**Oculus Rift Programming for Biomedical Application**

Moshe Stauber and Theodore Raphan

Dept. of Computer and Information Science

Brooklyn College

Brooklyn, New York 11210

**Introduction:**

The purpose of this project is to develop a gravity-dependent model for generating moving images in a virtual reality. That is, as the head is tilted with regard to gravity, the image and its motion relative to the head can be made to move at a specific angle relative to the head. With this type of visual environment subjects can be tested for reaction to virtually moving environments relative to the head at arbitrary angles. The equipment that has been used in this development is the Oculus Rift, which is a Head Mounted Display worn by the user over the eyes and can generate 3-D immersive virtual environments. Associated with this system is an internal Inertial Measurement Unit (IMU), which is capable of tracking the angular acceleration and velocity as well as the angular position of the head relative to the spatial vertical. The system also has an infrared camera, which can be mounted on a space fixed frame, providing measurements of the linear displacement of the head. The externally mounted camera together with information from the IMU provides measurements of the six degrees of freedom of head movement in space. In this report, we describe the hardware architecture, program environment, the Oculus SDK and associated libraries, and the program that was developed to generate moving images that are used in testing the visual component of vestibular activation when the head is moved.

**System Configuration:**

The Oculus Rift is a relatively new device and is lacking in documentation or examples from which to learn. We therefore describe the setup of the Oculus Rift and the interface to the computer that was used in this development.

The Oculus Rift is head mounted display, which has additional hardware to allow positional tracking of the user. The complete hardware has been marketed as the Oculus Rift DK2 (Development kit 2).

This Oculus Rift DK2 includes the following:

1. Head Mounted Display (HMD) - The Oculus Rift uses an Organic light-emitting diode (OLED) display panel with a resolution of 1920 x 1080. The emissive electroluminescent layer is a film of organic compound that emits light in response to an electric current. An OLED display works without a backlight; thus, it can display deep black levels and can be thinner and lighter than a liquid crystal display (LCD). In low ambient light conditions (such as a dark room), an OLED screen can achieve a higher contrast ratio than an LCD, regardless of whether the LCD uses cold cathode fluorescent lamps or an LED backlight. The HMD also contains the IMU, which is an electronic device that measures the linear and angular acceleration as described above. It also measure the magnetic field strength and direction, using a combination of sensors (accelerometers, gyroscopes and magnetometers) with an update rate of 1000 Hz to track the head movements of the wearer. The headset also has a built-in latency tester, which predicts the head position from velocity and acceleration data to ensure that there is a stable immersive experience for the user.
2. Tracking Camera – The external tracking camera is infra-red and should be positioned about 5 feet away from the HMD, it can be placed on top of the computer monitor or on a tripod since it also includes a standard tripod attachment. The camera creates a viewing frustum in which the Oculus runtime system tracks the position and translational movements of the HMD in 3 dimensional space. It picks up the blinking frequencies of an array of LED lights embedded in the faceplate of the HMD, where each LED broadcasts at specific flashing frequencies with various levels of brightness. This allows the tracking camera to identify the source based on the exact blinking patterns of the lights and therefore the HMDs exact position.
3. Headset Lenses – The Oculus Rift uses high quality lenses which magnify the display to allow for a wide field of view of 100 degrees. This distorts the image projected on the display creating a pincushion distortion which causes images to be pinched at their center. This is corrected by the Oculus SDK at render time by simulating the complementary barrel distortion which causes images to be spherised or "inflated" thereby negating the pincushion distortion. The separation of the lenses is adjustable by a dial on the bottom of the device, in order to accommodate a wide range of interpupillary distances. The same pair of lenses are used for all users, however there are multiple facial interfaces so that the user's eyes can be positioned at a different distance. This also allows for users wearing glasses to use the Oculus Rift, as well as users with widely varying facial shapes (Adapted from Wikipedia <https://en.wikipedia.org/wiki/Oculus_Rift>).
4. Computer Interface – The Rift uses an HDMI interface to display the video. The end of the headset cable has a cable connector box with two output cables, and one input connection. The two outputs are an HDMI connector and a USB connector, both of which must be plugged into the computer. The input is for the positional tracker sync cable - Insert one end of the sync cable to the top connector on the right side of the positional tracker with the positional tracker facing you. Connect the other end of the sync cable to the cable connector box on the headset cable. This synchronizes the motion tracking data from the IMU in the headset and the external camera.

**How the Oculus Rift Works:**

The purpose of virtual reality is to construct an experience that simulates a user’s physical presence in another environment. The Oculus Rift accomplishes this by acting both as a specialized input device and a specialized output device.

As an input device, Oculus Rift uses a combination of several sensors to allow an application to query for the current orientation and position of the user’s head, relative to some fixed coordinate system such as the frustum created by the tracking camera. This is commonly referred to as the *head pose.* The head pose information allows an application to change its output in response to changes in head pose, indicating the direction of gaze and how its position in space.

As an output device, the Oculus Rift is a display that creates a sense of immersion and presence by attempting to more closely reproduce the sensation of looking at an environment as if the observer were actually there, compared to viewing it on a monitor. The Oculus Rift does this in three ways:

* By providing a much wider field of view than conventional displays
* By providing a different image to each eye
* By blocking out the real environment around the subject, which would otherwise serve as a contradiction to the rendered environment

On the Oculus Rift display, frames are generated that conform to this wide field of view and offer a distinct image to each eye. To generate a single frame of output, an individual image is rendered and distorted for each eye before rendering the image for the next eye. After rendering and distorting the images for both eyes, the resulting output frame is sent to the display.

**Rendering an Immersive view:**

The first way the Rift increases immersion is via head tracking, eliminating part of the necessary mental translation when interacting with a computer-generated environment. When looking to the left, there is no need to calculate how much to move a mouse or joystick. You simply look left. This is as much an instance of a natural user interface as VR. The Oculus Rift enables this interaction by integrating sensor hardware that detects spatial acceleration on three axes and rotation rates on three axes. The Oculus SDK provides methods for retrieving raw sensor messages and coalescing them into a single head pose.

The second way the Oculus Rift increases immersion is by rendering a view that accurately mimics the way vision works, with a wide field of view and different images presented to each eye. The Rift hardware offers a much wider field of view than a typical monitor. Even a 30-inch monitor will usually occupy only about 50 degrees of your field of view, depending on how close you sit to it. The Oculus Rift provides a view of over 90 degrees vertically and more than 100 degrees horizontally. The Oculus Rift achieves this viewing range through the placement of the display and through the use of special lenses. Inside the Oculus Rift is a small high-resolution OLED display, and wearing the device places the display directly in front of the user at a distance of about 4cm. This alone makes the panel occupy a substantial field of view, although it is far too close to allow a user to easily bring the display into focus. Between the user’s eyes and the display panel are lenses designed to distort light in such a way as to both make the apparent focal depth infinitely far away (resolving the focus issue) and to make the panel appear much wider than it is, further increasing the field of view. The lenses are designed to present roughly the same image to the user whether or not the user’s point of view is off the axis of the lens, using collimated light. The effect is similar to a magnifying glass.

**Rendering to the Rift:**

The Oculus Rift requires split-screen stereo with half of the screen used for each eye, with distortion correction for each eye to cancel the previously mentioned lens-related distortion.

When using the Rift, the left eye sees the left half of the screen, and the right eye sees the right half. Although varying from person-to-person, human eye pupils are approximately 65 mm apart. This is known as interpupillary distance (IPD). The in-application cameras should be configured with the same separation.

**Note:** This is a translation of the camera, not a rotation, and it is this translation (and the parallax effect that goes with it) that causes the stereoscopic effect. This means that your application will need to render the entire scene twice, once with the left virtual camera, and once with the right.

Correcting for distortion can be challenging, with distortion parameters varying for different lens types and individual eye relief. To make development easier, Oculus SDK handles distortion correction automatically within the Oculus Compositor process; it also takes care of latency-reducing timewarp (head pose prediction) and presents frames to the headset.

With Oculus SDK doing a lot of the work, the main job of the application is to perform simulation and render stereo world based on the tracking pose. Stereo views can be rendered into either one or two individual textures and are submitted to the compositor by calling ovr\_SubmitFrame.

## Rendering Setup Outline:

The Oculus SDK makes use of a compositor process to present frames and handle distortion.

To target the Rift, you render the scene into one or two render textures, passing these textures into the API. The Oculus runtime handles distortion rendering, GPU synchronization, frame timing, and frame presentation to the HMD.

The following is an outline of the steps required for SDK rendering taken directly from the Oculus Rift web site:

<https://developer.oculus.com/documentation/pcsdk/latest/concepts/dg-render/>

1. Initialize:
   1. Initialize Oculus SDK and create an ovrSession object for the headset.
   2. Compute the desired FOV and texture sizes based on ovrHmdDesc data.
   3. Allocate ovrSwapTextureSet objects, used to represent eye buffers, in an API-specific way: ovr\_CreateSwapTextureSetGL for OpenGL.
2. Set up frame handling:
   1. Use ovr\_GetTrackingState and ovr\_CalcEyePoses to compute eye poses needed for view rendering based on frame timing information.
   2. Perform rendering for each eye in an engine-specific way, rendering into the current texture within the texture set. Current texture is identified by the ovrSwapTextureSet::CurrentIndexvariable.
   3. Call ovr\_SubmitFrame, passing swap texture set(s) from the previous step within aovrLayerEyeFov structure. Although a single layer is required to submit a frame, you can use multiple layers and layer types for advanced rendering. ovr\_SubmitFrame passes layer textures to the compositor which handles distortion, timewarp, and GPU synchronization before presenting it to the headset.
   4. Advance CurrentIndex within each used texture set to target the next consecutive texture buffer for the following frame.
3. Shutdown:
   1. Call ovr\_DestroySwapTextureSet to destroy swap texture buffers. Call ovr\_DestroyMirrorTexture to destroy a mirror texture. To destroy the ovrSession object, call ovr\_Destroy.

**Development Environment**:

Oculus provides prebuilt solutions included in the Samples Folder of the SDK that is available for download from Oculus:

<https://developer.oculus.com/downloads/>

Select the PC platform and the 0.7.0.0-beta version.

The environment that was used for developing programs for the Oculus Rift was Microsoft Visual Studio Community 2013. This is a free version of Microsoft Visual Studio and can be downloaded at:

<https://www.visualstudio.com/en-us/news/vs2013-community-vs.aspx>

Once installed, the prebuilt solutions (such as Oculusroomtiny (GL) ) for running an Oculus Rift application can be loaded into the Microsoft Visual Studio IDE.

The following process shows how to create a new project for Rift development using the Oculus SDK. This includes creating a new project in Microsoft Visual Studio and adding the required dependencies to the project, such as the include directories and Oculus libraries.

Microsoft Visual Studio 2013 opens to a start page with an option to create a new project. Shown in Fig 0.1.

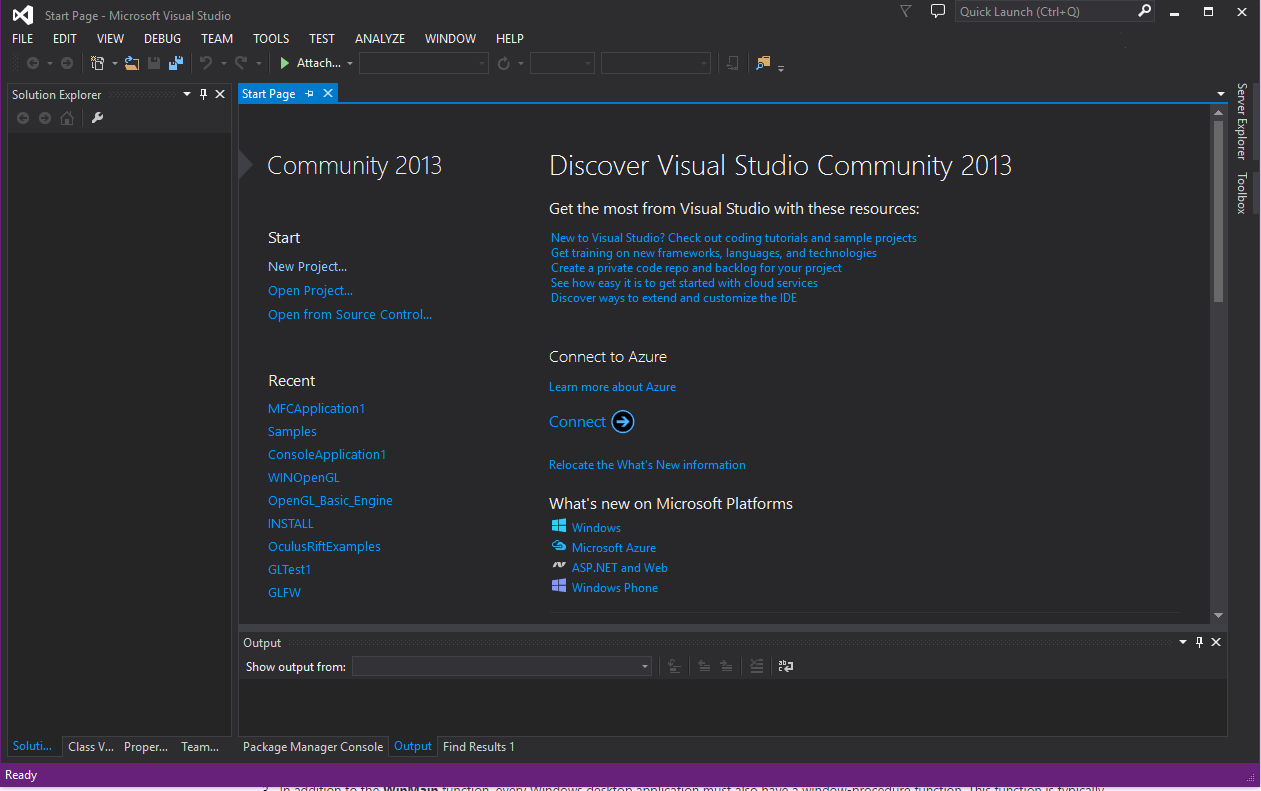


Fig 0.1: Microsoft Visual Studio 2013 Start page

Clicking “New Project” opens a new window with options to select the type of project to create. Fig 0.2

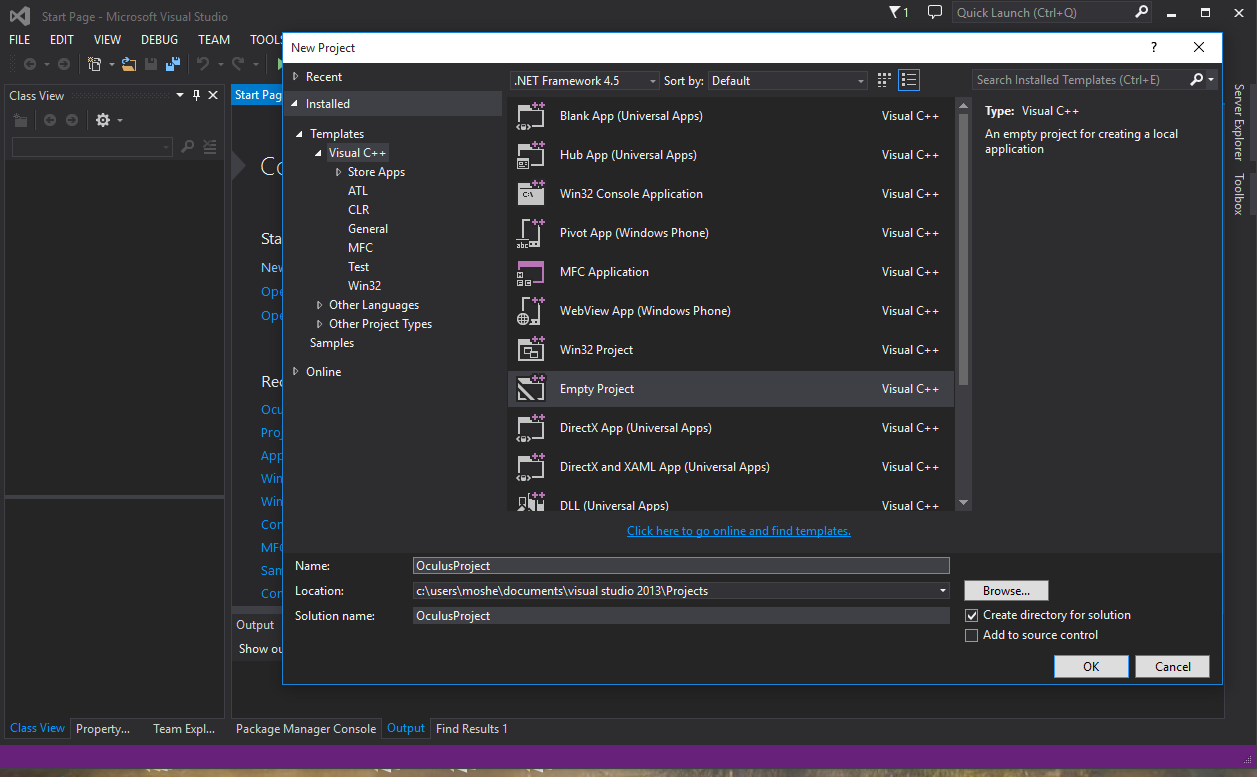


Fig 0.2: Project type selection

Select “Empty Project” and give it the desired name then press OK. This creates an empty solution (project) which is ready to be configured to work with the Oculus SDK.

In order to add the required dependencies, right-click on the solution name in the “Solution Explorer” tab on the left hand side of the window. This opens a context menu containing options to modify the project. Selecting the “properties” option opens a new window with the project’s configuration properties.

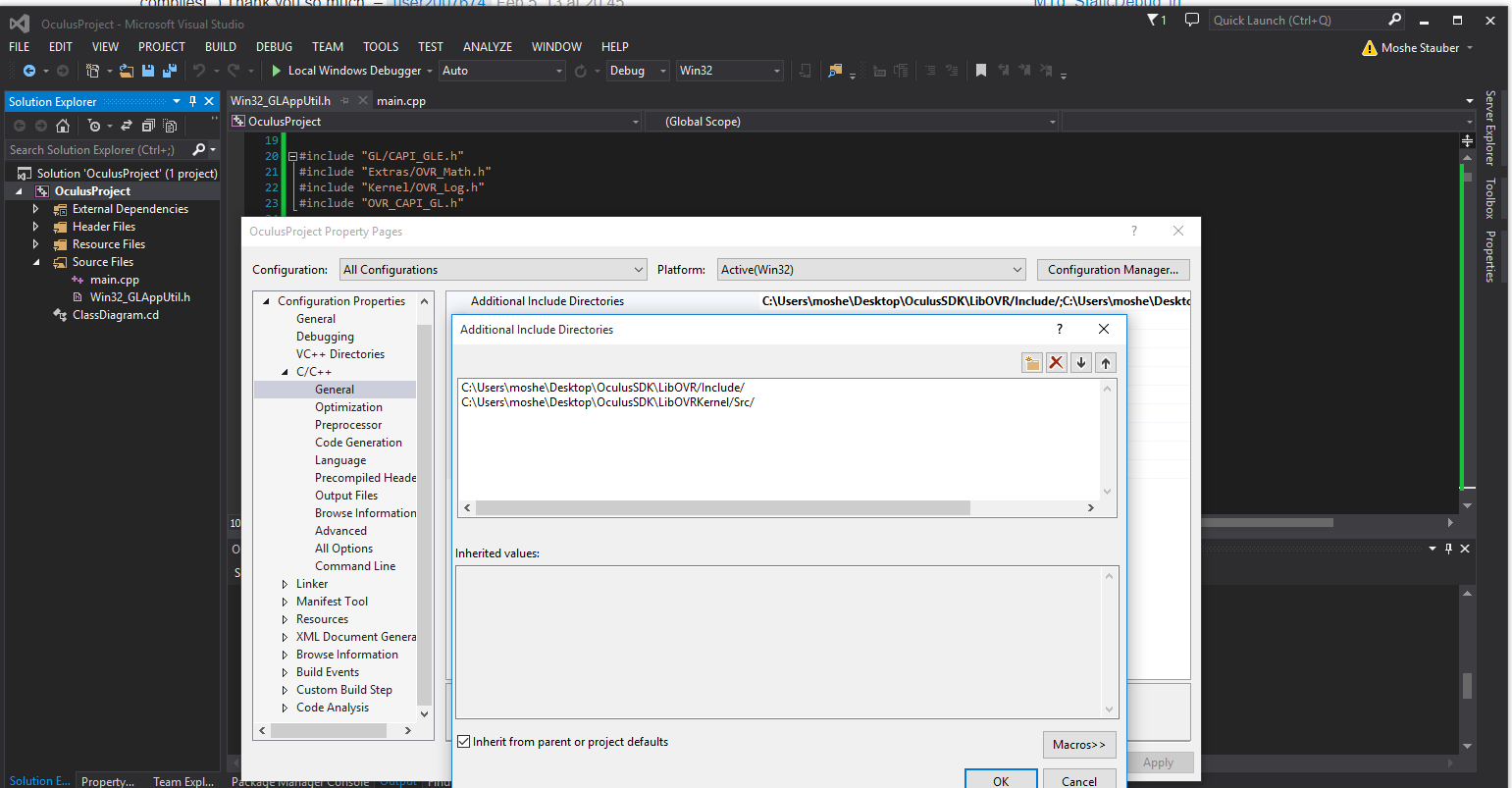
The Oculus Header and Source files required for this projects are **“OculusSDK/LibOVR/include/”** and **“OculusSDK/LibOVRKernel/Src/”** found in the Oculus SDK folder previously downloaded. To add these directories to the project, select “Configuration Properties”, then “C/C++”, “General”, and add these paths to the text field associated with “Additional Include Directories”. Fig 0.3

Fig 0.3: Adding Include Directories

Next, the Oculus libraries must be included in the project. These are **“LibOVR.lib”** and “**LibOVRKernel.lib”** which are found in the same Oculus SDK file downloaded. Be sure to use the libraries specific to the version of Visual Studio you are using. In the same configuration properties window, select “Linker”, then “Input”, and in the text field associated with “Additional Dependencies” add the path to these libraries. Fig 0.4

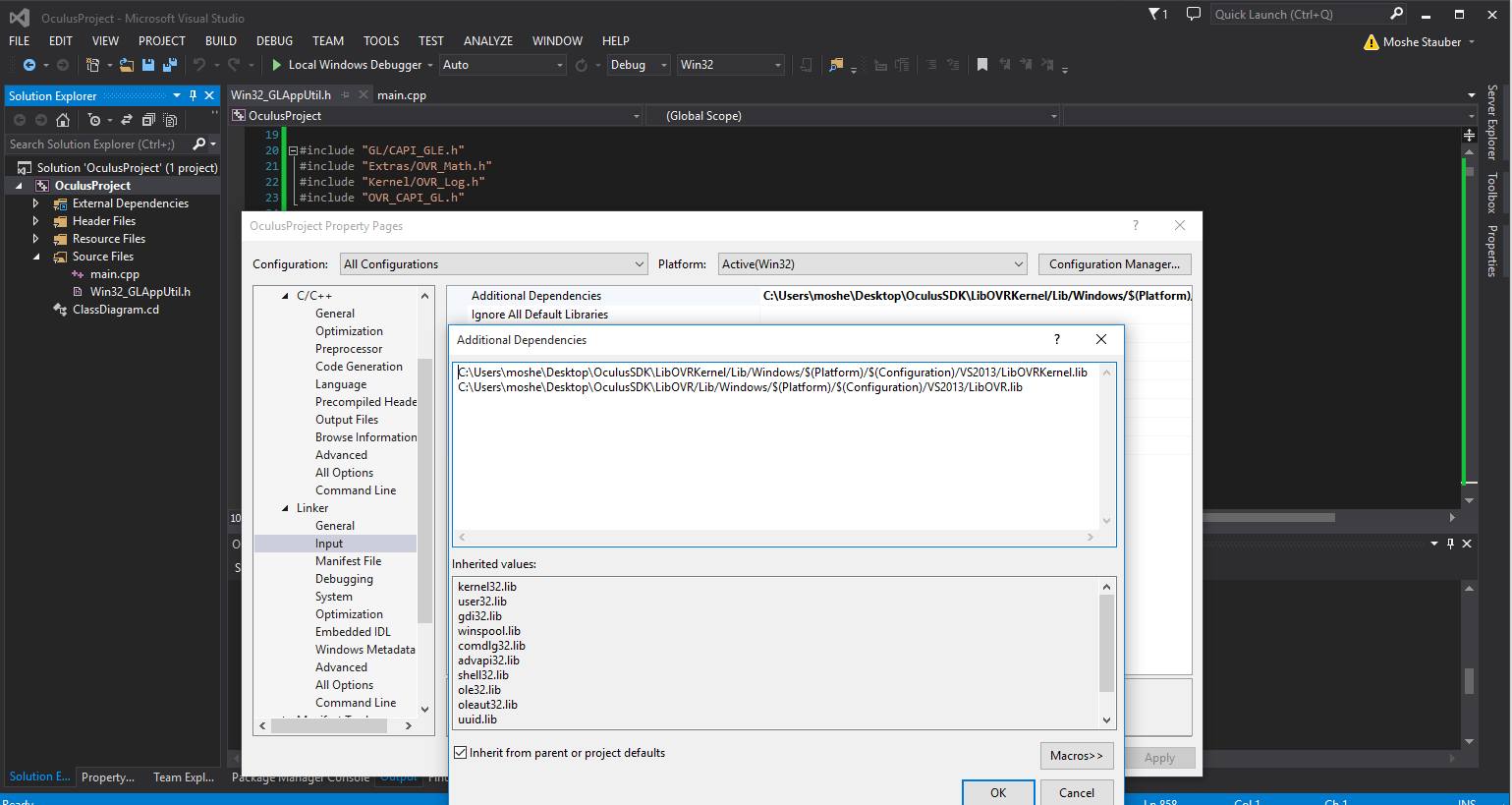


Fig 0.4: Linking Libraries

Visual Studio will automatically import the libraries which the Oculus Libraries depend on.

The entry point to the application consists of initializing the Oculus SDK and OpenGL context, entering the main rendering loop, and shutting down the Oculus SDK upon termination of the program. This is shown in Fig. 1.

*Platform* is a static C structure describing the global OpenGL state required for rendering with OpenGL. *Platform* initializes a window to be used for rendering with flags specific to OpenGL such as the colorDepth, doubleBuffering, as well as generating the frameBufferObjects for OpenGL to use in rendering. *Platform.Run()* takes as an argument a function pointer to the *MainLoop,* and runs *MainLoop* continuously while handling messages from the Oculus Runtime relaying information about the HMD.

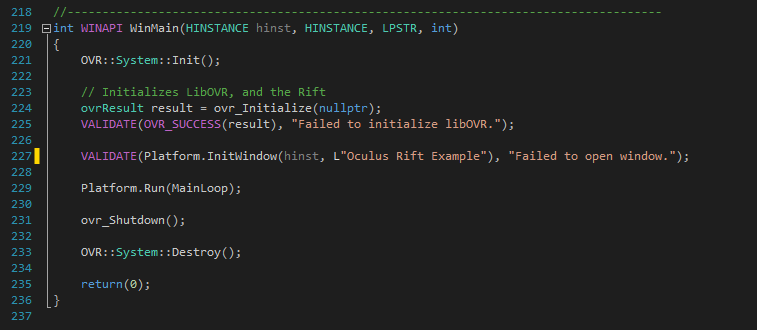


Fig 1. Main Application Entry Point, main.cpp shown in the side bar

*MainLoop* is also in main.cpp. This is where the OpenGL rendering is managed and the Oculus SDK calls are placed. A part of MainLoop (Fig. 2) shows the main structures that will be used throughout for the rendering, together with comments describing their purpose.

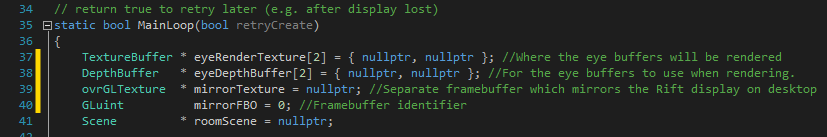
**

Fig. 2

The first task accomplished by MainLoop is to initializes the final rendering targets TextureBuffer\**eyeRenderTexture[2]*, as well as a frame buffer, which is of type *ovrGLTexture*\*mirrorTexture to mirror the Rift display on the desktop.It accomplishes this by using the Oculus Structure *ovrHmdDesc* descriptor, gives rendering information such as the display resolution and the default field of view for each “eye”.

The TextureBuffer\**eyeRenderTexture[2]* contains a camera view representing how each eye sees the world. A *TextureBuffer* contain a *TextureSet*, which is an Oculus structure called *ovrSwapTextureSet,* describes a set of textures that act as a rendered flip chain, the number of textures, and the index of the current texture to be submitted to the Oculus SDK for distortion-corrected rendering through *ovr\_SubmitFrame()* later on. A flip chain is an array of *ovrTexture*s and acts as an ordered sequence of images that create the appearance of visual motion. The optimal flip chain rate or frame rate has been estimated by Oculus to be about 75 frames per second to avoid motion sickness.

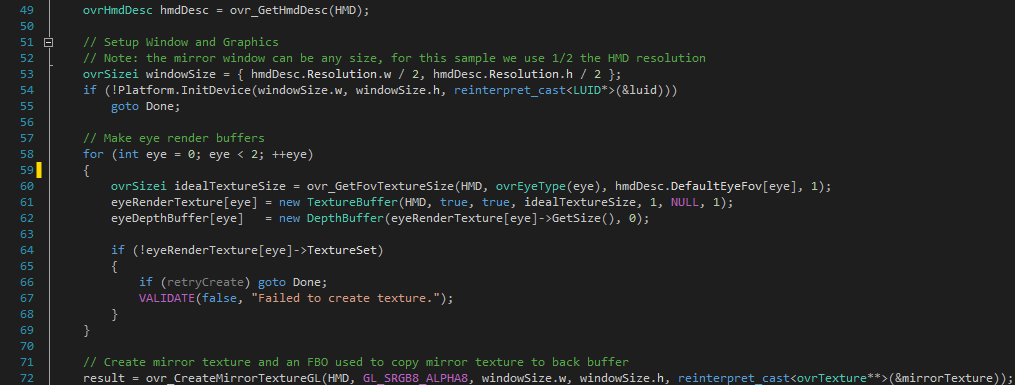


Fig. 2.

There is a separate OpenGL rendering call for each eye, based on the unique position and viewing angle of the eye. The eye pose is obtained by using the Oculus *ovrEyeRenderDesc* structure which contains rendering information for the eye such as the distortion viewport, view adjust, and other rendering parameters for the specified eye (Fig. 3).

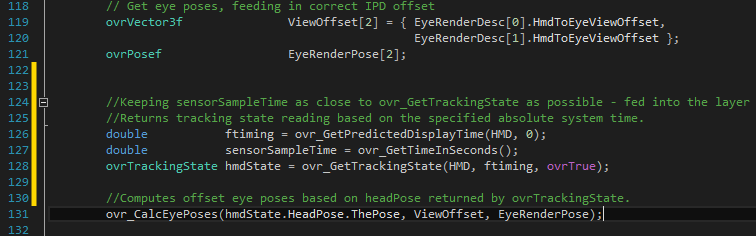


Fig. 3.

Eye pose information is used to create view and projection matrices unique to each eye. A For Loop is used to iterate over the two eyes, first setting a new texture and rendering target. We then calculate the projection matrix and view matrix using the eye pose information. Finally, we pass those matrices to our rendering process which renders a unique view of the scene for each eye stored in separate textures.

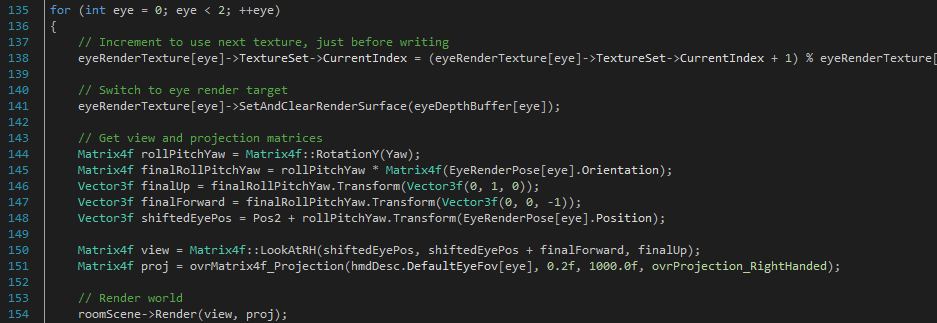


Fig. 4

The rendered textures are now passed to the Oculus SDK for distortion-corrected rendering by calling *ovr\_SubmitFrame()* (**Fig. 5**). The Oculus SDK function, *ovr\_SubmitFrame()*, uses a layering process to draw the final frame. A layer specifies a monoscopic or stereoscopic view. It consists of an *ovrLayerHeader* which specifies the kind of view it is, an *ovrSwapTextureSet* which is the set of textures (of both eyes) to be used, the viewport (a rectangular segment of the field), the specified field of view, and the position and orientation of each eye. The viewing scale to be used by the SDK for rendering is specified using Oculus *ovrViewScaleDesc* structure.

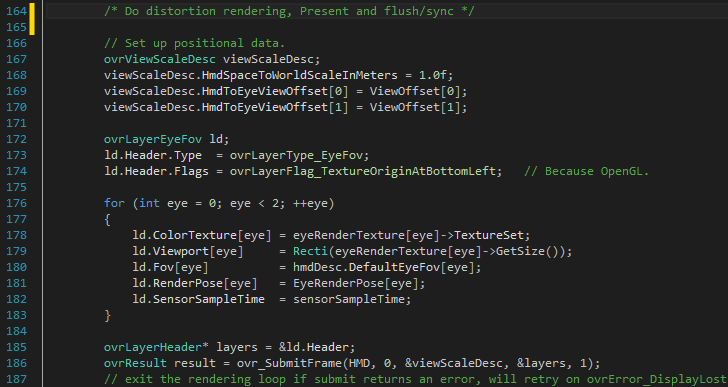


Fig. 5.

**Rendering Moving Stripes for Testing Motion Detection by Visual-Vestibular Interaction**:

In this section, we show the code below that renders a scene with four walls (lines 1-41). When the OpenGL Fragment shader is run (Lines 42-52). This colors the walls with stripes that can be made to move by sampling the texture and offsetting it by a given time. The complete program, which uses the Oculus libraries and renders the whole scene is given in the Appendix.

**void** Render(Matrix4f view, Matrix4f proj, GLfloat time)

1. {
2. Matrix4f combined = proj \* view \* GetMatrix();

5. glUseProgram(Fill->program);
6. glUniform1i(glGetUniformLocation(Fill->program, "Texture0"), 0);
7. glUniform1f(glGetAttribLocation(Fill->program, "Time"), (GLfloat)time);
8. glUniformMatrix4fv(glGetUniformLocation(Fill->program, "matWVP"), 1, GL\_TRUE, (**FLOAT**\*)&combined);
10. glActiveTexture(GL\_TEXTURE0);
11. glBindTexture(GL\_TEXTURE\_2D, Fill->texture->texId);
13. glBindBuffer(GL\_ARRAY\_BUFFER, vertexBuffer->buffer);
14. glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, indexBuffer->buffer);
16. GLuint posLoc = glGetAttribLocation(Fill->program, "Position");
17. GLuint colorLoc = glGetAttribLocation(Fill->program, "Color");
18. GLuint uvLoc = glGetAttribLocation(Fill->program, "TexCoord");
20. glEnableVertexAttribArray(posLoc);
21. glEnableVertexAttribArray(colorLoc);
22. glEnableVertexAttribArray(uvLoc);

25. glVertexAttribPointer(posLoc, 3, GL\_FLOAT, GL\_FALSE, **sizeof**(Vertex), (**void**\*)OVR\_OFFSETOF(Vertex, Pos));
26. glVertexAttribPointer(colorLoc, 4, GL\_UNSIGNED\_BYTE, GL\_TRUE, **sizeof**(Vertex), (**void**\*)OVR\_OFFSETOF(Vertex, C));
27. glVertexAttribPointer(uvLoc, 2, GL\_FLOAT, GL\_FALSE, **sizeof**(Vertex), (**void**\*)OVR\_OFFSETOF(Vertex, U));
29. glDrawElements(GL\_TRIANGLES, numIndices, GL\_UNSIGNED\_SHORT, NULL);
31. glDisableVertexAttribArray(posLoc);
32. glDisableVertexAttribArray(colorLoc);
33. glDisableVertexAttribArray(uvLoc);
35. glBindBuffer(GL\_ARRAY\_BUFFER, 0);
36. glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, 0);
38. glUseProgram(0);
40. }
41. **static** **const** GLchar\* FragmentShaderSrc =
42. "#version 450\n"
43. "uniform sampler2D Texture0;\n"
45. "in      vec4      oColor;\n"
46. "in      vec2      oTexCoord;\n"
47. "out     vec4      FragColor;\n"
49. "void main()\n"
50. "{\n"
51. "   FragColor = oColor \* texture2D(Texture0,oTexCoord);\n"
52. "}\n";

Appendix:

Main.cpp

1. #include "Win32\_GLAppUtil.h"
2. #include <Kernel/OVR\_System.h>
4. // Include the Oculus SDK
5. #include "OVR\_CAPI\_GL.h"
7. **using** **namespace** OVR;

10. // return true to retry later (e.g. after display lost)
11. **static** **bool** MainLoop(**bool** retryCreate)
12. {
13. TextureBuffer \* eyeRenderTexture[2] = { nullptr, nullptr };
14. DepthBuffer   \* eyeDepthBuffer[2] = { nullptr, nullptr };
15. ovrGLTexture  \* mirrorTexture = nullptr;
16. GLuint          mirrorFBO = 0;
17. Scene         \* roomScene = nullptr;
19. ovrHmd HMD;
20. ovrGraphicsLuid luid;
21. ovrResult result = ovr\_Create(&HMD, &luid);
22. **if** (!OVR\_SUCCESS(result))
23. **return** retryCreate;
25. ovrHmdDesc hmdDesc = ovr\_GetHmdDesc(HMD);
27. // Setup Window and Graphics
28. // Note: the mirror window can be any size, for this sample we use 1/2 the HMD resolution
29. ovrSizei windowSize = { hmdDesc.Resolution.w / 2, hmdDesc.Resolution.h / 2 };
30. **if** (!Platform.InitDevice(windowSize.w, windowSize.h, **reinterpret\_cast**<LUID\*>(&luid)))
31. **goto** Done;
33. // Start the sensor which informs of the Rift's pose and motion
34. result = ovr\_ConfigureTracking(HMD, ovrTrackingCap\_Orientation | ovrTrackingCap\_MagYawCorrection | ovrTrackingCap\_Position, 0);
35. **if** (!OVR\_SUCCESS(result))
36. {
37. **if** (retryCreate) **goto** Done;
38. VALIDATE(OVR\_SUCCESS(result), "Failed to configure tracking.");
39. }
41. // Make eye render buffers
42. **for** (**int** eye = 0; eye < 2; ++eye)
43. {
44. ovrSizei idealTextureSize = ovr\_GetFovTextureSize(HMD, ovrEyeType(eye), hmdDesc.DefaultEyeFov[eye], 1);
45. eyeRenderTexture[eye] = **new** TextureBuffer(HMD, **true**, **true**, idealTextureSize, 1, NULL, 1);
46. eyeDepthBuffer[eye] = **new** DepthBuffer(eyeRenderTexture[eye]->GetSize(), 0);
48. **if** (!eyeRenderTexture[eye]->TextureSet)
49. {
50. **if** (retryCreate) **goto** Done;
51. VALIDATE(**false**, "Failed to create texture.");
52. }
53. }
55. // Create mirror texture and an FBO used to copy mirror texture to back buffer
56. result = ovr\_CreateMirrorTextureGL(HMD, GL\_SRGB8\_ALPHA8, windowSize.w, windowSize.h, **reinterpret\_cast**<ovrTexture\*\*>(&mirrorTexture));
57. **if** (!OVR\_SUCCESS(result))
58. {
59. **if** (retryCreate) **goto** Done;
60. VALIDATE(**false**, "Failed to create mirror texture.");
61. }
63. // Configure the mirror read buffer
64. glGenFramebuffers(1, &mirrorFBO);
65. glBindFramebuffer(GL\_READ\_FRAMEBUFFER, mirrorFBO);
66. glFramebufferTexture2D(GL\_READ\_FRAMEBUFFER, GL\_COLOR\_ATTACHMENT0, GL\_TEXTURE\_2D, mirrorTexture->OGL.TexId, 0);
67. glFramebufferRenderbuffer(GL\_READ\_FRAMEBUFFER, GL\_DEPTH\_ATTACHMENT, GL\_RENDERBUFFER, 0);
68. glBindFramebuffer(GL\_READ\_FRAMEBUFFER, 0);
70. ovrEyeRenderDesc EyeRenderDesc[2];
71. EyeRenderDesc[0] = ovr\_GetRenderDesc(HMD, ovrEye\_Left, hmdDesc.DefaultEyeFov[0]);
72. EyeRenderDesc[1] = ovr\_GetRenderDesc(HMD, ovrEye\_Right, hmdDesc.DefaultEyeFov[1]);
74. // Turn off vsync to let the compositor do its magic
75. wglSwapIntervalEXT(0);
77. // Make scene - can simplify further if needed
78. roomScene = **new** Scene(**false**);
80. **bool** isVisible = **true**;
82. // Animate the texture
83. **static** **float** time = 0.0;
84. // Main loop
85. **while** (Platform.HandleMessages())
86. {
87. // Keyboard inputs to adjust player orientation
88. **static** **float** Yaw(3.141592f);
89. **if** (Platform.Key[VK\_LEFT])  Yaw += 0.02f;
90. **if** (Platform.Key[VK\_RIGHT]) Yaw -= 0.02f;
92. // Keyboard inputs to adjust player position
93. **static** Vector3f Pos2(0.0f, 1.6f, -5.0f);
94. **if** (Platform.Key['W'] || Platform.Key[VK\_UP])     Pos2 += Matrix4f::RotationY(Yaw).Transform(Vector3f(0, 0, -0.05f));
95. **if** (Platform.Key['S'] || Platform.Key[VK\_DOWN])   Pos2 += Matrix4f::RotationY(Yaw).Transform(Vector3f(0, 0, +0.05f));
96. **if** (Platform.Key['D'])                          Pos2 += Matrix4f::RotationY(Yaw).Transform(Vector3f(+0.05f, 0, 0));
97. **if** (Platform.Key['A'])                          Pos2 += Matrix4f::RotationY(Yaw).Transform(Vector3f(-0.05f, 0, 0));
98. Pos2.y = ovr\_GetFloat(HMD, OVR\_KEY\_EYE\_HEIGHT, Pos2.y);

101. //roomScene->Models[0]->Pos = Vector3f((cubeClock+=0.015f), 0, (cubeClock += 0.015f));
103. // Get eye poses, feeding in correct IPD offset
104. ovrVector3f               ViewOffset[2] = { EyeRenderDesc[0].HmdToEyeViewOffset,
105. EyeRenderDesc[1].HmdToEyeViewOffset };
106. ovrPosef                  EyeRenderPose[2];
108. ovrFrameTiming   ftiming = ovr\_GetFrameTiming(HMD, 0);
109. ovrTrackingState hmdState = ovr\_GetTrackingState(HMD, ftiming.DisplayMidpointSeconds);
110. ovr\_CalcEyePoses(hmdState.HeadPose.ThePose, ViewOffset, EyeRenderPose);
112. **if** (time > 1.0) time = 0.0;
113. time += 0.01f;
115. **if** (isVisible)
116. {
117. **for** (**int** eye = 0; eye < 2; ++eye)
118. {
119. // Increment to use next texture, just before writing
120. eyeRenderTexture[eye]->TextureSet->CurrentIndex = (eyeRenderTexture[eye]->TextureSet->CurrentIndex + 1) % eyeRenderTexture[eye]->TextureSet->TextureCount;
122. // Switch to eye render target
123. eyeRenderTexture[eye]->SetAndClearRenderSurface(eyeDepthBuffer[eye]);
125. // Get view and projection matrices
126. Matrix4f rollPitchYaw = Matrix4f::RotationY(Yaw);
127. Matrix4f finalRollPitchYaw = rollPitchYaw \* Matrix4f(EyeRenderPose[eye].Orientation);
128. Vector3f finalUp = finalRollPitchYaw.Transform(Vector3f(0, 1, 0));
129. Vector3f finalForward = finalRollPitchYaw.Transform(Vector3f(0, 0, -1));
130. Vector3f shiftedEyePos = Pos2 + rollPitchYaw.Transform(EyeRenderPose[eye].Position);
132. Matrix4f view = Matrix4f::LookAtRH(shiftedEyePos, shiftedEyePos + finalForward, finalUp);
133. Matrix4f proj = ovrMatrix4f\_Projection(hmdDesc.DefaultEyeFov[eye], 0.2f, 1000.0f, ovrProjection\_RightHanded);
135. // Render world
136. roomScene->Render(view, proj, time);
138. // Avoids an error when calling SetAndClearRenderSurface during next iteration.
139. eyeRenderTexture[eye]->UnsetRenderSurface();
140. }
141. }
143. // Do distortion rendering, Present and flush/sync
145. // Set up positional data.
146. ovrViewScaleDesc viewScaleDesc;
147. viewScaleDesc.HmdSpaceToWorldScaleInMeters = 1.0f;
148. viewScaleDesc.HmdToEyeViewOffset[0] = ViewOffset[0];
149. viewScaleDesc.HmdToEyeViewOffset[1] = ViewOffset[1];
151. ovrLayerEyeFov ld;
152. ld.Header.Type = ovrLayerType\_EyeFov;
153. ld.Header.Flags = ovrLayerFlag\_TextureOriginAtBottomLeft;   // Because OpenGL.
155. **for** (**int** eye = 0; eye < 2; ++eye)
156. {
157. ld.ColorTexture[eye] = eyeRenderTexture[eye]->TextureSet;
158. ld.Viewport[eye] = Recti(eyeRenderTexture[eye]->GetSize());
159. ld.Fov[eye] = hmdDesc.DefaultEyeFov[eye];
160. ld.RenderPose[eye] = EyeRenderPose[eye];
161. }
163. ovrLayerHeader\* layers = &ld.Header;
164. ovrResult result = ovr\_SubmitFrame(HMD, 0, &viewScaleDesc, &layers, 1);
165. // exit the rendering loop if submit returns an error, will retry on ovrError\_DisplayLost
166. **if** (!OVR\_SUCCESS(result))
167. **goto** Done;
169. isVisible = (result == ovrSuccess);
171. // Blit mirror texture to back buffer
172. glBindFramebuffer(GL\_READ\_FRAMEBUFFER, mirrorFBO);
173. glBindFramebuffer(GL\_DRAW\_FRAMEBUFFER, 0);
174. GLint w = mirrorTexture->OGL.Header.TextureSize.w;
175. GLint h = mirrorTexture->OGL.Header.TextureSize.h;
176. glBlitFramebuffer(0, h, w, 0,
177. 0, 0, w, h,
178. GL\_COLOR\_BUFFER\_BIT, GL\_NEAREST);
179. glBindFramebuffer(GL\_READ\_FRAMEBUFFER, 0);
181. SwapBuffers(Platform.hDC);
182. }
184. Done:
185. **delete** roomScene;
186. **if** (mirrorFBO) glDeleteFramebuffers(1, &mirrorFBO);
187. **if** (mirrorTexture) ovr\_DestroyMirrorTexture(HMD, **reinterpret\_cast**<ovrTexture\*>(mirrorTexture));
188. **for** (**int** eye = 0; eye < 2; ++eye)
189. {
190. **delete** eyeRenderTexture[eye];
191. **delete** eyeDepthBuffer[eye];
192. }
193. Platform.ReleaseDevice();
194. ovr\_Destroy(HMD);
196. // Retry on ovrError\_DisplayLost
197. **return** retryCreate || OVR\_SUCCESS(result) || (result == ovrError\_DisplayLost);
198. }
200. //-------------------------------------------------------------------------------------
201. **int** WINAPI WinMain(**HINSTANCE** hinst, **HINSTANCE**, **LPSTR**, **int**)
202. {
203. OVR::System::Init();
205. // Initializes LibOVR, and the Rift
206. ovrResult result = ovr\_Initialize(nullptr);
207. VALIDATE(OVR\_SUCCESS(result), "Failed to initialize libOVR.");
209. VALIDATE(Platform.InitWindow(hinst, L"Oculus Room Tiny (GL)"), "Failed to open window.");
211. Platform.Run(MainLoop);
213. ovr\_Shutdown();
215. OVR::System::Destroy();
217. **return**(0);
218. }

Win32\_GLAppUtil.h

1. #include <GL/CAPI\_GLE.h>
2. #include <Extras/OVR\_Math.h>
3. #include <Kernel/OVR\_Log.h>
4. #include "OVR\_CAPI\_GL.h"
6. **using** **namespace** OVR;
8. #ifndef VALIDATE
9. #define VALIDATE(x, msg) if (!(x)) { MessageBoxA(NULL, (msg), "Oculus1", MB\_ICONERROR | MB\_OK); exit(-1); }
10. #endif
12. //---------------------------------------------------------------------------------------
13. **struct** DepthBuffer
14. {
15. GLuint        texId;
17. DepthBuffer(Sizei size, **int** sampleCount)
18. {
19. OVR\_ASSERT(sampleCount <= 1); // The code doesn't currently handle MSAA textures.
21. glGenTextures(1, &texId);
22. glBindTexture(GL\_TEXTURE\_2D, texId);
23. glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);
24. glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);
25. glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);
26. glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);
28. GLenum internalFormat = GL\_DEPTH\_COMPONENT24;
29. GLenum type = GL\_UNSIGNED\_INT;
30. **if** (GLE\_ARB\_depth\_buffer\_float)
31. {
32. internalFormat = GL\_DEPTH\_COMPONENT32F;
33. type = GL\_FLOAT;
34. }
36. glTexImage2D(GL\_TEXTURE\_2D, 0, internalFormat, size.w, size.h, 0, GL\_DEPTH\_COMPONENT, type, NULL);
37. }
38. ~DepthBuffer()
39. {
40. **if** (texId)
41. {
42. glDeleteTextures(1, &texId);
43. texId = 0;
44. }
45. }
46. };
48. //--------------------------------------------------------------------------
49. **struct** TextureBuffer
50. {
51. ovrHmd              hmd;
52. ovrSwapTextureSet\*  TextureSet;
53. GLuint              texId;
54. GLuint              fboId;
55. Sizei               texSize;
57. TextureBuffer(ovrHmd hmd, **bool** rendertarget, **bool** displayableOnHmd, Sizei size, **int** mipLevels, unsigned **char** \* data, **int** sampleCount) :
58. hmd(hmd),
59. TextureSet(nullptr),
60. texId(0),
61. fboId(0),
62. texSize(0, 0)
63. {
64. OVR\_ASSERT(sampleCount <= 1); // The code doesn't currently handle MSAA textures.
66. texSize = size;
68. **if** (displayableOnHmd)
69. {
70. // This texture isn't necessarily going to be a rendertarget, but it usually is.
71. OVR\_ASSERT(hmd); // No HMD? A little odd.
72. OVR\_ASSERT(sampleCount == 1); // ovr\_CreateSwapTextureSetD3D11 doesn't support MSAA.
74. ovrResult result = ovr\_CreateSwapTextureSetGL(hmd, GL\_SRGB8\_ALPHA8, size.w, size.h, &TextureSet);
76. **if** (OVR\_SUCCESS(result))
77. {
78. **for** (**int** i = 0; i < TextureSet->TextureCount; ++i)
79. {
80. ovrGLTexture\* tex = (ovrGLTexture\*)&TextureSet->Textures[i];
81. glBindTexture(GL\_TEXTURE\_2D, tex->OGL.TexId);
83. **if** (rendertarget)
84. {
85. glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);
86. glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);
87. glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);
88. glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);
89. }
90. **else**
91. {
92. glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);
93. glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);
94. glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);
95. glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);
96. }
97. }
98. }
99. }
100. **else**
101. {
102. glGenTextures(1, &texId);
103. glBindTexture(GL\_TEXTURE\_2D, texId);
105. **if** (rendertarget)
106. {
107. glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);
108. glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);
109. glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);
110. glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);
111. }
112. **else**
113. {
114. glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);
115. glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);
116. glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);
117. glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);
118. }
120. glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_SRGB8\_ALPHA8, texSize.w, texSize.h, 0, GL\_RGBA, GL\_UNSIGNED\_BYTE, data);
121. }
123. **if** (mipLevels > 1)
124. {
125. glGenerateMipmap(GL\_TEXTURE\_2D);
126. }
128. glGenFramebuffers(1, &fboId);
129. }
131. ~TextureBuffer()
132. {
133. **if** (TextureSet)
134. {
135. ovr\_DestroySwapTextureSet(hmd, TextureSet);
136. TextureSet = nullptr;
137. }
138. **if** (texId)
139. {
140. glDeleteTextures(1, &texId);
141. texId = 0;
142. }
143. **if** (fboId)
144. {
145. glDeleteFramebuffers(1, &fboId);
146. fboId = 0;
147. }
148. }
150. Sizei GetSize() **const**
151. {
152. **return** texSize;
153. }
155. **void** SetAndClearRenderSurface(DepthBuffer\* dbuffer)
156. {
157. auto tex = **reinterpret\_cast**<ovrGLTexture\*>(&TextureSet->Textures[TextureSet->CurrentIndex]);
159. glBindFramebuffer(GL\_FRAMEBUFFER, fboId);
160. glFramebufferTexture2D(GL\_FRAMEBUFFER, GL\_COLOR\_ATTACHMENT0, GL\_TEXTURE\_2D, tex->OGL.TexId, 0);
161. glFramebufferTexture2D(GL\_FRAMEBUFFER, GL\_DEPTH\_ATTACHMENT, GL\_TEXTURE\_2D, dbuffer->texId, 0);
163. glViewport(0, 0, texSize.w, texSize.h);
164. glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);
165. glEnable(GL\_FRAMEBUFFER\_SRGB);
166. }
168. **void** UnsetRenderSurface()
169. {
170. glBindFramebuffer(GL\_FRAMEBUFFER, fboId);
171. glFramebufferTexture2D(GL\_FRAMEBUFFER, GL\_COLOR\_ATTACHMENT0, GL\_TEXTURE\_2D, 0, 0);
172. glFramebufferTexture2D(GL\_FRAMEBUFFER, GL\_DEPTH\_ATTACHMENT, GL\_TEXTURE\_2D, 0, 0);
173. }
174. };
176. //-------------------------------------------------------------------------------------------
177. **struct** OGL
178. {
179. **static** **const** **bool**       UseDebugContext = **false**;
181. **HWND**                    Window;
182. **HDC**                     hDC;
183. HGLRC                   WglContext;
184. OVR::GLEContext         GLEContext;
185. **bool**                    Running;
186. **bool**                    Key[256];
187. **int**                     WinSizeW;
188. **int**                     WinSizeH;
189. GLuint                  fboId;
190. **HINSTANCE**               hInstance;
192. **static** **LRESULT** CALLBACK WindowProc(\_In\_ **HWND** hWnd, \_In\_ **UINT** Msg, \_In\_ **WPARAM** wParam, \_In\_ **LPARAM** lParam)
193. {
194. OGL \*p = **reinterpret\_cast**<OGL \*>(GetWindowLongPtr(hWnd, 0));
195. **switch** (Msg)
196. {
197. **case** WM\_KEYDOWN:
198. p->Key[wParam] = **true**;
199. **break**;
200. **case** WM\_KEYUP:
201. p->Key[wParam] = **false**;
202. **break**;
203. **case** WM\_DESTROY:
204. p->Running = **false**;
205. **break**;
206. **default**:
207. **return** DefWindowProcW(hWnd, Msg, wParam, lParam);
208. }
209. **if** ((p->Key['Q'] && p->Key[VK\_CONTROL]) || p->Key[VK\_ESCAPE])
210. {
211. p->Running = **false**;
212. }
213. **return** 0;
214. }
216. OGL() :
217. Window(nullptr),
218. hDC(nullptr),
219. WglContext(nullptr),
220. GLEContext(),
221. Running(**false**),
222. WinSizeW(0),
223. WinSizeH(0),
224. fboId(0),
225. hInstance(nullptr)
226. {
227. // Clear input
228. **for** (**int** i = 0; i < **sizeof**(Key) / **sizeof**(Key[0]); ++i)
229. Key[i] = **false**;
230. }
232. ~OGL()
233. {
234. ReleaseDevice();
235. CloseWindow();
236. }
238. **bool** InitWindow(**HINSTANCE** hInst, **LPCWSTR** title)
239. {
240. hInstance = hInst;
241. Running = **true**;
243. WNDCLASSW wc;
244. memset(&wc, 0, **sizeof**(wc));
245. wc.style = CS\_CLASSDC;
246. wc.lpfnWndProc = WindowProc;
247. wc.cbWndExtra = **sizeof**(**struct** OGL \*);
248. wc.hInstance = GetModuleHandleW(NULL);
249. wc.lpszClassName = L"ORT";
250. RegisterClassW(&wc);
252. // adjust the window size and show at InitDevice time
253. Window = CreateWindowW(wc.lpszClassName, title, WS\_OVERLAPPEDWINDOW, 0, 0, 0, 0, 0, 0, hInstance, 0);
254. **if** (!Window) **return** **false**;
256. SetWindowLongPtr(Window, 0, **LONG\_PTR**(**this**));
258. hDC = GetDC(Window);
260. **return** **true**;
261. }
263. **void** CloseWindow()
264. {
265. **if** (Window)
266. {
267. **if** (hDC)
268. {
269. ReleaseDC(Window, hDC);
270. hDC = nullptr;
271. }
272. DestroyWindow(Window);
273. Window = nullptr;
274. UnregisterClassW(L"OGL", hInstance);
275. }
276. }
278. // Note: currently there is no way to get GL to use the passed pLuid
279. **bool** InitDevice(**int** vpW, **int** vpH, **const** LUID\* /\*pLuid\*/, **bool** windowed = **true**)
280. {
281. WinSizeW = vpW;
282. WinSizeH = vpH;
284. RECT size = { 0, 0, vpW, vpH };
285. AdjustWindowRect(&size, WS\_OVERLAPPEDWINDOW, **false**);
286. **const** **UINT** flags = SWP\_NOMOVE | SWP\_NOZORDER | SWP\_SHOWWINDOW;
287. **if** (!SetWindowPos(Window, nullptr, 0, 0, size.right - size.left, size.bottom - size.top, flags))
288. **return** **false**;
290. PFNWGLCHOOSEPIXELFORMATARBPROC wglChoosePixelFormatARBFunc = nullptr;
291. PFNWGLCREATECONTEXTATTRIBSARBPROC wglCreateContextAttribsARBFunc = nullptr;
292. {
293. // First create a context for the purpose of getting access to wglChoosePixelFormatARB / wglCreateContextAttribsARB.
294. PIXELFORMATDESCRIPTOR pfd;
295. memset(&pfd, 0, **sizeof**(pfd));
296. pfd.nSize = **sizeof**(pfd);
297. pfd.nVersion = 1;
298. pfd.iPixelType = PFD\_TYPE\_RGBA;
299. pfd.dwFlags = PFD\_SUPPORT\_OPENGL | PFD\_DRAW\_TO\_WINDOW | PFD\_DOUBLEBUFFER;
300. pfd.cColorBits = 32;
301. pfd.cDepthBits = 16;
302. **int** pf = ChoosePixelFormat(hDC, &pfd);
303. VALIDATE(pf, "Failed to choose pixel format.");
305. VALIDATE(SetPixelFormat(hDC, pf, &pfd), "Failed to set pixel format.");
307. HGLRC context = wglCreateContext(hDC);
308. VALIDATE(context, "wglCreateContextfailed.");
309. VALIDATE(wglMakeCurrent(hDC, context), "wglMakeCurrent failed.");
311. wglChoosePixelFormatARBFunc = (PFNWGLCHOOSEPIXELFORMATARBPROC)wglGetProcAddress("wglChoosePixelFormatARB");
312. wglCreateContextAttribsARBFunc = (PFNWGLCREATECONTEXTATTRIBSARBPROC)wglGetProcAddress("wglCreateContextAttribsARB");
313. OVR\_ASSERT(wglChoosePixelFormatARBFunc && wglCreateContextAttribsARBFunc);
315. wglDeleteContext(context);
316. }
318. // Now create the real context that we will be using.
319. **int** iAttributes[] =
320. {
321. // WGL\_DRAW\_TO\_WINDOW\_ARB, GL\_TRUE,
322. WGL\_SUPPORT\_OPENGL\_ARB, GL\_TRUE,
323. WGL\_COLOR\_BITS\_ARB, 32,
324. WGL\_DEPTH\_BITS\_ARB, 16,
325. WGL\_DOUBLE\_BUFFER\_ARB, GL\_TRUE,
326. WGL\_FRAMEBUFFER\_SRGB\_CAPABLE\_ARB, GL\_TRUE,
327. 0, 0
328. };
330. **float** fAttributes[] = { 0, 0 };
331. **int**   pf = 0;
332. **UINT**  numFormats = 0;
334. VALIDATE(wglChoosePixelFormatARBFunc(hDC, iAttributes, fAttributes, 1, &pf, &numFormats),
335. "wglChoosePixelFormatARBFunc failed.");
337. PIXELFORMATDESCRIPTOR pfd;
338. memset(&pfd, 0, **sizeof**(pfd));
339. VALIDATE(SetPixelFormat(hDC, pf, &pfd), "SetPixelFormat failed.");
341. GLint attribs[16];
342. **int**   attribCount = 0;
343. **if** (UseDebugContext)
344. {
345. attribs[attribCount++] = WGL\_CONTEXT\_FLAGS\_ARB;
346. attribs[attribCount++] = WGL\_CONTEXT\_DEBUG\_BIT\_ARB;
347. }
349. attribs[attribCount] = 0;
351. WglContext = wglCreateContextAttribsARBFunc(hDC, 0, attribs);
352. VALIDATE(wglMakeCurrent(hDC, WglContext), "wglMakeCurrent failed.");
354. OVR::GLEContext::SetCurrentContext(&GLEContext);
355. GLEContext.Init();
357. glGenFramebuffers(1, &fboId);
359. glEnable(GL\_DEPTH\_TEST);
360. glFrontFace(GL\_CW);
361. glEnable(GL\_CULL\_FACE);
363. **if** (UseDebugContext && GLE\_ARB\_debug\_output)
364. {
365. glDebugMessageCallbackARB(DebugGLCallback, NULL);
366. **if** (glGetError())
367. {
368. OVR\_DEBUG\_LOG(("glDebugMessageCallbackARB failed."));
369. }
371. glEnable(GL\_DEBUG\_OUTPUT\_SYNCHRONOUS\_ARB);
373. // Explicitly disable notification severity output.
374. glDebugMessageControlARB(GL\_DEBUG\_SOURCE\_API, GL\_DONT\_CARE, GL\_DEBUG\_SEVERITY\_NOTIFICATION, 0, NULL, GL\_FALSE);
375. }
377. **return** **true**;
378. }
380. **bool** HandleMessages(**void**)
381. {
382. MSG msg;
383. **while** (PeekMessage(&msg, NULL, 0U, 0U, PM\_REMOVE))
384. {
385. TranslateMessage(&msg);
386. DispatchMessage(&msg);
387. }
388. **return** Running;
389. }
391. **void** Run(**bool**(\*MainLoop)(**bool** retryCreate))
392. {
393. // false => just fail on any error
394. VALIDATE(MainLoop(**false**), "Oculus Rift not detected.");
395. **while** (HandleMessages())
396. {
397. // true => we'll attempt to retry for ovrError\_DisplayLost
398. **if** (!MainLoop(**true**))
399. **break**;
400. // Sleep a bit before retrying to reduce CPU load while the HMD is disconnected
401. Sleep(10);
402. }
403. }
405. **void** ReleaseDevice()
406. {
407. **if** (fboId)
408. {
409. glDeleteFramebuffers(1, &fboId);
410. fboId = 0;
411. }
412. **if** (WglContext)
413. {
414. wglMakeCurrent(NULL, NULL);
415. wglDeleteContext(WglContext);
416. WglContext = nullptr;
417. }
418. GLEContext.Shutdown();
419. }
421. **static** **void** GLAPIENTRY DebugGLCallback(GLenum source, GLenum type, GLuint id, GLenum severity, GLsizei length, **const** GLchar\* message, **const** **void**\* userParam)
422. {
423. OVR\_DEBUG\_LOG(("Message from OpenGL: %s\n", message));
424. }
425. };
427. // Global OpenGL state
428. **static** OGL Platform;
430. //------------------------------------------------------------------------------
431. **struct** ShaderFill
432. {
433. GLuint            program;
434. TextureBuffer   \* texture;
436. ShaderFill(GLuint vertexShader, GLuint pixelShader, TextureBuffer\* \_texture)
437. {
438. texture = \_texture;
440. program = glCreateProgram();
442. glAttachShader(program, vertexShader);
443. glAttachShader(program, pixelShader);
445. glLinkProgram(program);
447. glDetachShader(program, vertexShader);
448. glDetachShader(program, pixelShader);
450. GLint r;
451. glGetProgramiv(program, GL\_LINK\_STATUS, &r);
452. **if** (!r)
453. {
454. GLchar msg[1024];
455. glGetProgramInfoLog(program, **sizeof**(msg), 0, msg);
456. OVR\_DEBUG\_LOG(("Linking shaders failed: %s\n", msg));
457. }
458. }
460. ~ShaderFill()
461. {
462. **if** (program)
463. {
464. glDeleteProgram(program);
465. program = 0;
466. }
467. **if** (texture)
468. {
469. **delete** texture;
470. texture = nullptr;
471. }
472. }
473. };
475. //----------------------------------------------------------------
476. **struct** VertexBuffer
477. {
478. GLuint    buffer;
480. VertexBuffer(**void**\* vertices, **size\_t** size)
481. {
482. glGenBuffers(1, &buffer);
483. glBindBuffer(GL\_ARRAY\_BUFFER, buffer);
484. glBufferData(GL\_ARRAY\_BUFFER, size, vertices, GL\_STATIC\_DRAW);
485. }
486. ~VertexBuffer()
487. {
488. **if** (buffer)
489. {
490. glDeleteBuffers(1, &buffer);
491. buffer = 0;
492. }
493. }
494. };
496. //----------------------------------------------------------------
497. **struct** IndexBuffer
498. {
499. GLuint    buffer;
501. IndexBuffer(**void**\* indices, **size\_t** size)
502. {
503. glGenBuffers(1, &buffer);
504. glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, buffer);
505. glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, size, indices, GL\_STATIC\_DRAW);
506. }
507. ~IndexBuffer()
508. {
509. **if** (buffer)
510. {
511. glDeleteBuffers(1, &buffer);
512. buffer = 0;
513. }
514. }
515. };
517. //---------------------------------------------------------------------------
518. **struct** Model
519. {
520. **struct** Vertex
521. {
522. Vector3f  Pos;
523. **DWORD**     C;
524. **float**     U, V;
525. };
527. Vector3f        Pos;
528. Quatf           Rot;
529. Matrix4f        Mat;
530. **int**             numVertices, numIndices;
531. Vertex          Vertices[2000]; // Note fixed maximum
532. GLushort        Indices[2000];
533. ShaderFill    \* Fill;
534. VertexBuffer  \* vertexBuffer;
535. IndexBuffer   \* indexBuffer;
537. Model(Vector3f pos, ShaderFill \* fill) :
538. numVertices(0),
539. numIndices(0),
540. Pos(pos),
541. Rot(),
542. Mat(),
543. Fill(fill),
544. vertexBuffer(nullptr),
545. indexBuffer(nullptr)
546. {}
548. ~Model()
549. {
550. FreeBuffers();
551. }
553. Matrix4f& GetMatrix()
554. {
555. Mat = Matrix4f(Rot);
556. Mat = Matrix4f::Translation(Pos) \* Mat;
557. **return** Mat;
558. }
560. **void** AddVertex(**const** Vertex& v) { Vertices[numVertices++] = v; }
561. **void** AddIndex(GLushort a) { Indices[numIndices++] = a; }
563. **void** AllocateBuffers()
564. {
565. vertexBuffer = **new** VertexBuffer(&Vertices[0], numVertices \* **sizeof**(Vertices[0]));
566. indexBuffer = **new** IndexBuffer(&Indices[0], numIndices \* **sizeof**(Indices[0]));
567. }
569. **void** FreeBuffers()
570. {
571. **delete** vertexBuffer; vertexBuffer = nullptr;
572. **delete** indexBuffer; indexBuffer = nullptr;
573. }
575. **void** AddSolidColorBox(**float** x1, **float** y1, **float** z1, **float** x2, **float** y2, **float** z2, **DWORD** c)
576. {
577. Vector3f Vert[][2] =
578. {
579. Vector3f(x1, y2, z1), Vector3f(z1, x1), Vector3f(x2, y2, z1), Vector3f(z1, x2),
580. Vector3f(x2, y2, z2), Vector3f(z2, x2), Vector3f(x1, y2, z2), Vector3f(z2, x1),
581. Vector3f(x1, y1, z1), Vector3f(z1, x1), Vector3f(x2, y1, z1), Vector3f(z1, x2),
582. Vector3f(x2, y1, z2), Vector3f(z2, x2), Vector3f(x1, y1, z2), Vector3f(z2, x1),
583. Vector3f(x1, y1, z2), Vector3f(z2, y1), Vector3f(x1, y1, z1), Vector3f(z1, y1),
584. Vector3f(x1, y2, z1), Vector3f(z1, y2), Vector3f(x1, y2, z2), Vector3f(z2, y2),
585. Vector3f(x2, y1, z2), Vector3f(z2, y1), Vector3f(x2, y1, z1), Vector3f(z1, y1),
586. Vector3f(x2, y2, z1), Vector3f(z1, y2), Vector3f(x2, y2, z2), Vector3f(z2, y2),
587. Vector3f(x1, y1, z1), Vector3f(x1, y1), Vector3f(x2, y1, z1), Vector3f(x2, y1),
588. Vector3f(x2, y2, z1), Vector3f(x2, y2), Vector3f(x1, y2, z1), Vector3f(x1, y2),
589. Vector3f(x1, y1, z2), Vector3f(x1, y1), Vector3f(x2, y1, z2), Vector3f(x2, y1),
590. Vector3f(x2, y2, z2), Vector3f(x2, y2), Vector3f(x1, y2, z2), Vector3f(x1, y2)
591. };
593. GLushort CubeIndices[] =
594. {
595. 0, 1, 3, 3, 1, 2,
596. 5, 4, 6, 6, 4, 7,
597. 8, 9, 11, 11, 9, 10,
598. 13, 12, 14, 14, 12, 15,
599. 16, 17, 19, 19, 17, 18,
600. 21, 20, 22, 22, 20, 23
601. };
603. **for** (**int** i = 0; i < **sizeof**(CubeIndices) / **sizeof**(CubeIndices[0]); ++i)
604. AddIndex(CubeIndices[i] + GLushort(numVertices));

607. // Generate a quad for each box face
608. **for** (**int** v = 0; v < 6 \* 4; v++)
609. {
610. // Make vertices, with some token lighting
611. Vertex vvv; vvv.Pos = Vert[v][0]; vvv.U = Vert[v][1].x; vvv.V = Vert[v][1].y;
612. **float** dist1 = (vvv.Pos - Vector3f(-2, 4, -2)).Length();
613. **float** dist2 = (vvv.Pos - Vector3f(3, 4, -3)).Length();
614. **float** dist3 = (vvv.Pos - Vector3f(-4, 3, 25)).Length();
615. **int**   bri = rand() % 160;
616. **float** B = ((c >> 16) & 0xff) \* (bri + 192.0f \* (0.65f + 8 / dist1 + 1 / dist2 + 4 / dist3)) / 255.0f;
617. **float** G = ((c >> 8) & 0xff) \* (bri + 192.0f \* (0.65f + 8 / dist1 + 1 / dist2 + 4 / dist3)) / 255.0f;
618. **float** R = ((c >> 0) & 0xff) \* (bri + 192.0f \* (0.65f + 8 / dist1 + 1 / dist2 + 4 / dist3)) / 255.0f;
619. vvv.C = (c & 0xff000000) +
620. ((R > 255 ? 255 : **DWORD**(R)) << 16) +
621. ((G > 255 ? 255 : **DWORD**(G)) << 8) +
622. (B > 255 ? 255 : **DWORD**(B));
623. AddVertex(vvv);
624. }
626. }
628. **void** Render(Matrix4f view, Matrix4f proj, GLfloat time)
629. {
630. Matrix4f combined = proj \* view \* GetMatrix();

633. glUseProgram(Fill->program);
634. glUniform1i(glGetUniformLocation(Fill->program, "Texture0"), 0);
635. glUniform1f(glGetAttribLocation(Fill->program, "Time"), (GLfloat)time);
636. glUniformMatrix4fv(glGetUniformLocation(Fill->program, "matWVP"), 1, GL\_TRUE, (**FLOAT**\*)&combined);
638. glActiveTexture(GL\_TEXTURE0);
639. glBindTexture(GL\_TEXTURE\_2D, Fill->texture->texId);
641. glBindBuffer(GL\_ARRAY\_BUFFER, vertexBuffer->buffer);
642. glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, indexBuffer->buffer);
644. GLuint posLoc = glGetAttribLocation(Fill->program, "Position");
645. GLuint colorLoc = glGetAttribLocation(Fill->program, "Color");
646. GLuint uvLoc = glGetAttribLocation(Fill->program, "TexCoord");
648. glEnableVertexAttribArray(posLoc);
649. glEnableVertexAttribArray(colorLoc);
650. glEnableVertexAttribArray(uvLoc);

653. glVertexAttribPointer(posLoc, 3, GL\_FLOAT, GL\_FALSE, **sizeof**(Vertex), (**void**\*)OVR\_OFFSETOF(Vertex, Pos));
654. glVertexAttribPointer(colorLoc, 4, GL\_UNSIGNED\_BYTE, GL\_TRUE, **sizeof**(Vertex), (**void**\*)OVR\_OFFSETOF(Vertex, C));
655. glVertexAttribPointer(uvLoc, 2, GL\_FLOAT, GL\_FALSE, **sizeof**(Vertex), (**void**\*)OVR\_OFFSETOF(Vertex, U));
657. glDrawElements(GL\_TRIANGLES, numIndices, GL\_UNSIGNED\_SHORT, NULL);
659. glDisableVertexAttribArray(posLoc);
660. glDisableVertexAttribArray(colorLoc);
661. glDisableVertexAttribArray(uvLoc);
663. glBindBuffer(GL\_ARRAY\_BUFFER, 0);
664. glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, 0);
666. glUseProgram(0);
668. }
669. };
671. //-------------------------------------------------------------------------
672. **struct** Scene
673. {
674. **int**     numModels;
675. Model \* Models[10];
677. **void**    Add(Model \* n)
678. {
679. Models[numModels++] = n;
680. }
682. **void** Render(Matrix4f view, Matrix4f proj, GLfloat time)
683. {
684. **for** (**int** i = 0; i < numModels; ++i)
685. Models[i]->Render(view, proj, time);
686. }
688. GLuint CreateShader(GLenum type, **const** GLchar\* src)
689. {
690. GLuint shader = glCreateShader(type);
692. glShaderSource(shader, 1, &src, NULL);
693. glCompileShader(shader);
695. GLint r;
696. glGetShaderiv(shader, GL\_COMPILE\_STATUS, &r);
697. **if** (!r)
698. {
699. GLchar msg[1024];
700. glGetShaderInfoLog(shader, **sizeof**(msg), 0, msg);
701. **if** (msg[0]) {
702. OVR\_DEBUG\_LOG(("Compiling shader failed: %s\n", msg));
703. }
704. **return** 0;
705. }
707. **return** shader;
708. }
710. **void** Init(**int** includeIntensiveGPUobject)
711. {
712. **static** **const** GLchar\* VertexShaderSrc =
713. "#version 450\n"
714. "uniform mat4 matWVP;\n"
715. "uniform float     Time;\n"
716. "in      vec4 Position;\n"
717. "in      vec4 Color;\n"
718. "in      vec2 TexCoord;\n"
719. "out     vec2 oTexCoord;\n"
720. "out     vec4 oColor;\n"
722. "void main()\n"
723. "{\n"
724. "   gl\_Position = (matWVP \* Position);\n"
725. "   oTexCoord   = vec2(TexCoord.x + Time, TexCoord.y);\n"
726. "   oColor.rgb  = pow(Color.rgb, vec3(2.2));\n"   // convert from sRGB to linear
727. "   oColor.a    = Color.a;\n"
728. "}\n";
730. **static** **const** GLchar\* FragmentShaderSrc =
731. "#version 450\n"
732. "uniform sampler2D Texture0;\n"
734. "in      vec4      oColor;\n"
735. "in      vec2      oTexCoord;\n"
736. "out     vec4      FragColor;\n"
738. "void main()\n"
739. "{\n"
740. "   FragColor = oColor \* texture2D(Texture0,oTexCoord);\n"
742. "}\n";
744. GLuint    vshader = CreateShader(GL\_VERTEX\_SHADER, VertexShaderSrc);
745. GLuint    fshader = CreateShader(GL\_FRAGMENT\_SHADER, FragmentShaderSrc);
747. // Make textures
748. ShaderFill \* grid\_material;
750. {
751. **const** **int** checkImageHeight = 16;
752. **const** **int** checkImageWidth = 16;
753. GLubyte checkImage[checkImageHeight][checkImageWidth][4];
755. **static** GLuint texName;
757. **int** i, j, c;
759. **for** (i = 0; i < checkImageHeight; i++) {
760. **for** (j = 0; j < checkImageWidth; j++) {
761. c = ( ( ( (j & 0x8) ) == 0) ) \* 255;
762. checkImage[i][j][0] = (GLubyte)c;
763. checkImage[i][j][1] = (GLubyte)c;
764. checkImage[i][j][2] = (GLubyte)c;
765. checkImage[i][j][3] = (GLubyte)255;
766. }
767. }
769. TextureBuffer \* generated\_texture = **new** TextureBuffer(nullptr, **false**, **false**, Sizei(checkImageWidth,checkImageHeight), 4, (unsigned **char** \*)checkImage, 1);
770. grid\_material = **new** ShaderFill(vshader, fshader, generated\_texture);
771. }

774. glDeleteShader(vshader);
775. glDeleteShader(fshader);
777. // Construct geometry
778. Model \* m;// = new Model(Vector3f(0, 0, 0), grid\_material);
780. m = **new** Model(Vector3f(0, 0, 0), grid\_material);  // Walls
781. m->AddSolidColorBox(-10.1f, 0.0f, -20.0f, -10.0f, 4.0f, 20.0f, 0xff808080); // Left Wall
782. m->AddSolidColorBox(-10.0f, -0.1f, -20.1f, 10.0f, 4.0f, -20.0f, 0xff808080); // Back Wall
783. m->AddSolidColorBox(10.0f, -0.1f, -20.0f, 10.1f, 4.0f, 20.0f, 0xff808080); // Right Wall
784. m->AddSolidColorBox(-10.0f, -0.1f, 20.0f, 10.0f, 4.0f, 20.0f, 0xff808080); // Front Wall
785. m->AllocateBuffers();
786. Add(m);
787. }
789. Scene() : numModels(0) {}
790. Scene(**bool** includeIntensiveGPUobject) :
791. numModels(0)
792. {
793. Init(includeIntensiveGPUobject);
794. }
795. **void** Release()
796. {
797. **while** (numModels-- > 0)
798. **delete** Models[numModels];
799. }
800. ~Scene()
801. {
802. Release();
803. }
804. };

The following References were utilized in preparing this paper:

Davis, Bradley Ausitin and Bryla, Karen and Benton, Phillips, Alexander Benton (2015) *Oculus Rift In Action,* Shelter Island, NY:Manning Publications

Terndrup, Matthew (2014), *Reverse Engineering the Oculus Rift DK2 Provides Brilliant Insight into Inner Workings,* Web. <http://www.roadtovr.com/reverse-engineering-oculus-rift-dk2-positional-tracking-camera-linux-sdk/>

Wikipedia Oculus Rift, https://en.wikipedia.org/wiki/Oculus\_Rift