpp,
is called. This method is called once at the beginning of the game. The color of each pixel has a
red, green, and blue color values (also called RGB color values) ranging from 0 to 255. Color
black has RGB values \{0, 0, 0\}, and white is \{255, 255, 255\}. The light of the room affects the
RGB values. However, the light of the room affects each of the RGB values by the same percent.
For example, in an average room, a particular color might have values \{20, 20, 40\}. In a lighter
room, each of those RGB values might be raised by 10\%, which would be \{20 * 1.1, 20 * 1.1, 40
* 1.1\} = \{22, 22, 44\}.

For the two color post-its I am tracking, the purple post-it has RGB color values \{blue = 116,
green=58, red=175\} in average room lighting. The blue post-it has color values \{blue=178,
green=130, red=0\}. I found the RGB values of these reading images from a web cam using
OpenCV, and displaying the RGB values of the pixel in the middle of the image onto a console
window while holding the color post-it in front of the camera.

Because the lighting of the room can be different for the game because of the time of day and the
lighting of the room, scales were applied to the average RGB values at the beginning of the game
in order to calculate our target color value. After applying various scales to our RGB values
ranging from .7 to 1.3, whichever scale that found the most pixels at the beginning of the program was applied to the average color values and became our target color value for the game.

The target color RGB values are found by first obtaining an image from the web-camera. A scale is multiplied to the target RGB element values of a target color (starting with .7). Each pixel on the image is compared to the target RGB value, and a counter keeps track of the number of pixels match the target RGB values or are close to them. Then the next scale (.1 higher than the previous scale) is applied to the target RGB values and counts the number of matching pixels. When all scales have been testing, the scale that matched the most pixels is applied to the target RGB values of the color, and is used throughout the remaining of the program.

C. Finding Locations of Targets

We previously discussed setting up the OpenCV and calculating the color scales which are both done at the beginning of the program run-time. We will now discuss the motion tracking logic that is processed throughout the run-time of the game which is finding the locations of the color targets, and calculating the user's head movements. OpenCV logic involved capturing a frame image from the web-camera, then converting the image into a IplImage which is the OpenCV structure. Once the IplImage is obtained, findColorLocations in MotionTracking.cpp is called to find the locations of the two post-its on the side of the user's head.

To find the locations of the post-its, a frame is obtained from the web-camera and is stored as a IplImage. The RGB elements of each pixel are compared to the RGB elements of the target blue and purple colors. If each RGB element of the pixel is close (or within a threshold) to the RGB elements of the target blue color, then the X and Y coordinates of the pixel on the image is added
to a vector from the C++ Standard Template Library. If the RGB values of the pixel are not close to the blue target value, the values for the pixel are compared to the target purple values. If the pixel RGB values are close to the target purple RGB values, the pixel locations are added to a list different than the list of blue matches. Then the next pixel is evaluated.

When all pixels have been evaluated, there will be a list of blue pixel locations and a list of purple pixel locations. If the size of each list of greater than or equal to 1 (there was at least 1 pixel location added to the list), the average X and Y locations are calculated for each list. These averages are used as the locations of the colored post-its. At this point, we will have a X and Y coordinate location for the blue post-it, and a X and Y coordinate location for the purple post-it. If key 'R' had been pressed to set or reset the starting color locations before the findColorLocations method call, the found locations are used as the starting color locations. The starting color locations will remain the same throughout the program unless the color reset key 'R' is pressed again on the keyboard. If 'R' is pressed, the next calculated color locations will be our starting locations. The coordinate locations will be our current color locations. If no blue pixels were found or no purple pixels were found, the image from the web-camera is disregarded and the current color locations will remain the same.

D. Displaying OpenCV Window

To visually see the motion tracking, an OpenCV window is shown after the average blue and purple pixel locations are found. As the user moves his head around, the window displaying the motion tracking will reflect the movements. To view this window, the game screen must be windowed instead of full-screen. However, the stereoscopic vision will not be shown in windowed mode.
The OpenCV window can be displayed in two different ways in the project. The first display mode will show all of the blue pixels as blue, the purple pixels as purple, and the rest of the pixels will be black. The two display modes can be toggled by pressing F1 on the keyboard. To implement the first display mode, the RGB elements of each pixel of the IplImage, which is the image taken from the web-camera, are changed while processing the image in the `findColorLocations` method. While going through each pixel and determining whether it is blue, purple, or other: the RGB elements of the pixel are altered. If it matches the blue target RGB values, the red element of the pixel is set to 0, the green element is set to 0, and the blue element is set to 255. This color setting is the most blue color that can be created. If the pixel matches the purple target values, the pixel values are set to {255, 0, 255}. If it doesn't match the blue or purple pixel values, the RGB values are set to black {0, 0, 0}. After the locations of the Post-Its are calculated, the OpenCV function `cvShowImage` displays the modified image using the IplImage as an argument. This first display mode is useful determining how well the program is processing the image.

The second display mode will draw an image showing the locations of the average color pixels. The image will display squares around the average locations. To create this image, the program will loop through each pixel of the IplImage after the averages have been calculated. If the current pixel location is within 5 pixels of the average blue X-axis location and within 5 pixels of the average blue Y-axis location, then the pixel RGB values are changed to blue {0, 0, 255}. If the current pixel is within the same range of the purple average location, the pixel RGB values are changed to purple {255, 0, 255}. Otherwise, the RGB elements of the pixel are set to black {0, 0, 0}. When all pixels have been modified, the image is displayed to the screen using the
same OpenCV function as display mode one \((cvShowImage)\). This second viewing mode is useful for determining how the motion tracking is working. A screen shot of this second viewing mode OpenCV window is shown below.

![OpenCV window showing color locations.](image)

**Figure 9**: *OpenCV window showing color locations.*

E. Calculating Head Movements

We previously discussed calculating the locations of the target colors. We will now discuss using these values to calculate the user's head movements. The method \(\text{calculateHeadMovements}\) compares the current color locations with the starting color locations to calculate head movements. There is a threshold defined so that the user can move his head around slightly without affecting the game. The following code checks if the blue post-it moved to the left:
if (location_of_blue.x > starting_location_of_blue.x + movement_threshold)
{
    blue_left = true;
    percent_blue_move_left = min(1, abs((location_of_blue.x - last_location_of_blue.x - movement_threshold) / max_move));
} else
    blue_left = false;

The + X coordinate is left because the web-camera is facing the user. The location_of_blue variable is a pixel coordinate structure of the current blue location calculated by findColorLocations method, and the starting_location_of_blue is the starting location calculated at the beginning of the program or when the reset key is pressed. If the current blue X location is further to the left than the starting location plus the threshold, a percent of the move is calculated from the threshold location to a maximum location. Similarly, blue right is calculated using location_of_blue.x and starting_location_of_blue.x. Purple right and left are calculated using location_of_purple.x and starting_location_of_purple.x. Blue up and down are calculated using location_of_blue.y and starting_location_of_blue.y. Lastly, purple up and down are calculated using location_of_purple.y and starting_location_of_purple.y.

If both the purple and blue post-its have moved up (blue_up and purple_up are both true), the head position is set to moving up (head_position = HEAD_MOVE_UP). The percent moved up is calculated as the average of the percent_blue_move_up and percent_purple_move_up variables. Likewise, head_position is set to HEAD_MOVE_LEFT if both the purple and blue post-its have moved left, HEAD_MOVE_RIGHT if both post-its have moved right, and HEAD_MOVE_DOWN if both post-its have moved down. Percent head movement calculations...
are calculated in each direction the same way as percent_move_up. By calculating the head movement direction and the percentage of movement, we successfully found the movement of the user's head.

F. Smoothing

The values that were calculated in the previous section can be used directly in the game and would be successful motion tracking. Alternatively, a smoothing operation can be used to make the motion tracking more smooth. Smoothing is used in the project. The smoothing operation is applied in the calculateHeadMovements method in MotionTracking.cpp. When using smoothing, a percentage of movement value is added to a queue each time the motion tracking is updated throughout the game. The program will calculate the average of values in the queue, and this will be used in the game instead of the direct value that is found at each motion tracking update. As the game progresses and the motion tracking is updated, the queue will reach its maximum size. When this happens, the oldest value will be “popped” out of the queue as a new one is added.

The maximum queue size can be modified by pressing F7 on the keyboard to decrease the smoothing queue size, and F8 to increase the queue size. If the maximum queue size is too big, then the motion tracking will be less responsive as the user moves his head. If the queue size is too small, the motion tracking might be too responsive and less smooth. The maximum queue size is 1 at the beginning of the game, and must be increased by keyboard command for the smoothing to be in effect.
Game Cycle

The program begins by initializing the motion tracking (initializing the camera, calculating color scales), initializing Ogre, and initializing NvAPI. The program then begins the drawing and updating the game logic. To update the game logic, the method `updateGame` is called from the `frameRenderingQueued` method, which is called while the frame is being drawn. Using the `frameRenderingQueued` method, the frame is drawn and the game logic is being updated for the following frame in parallel by the GPU to improve performance. All game updates will be applied to the following frame. Figure 10 is a diagram of the game update cycle flow.
Figure 10: Game update cycle.

A. Keyboard Input to Control Avatar

We will start by analyzing update step 2: processing the keyboard, and then look at resetting the viewing rotations at the end. The OIS (Object-Oriented Input System) library, which comes with the Ogre Engine, is used to process keyboard input. In the `processKeyboard` method (called from `updateGame`), key pressed are processed which modify the avatar's movement. The OIS
Keyboard method `isKeyDown(OIS::KC_SPACE)` returns a boolean value of true if the space key is pressed. If the space key is pressed, the avatar begins to jump. If the left or right arrow keys are pressed, the avatar turns by rotating a fixed degree along the Y-axis. If the up arrow key is pressed, an integer variable `moving_speed` is increased. This variable is used as a step size to move the avatar forward when updating the avatar movement. This variable can be a positive or negative value, and has a maximum and negative value. When the down arrow key is pressed, the value for `moving_speed` is decreased. Because the game logic is updated frequently (about 60 times a second), if a user presses a key, the key is likely to be pressed for multiple times the program passes through the `processKeyboard` method. Modifying the speed and jumping are avatar movement changes that should be modified only once per key press. To make these modification happen only once per key press, boolean flags are used to specify that the key was pressed and the not released. However, turning by pressing the left or right arrow keys does not use these flags, so the avatar turns as long as key is held down.

B. Moving the Avatar

After the keyboard input is processed, the `updateMovement` method will handle the avatar movement. To move the avatar, the main player's camera will translate forward about its Z-axis. The following line of code moves the avatar forward or backward, depending on the `moving_speed` variable:

```cpp
mCamera->moveRelative(Ogre::Vector3(0, 0, moving_speed));
```

The variable `mCamera` is the main first-person game camera. The variable `moving_speed` is used as a step size and can range from a negative value for maximum speed backwards, to a positive value for maximum speed forwards. The third parameter for the `Vector3` constructor is the Z
coordinate. The moveRelative method moves the avatar according to the camera's orientation, instead of by the world coordinate system.

If the avatar is jumping, the avatar will either be on its way up or going down. If it is going up, the avatar takes a step along the positive Y-axis with the following code:

```cpp
mCamera->moveRelative(Ogre::Vector3(0, 1, 0));
```

The second argument of the Vector3 constructor is the amount moved along the Y-axis. If the position of the avatar is at the maximum height, the avatar will instead move a -1 step on the Y-axis, and continue until it reaches the ground.

C. Checking Collisions

Collisions in the game are detected when the avatar moves into the walls, and into the target objects. The distance formula is used: if the distance between the avatar and an object is smaller or equal to the bounding sphere of the object, then the avatar is collided with the object. The Ogre Entity object has a method called `getBoundingRadius` which returns the radius of the model. The Ogre Node object has a method called `getPosition` which returns a Vector3 object containing the X, Y, and Z coordinates. The Vector3 class has a distance method to calculate the distance between two points. The following code checks the collisions of the walls:

```cpp
bool collision = false;

for (int i = 0; i < walls.size() && !collision; i++)
    if (walls[i].node->getPosition().distance(mCamera->getPosition()) <=
        walls[i].entity>getBoundingRadius())
        collision = true;
```
The game checks if there is a collision by calculating the distance between the position of the avatar and the position of each wall, and checking if the distance is less than or equal to the size of the mode's radius. If the avatar collides with a wall, the avatar moves a step in the opposite direction which gets the avatar out of the collision. Then the movement speed step size is set to 0.

```cpp
mCamera->moveRelative(Ogre::Vector3(0, 0, -moving_speed));
moving_speed = 0;
```

Collisions with the avatar and the target models are calculated in the same way as the walls. When the avatar collides with a target, the target model is set to invisible. After all three target models have been collided with, the game is finished.

D. Updating Motion Tracking

Because the motion tracking logic requires a lot of processing and does not need to be updated each frame, the motion tracking is broken down into sections (capturing the image, decompressing to a IplImage, locating the target colors, and calculating head movements) and processed over a period of multiple frames. Increasing number of frames that the motion tracking is updated over can improve performance. The motion tracking can be processed every frame, every 6 frames, every 12 frames, or every 18 frames. The user can press F9 to decrease the frequency (over a greater number of frames) that the motion tracking is updated, and F10 to increase the frequency.
In the `update` method in MotionTracking.cpp, a variable is used to keep track of the frame. At end of each frame, the frame counter is incremented by 1 and modulus by framesPerUpdate with the following code:

```cpp
f_counter = (f_counter + 1) % framesPerUpdate;
```

A switch statement uses the frame counter variable, and a specific motion tracking section is computed depending on the frame count. We will examine the section allocation when the motion tracking is updated every 6 frames. If the counter is 0, the OpenCV function `cvGrabFrame()` is called which quickly captures a frame from the web-camera. If the frame is 1, the function `cvRetrieveFrame` is called. This function creates an IplImage object from the image that had been obtained from the web-camera. The function `cvRetrieveFrame` has a greater performance with a decreased pixel value, which is why the number of pixels of the capture was changed to 300 by 300 in the `setupCV` method. If the frame is 4, the `findColorLocations` method is called, where the locations of the color Post-its are found. On frame 5, the `computeHeadMovements` method is called, which determines the head movements of the user, calculates the percent of move, and applies the smoothing operation. When the framesPerUpdate is 12, the update motion tracking algorithm skips a frame between each motion tracking process. It skips two frames when the framesPerUpdate is 18.

In the `updateGame` method, after the switch statement which handles the motion tracking depending on the frame count, the method `movementControlCamera` is called every frame. This method is only used in control mode 2, where the motion tracking controls movement of the avatar. This method is used each frame instead of only the frames that the motion tracking is updated because while turning, the avatar is turned at each frame. This method uses the head
positions and the percent of movement. These values remain constant through the motion tracking processing until they are modified by `computeHeadMovements` when the motion tracking cycle is completed. Unlike the keyboard controls for the turning, control mode 2 allows for a range of turning values based on the percent that the user's head is turned. The turning and jumping controls are handled with the following code:

```cpp
if (control == CTR2)
{
    if (goingUp == false && goingDown == false)
    {
        if (head_position == HEAD_MOVE_LEFT)
        {
            mCamera->yaw(Ogre::Radian(Ogre::Degree(.5 * percent_move_left)));
        }
        else if (head_position == HEAD_MOVE_RIGHT)
        {
            mCamera->yaw(Ogre::Radian(Ogre::Degree(-.5 * percent_move_right)));
        }
        if (head_position == HEAD_MOVE_UP)
        {
            if (goingUp == false && goingDown == false)
            {
                goingUp = true;
            }
        }
    }
}
```

The avatar turned by rotating about the Y-axis in the direction that the user's head is turned. The percent values are decimal numbers that can range from 0 to 1. The higher the value, the greater the turn on each frame. If the head position is up and the avatar is not currently jumping, the `goingUp` boolean flag is set to true, which causes the avatar to start moving up in the `updateMovement` method. While jumping, the avatar is not user controlled. The avatar will
continue moving up or down in the air, and moving forward until the avatar lands back on the ground.

E. Applying the Viewing Rotations

The last section of the game update before the following frame is drawn is applying the viewing rotations. The viewing rotations are handled by the `applyViewRotations` method. Viewing rotations are temporary rotations for the following frame and are removed after the frame. When the avatar is turning, the camera rotations about the Z-axis in the correct direction, which creates the appearance that the avatar is leaning into the turn, as if on a bicycle. When the user is in control mode 1 (where the user presses the left and right arrow keys to turn), the avatar has a fixed degree of rolling rotation. To roll the camera left when there is a fixed degree, the following line of code is used:

```c++
mCamera->roll(Ogre::Radian(Ogre::Degree(roll_degree)));```

`mCamera` is the first person camera of the game. `roll_degree` is a set number. The camera will roll to the right when negative `roll_degree` is used.

When the game is in control mode 2 (where the avatar's turns are controlled by the user's head movements), the camera is rolled by a percentage of the `roll_degree` value. So the further the user moves his head, the greater the camera will be rolled. The percentage used is the percent that the user turns his head from the threshold to the maximum move location, which is calculated in the `computeHeadMovements` method and described in the previous section (Update Motion Tracking). The line of code to roll the avatar left is:

```c++
mCamera->roll(Ogre::Radian(Ogre::Degree(roll_degree * percent_move_left)));```
The variable percent_move_left is a global variable and is modified in the
`computeHeadMovements` method. This variable will range from 0 to 1. If the avatar is turning
right, negative roll_degree and percent_move_right is used.

Aside from turning the avatar, the `applyViewRotations` method also handles the head movement
rotations from control mode 1 (where the user’s head movements modify the viewing rotation of
the avatar instead of controlling the avatar). If the user turns his head left or right, the camera
will rotate about the Y-axis. If the user turns his head to the left, the following line of code will
rotate the camera:

```cpp
mCamera->yaw(Ogre::Radian(Ogre::Degree(30 * percent_move_left)));
```

The variable percent_move_left will range from 0 to 1, so the degree rotated will range from 0 to
30. If the users head moves his head to the right, percent_move_right variable will be used, and
the degree will range from 0 to negative 30.

F. Updating Display Information

Information is displayed on the screen showing game settings and other information about the
current game. This information is updated at the end of update cycle, and is updated in the
`updateDisplayInfo` method in Main.cpp. The information shown in the control move (whether
head movements control avatar or in-game view camera), Head Direction (Normal, Left, Right,
or Up), Percent of Head Movement (ranging from 0 to 100), Targets Remaining, motion tracking
update frequency, motion tracking smoothing queue size, and Time Played in seconds. The
information can be turned on and off by pressing S.
G. Removing the Viewing Rotations

After the display information is updated, the update for the frame has been completed. We have previously discussed each of the game cycle update steps except for the first step: removing the viewing rotations. The motion tracking and character movement has been updating, and the viewing rotations have been applied. The frame can now be drawn. At the beginning of the next update, the viewing rotations are removed so that the avatar movement and turning rotations are updated correctly. Because the moveRelative method is used when moving the avatar forward and backward (moveRelative moves the camera relative to its orientation), the viewing rotations are removed for the update so that the avatar is translated in the correct direction. Before the avatar turns, the roll degree has to be removed.

To remove the viewing rotations, the resetViewRotations method is called at the beginning of the update. The method resetViewRotations is the exact opposite as applyViewRotations method. If the game is in control mode 1 and the user turns his head left, the avatar will rotate about the Y-axis degree 30 * percent_move_left in applyViewRotations method. In resetViewRotations method, the avatar will rotated about the Y-axis -30 * percent_move_left. If the game is in control mode 2 and the user turns his head left, the avatar will roll degree 30 * percent_move_left in applyViewRotations method. In resetViewRotations method, the avatar will roll -30 * percent_move_left. The methods are also opposites for head movements to the right, and for turning with keyboard input. The head movement variables, which indicating the direction of move and the percentages, will remain the same from applyViewRotations and resetViewRotations methods because the motion tracking logic has not been updated between
these two methods. Because these variables don't change, the resetting the view rotations will always reset the view rotations exactly back to the normal orientation.
Analysis and Conclusion

Project Analysis

Having completed the project implementation and testing it, various observations of the feel of the game and input modes can be made. For input mode 1, where head movements rotate the games first-person camera, the most reasonable head movements to capture this is to have the user rotate his head rather than leaning in a direction. Rotating your head during the simulation had problems for two reasons. The first reason is that it felt unnatural to keep your eyes forward to look at the screen while rotating your head. Rotating your head might work better if the screen was surrounding you which I'll discuss further in the next section. The second reason is that because the user is wearing the stereoscopic vision glasses, as you turn your head and keep your eyes forward, you are looking through more glass which changes the color do the screen. Because of these two issues, I allowed head movements in this control mode can be made by either head translations (or leaning in a direction), or by rotations.

Control mode 2, where the user's head movements controlled the turning of the avatar, felt the most natural. Leaning into turns with your head as the first-person camera rotated about both the Y and Z axis (rolling and turning in the correct direction) gave the illusion of being on skateboard or bicycle. Having to use the keyboard for acceleration was a slight break-down of this illusion, but not too much as to make the game feel awkward or unnatural. Additional motion tracking controls over the avatar to remove the keyboard could be added on to the project.

The stereoscopic vision in combination with the motion tracking controls gave the game an immersive feel. A drawback was with head-rotations viewing the screen through the glasses, but
that would probably have been in issue on any stereoscopic vision technology. Another drawback is that the game had to be full-screen for the stereo vision to work. When the game was in full-screen, the OpenCV window displaying the calculated motion tracking locations cannot be seen. Other than these issues, Nvidia 3D Vision worked well with the project. The stereoscopic settings can be modified through the keyboard and function calls, and through the Nvidia Control Panel outside of the program for the users preferences.

A challenge of this project for me was using a variety of libraries into one project, especially performance using OpenCV and Ogre Graphics Engine. My target frames per second (FPS) for this project was 60, which is a common expectation for graphics simulation and video games. I was working on high performance computers (Nvidia 3D Vision requires relatively high performance computers) and the graphics in the project was not complex, so there was no challenge in obtaining 60 frames or more per second without any motion tracking (I later set the program so that it would not draw or update the game more than 60 times a second). Images can not be obtained and processed with a web-camera and OpenCV 60 times a second on the computers I was working on. Even simply obtaining an image from a web-camera with cvQueryFrame() function without any processing on the image can not be done 60 times a second. To include motion tracking without reducing the performance of the game, the motion tracking was accomplished in portions over a period of multiple frames. For my computer, updating the motion tracking over 6 frames is the best because it's motion tracking is updated frequently enough to feel instant, and keeps the FPS between 55 and 60. Using the NVApi did not reduce the performance of the project. NVAPI functions were called only before the game started for initialization and when keys on the keyboard were pressed.
Ideas for Further Development

If more time was given to work on this project, many additions and improvements can be made. Additions can be made to the motion tracking such as hand and body tracking, as well as the distance from the user to the web-camera. A web-camera and OpenCV can be used for full body tracking, or a different device can be used such as the Microsoft Kinect. Using hand and body motion tracking, the commands that were used given the keyboard can be given through motion tracking. Distance from the camera can be measured by detecting locations on the users body, and calculating the distance between them. Distance can also be measured by using two web-cameras, where one camera is placed on the left side of the screen and one is placed on the right side. Both cameras will detect the same section of the users body. As the user moves closer to the screen, the body section that was detected on one camera will be further away from the location that was found on the other camera.

Another way to make the virtual reality more immersive would be to use multiple monitor surrounding the user in combination with motion tracking. Although Nvidia 3D Vision supports multiple monitors using stereoscopic vision, multiple 3D monitors might be too expensive for this, but using regular display would still create a good effect. The user can look to his surroundings for the first-person display and control the avatar with body motions.
Conclusion

In conclusion, this project was successfully completed because the project was implemented, and the project was explained and analyzed in this paper. The following software libraries and hardware were used to create the project implementation: Ogre Graphics Engine library to build the game, Nvidia 3D Vision hardware with the NvAPI library to modify the stereoscopic properties, and the OpenCV library with a web-camera to capture images for motion tracking. A searching game was created where the user viewed the scene in stereoscopic 3D and the user can modify the stereoscopic properties. Input of the game was a combination of motion tracking and keyboard. The stereoscopic glasses that the user wears are tracked to calculate head movement. One mode of the game was created where the user can control the turning of the avatar with head movements, and another mode was created where the first-person view in the game was controlled by the user head movements. Control mode settings can be modified the user. The motion tracking smoothing and the motion tracking update frequency can be modified by the user. Lastly, the project is successful because this paper has described the gameplay of the implementation, project architecture, implementation details, and an analysis of the implementation.
References


<http://nzphoto.tripod.com/sterea/anaglyphs.htm#intro>

*Gary Bradski & Adrian Kahler*. *O'Reilly*. October 1, 2008

<http://www.ogre3d.org/>

<http://science.howstuffworks.com/3-d-glasses2.htm>

<http://www.naturalpoint.com/trackir/>


<http://www.nvidia.com/object/3d-vision-main.html>
<http://developer.nvidia.com/nvapi>

<http://en.wikipedia.org/wiki/Anaglyph_image>

<http://opencv.willowgarage.com/wiki/Welcome>