Virtual Reality System Concepts Illustrated Using OSVR

Author: Russell M. Taylor II

The immersive nature of virtual and augmented reality systems engages the human visual system in ways that require wider field of view and lower latency than other 3D computer graphics systems to provide artifact-free rendering that avoids inducing fatigue and discomfort in viewers. The need to construct consistent transformations between multiple objects in the system (head, hands, and/or screens) requires a common space. The need to precisely match the viewing direction requires off-axis projection matrices that are carefully matched to the relative positions of the viewer's eyes and screens. System lens designs often induce chromatic aberration and nonlinear distortions of the screen images that depend on the location of the viewer's eyes with respect to the lenses and the location of the lenses with respect to the screens. The temporal sampling apertures of tracking systems and the finite times required to render and scan out the images, together with operating-system-induced delays, introduce latency between the viewer's head pose and the images being displayed at a given moment in time.

VR systems have developed a set of approaches to addressing these issues. The result of applying these concepts is a geometric rendering state that describes to the application how to render each eye. As the visual sense is the primary perceptual channel exploited by virtual reality systems, that is the focus of this chapter – getting the visual aspects correct is paramount.

This chapter introduces each of these issues, presenting both theoretical descriptions and example implementations in the Open-Source VR system (OSVR.org). OSVR is a universal open source VR ecosystem for technologies across different brands and companies pioneered and led by Yuval Boger with a core development team at Sensics including Ryan Pavlik, Jeremy Bell, Greg Aring, Georgiy Frolov, and Kevin Godby. Chapter author Russ Taylor provided consulting support including developing the *RenderManager* rendering kit that implements many of the functions described herein. Many others from the growing OSVR community have contributed to its development. The Apache 2 open-source and open-hardware licensing for OSVR makes it an effective base for building both research and commercial solutions.

Common Space

To enable proper rendering, the viewer's head (and sometimes eyes) are tracked. Because of the need to interact with the virtual world, their hands are also often tracked; indeed, to enable a feeling of presence, some systems track a whole-body skeleton. In head-mounted systems the screens are attached to the viewer's head, in CAVE-like systems they are located in the real world, and in projection-based systems they are projected onto objects located in the real world. This section describes the spaces needed to support viewing and interacting with the virtual world.

As seen in figure 32.1, VR systems often involve separate devices at different locations. The computational loads involved, the need for physical separation between cameras and the objects they are tracking, and the fact that different vendors provide different tracking systems means that often more than one device is used to perform tracking. An inertial measurement unit augmented by an external camera may be used to track the head while a depth camera is used to determine the whole-body skeleton. For screen-based systems, the screens are necessarily at different fixed locations in the room while the viewer's head is tracked separately. A force-feedback device may be used to track the hand and interact with the world while a camera-based system is used to track the viewer's head.

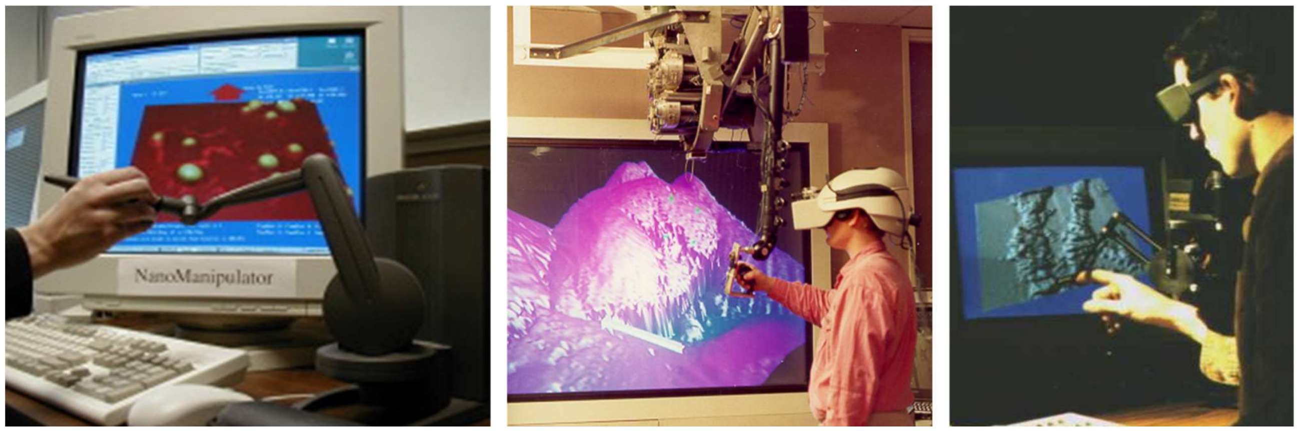


Figure 32.1: This shows three versions of the *Nanomanipulator* system that enabled a viewer to see and touch real atoms and molecules using a scanned-probe microscope [Taylor93]. The left image shows a non-VR mode of operation using standard 3D graphics and a force display pen where only the hand is tracked. The center image shows a magnetically-tracked head-mounted display system co-registered with a mechanical force-feedback hand-tracking system (the image seen in one eye is displayed on the screen for collaborators). The right image shows optically tracked glasses with the viewer looking at a stereo display and using a force-feedback hand-tracking system.

Consistent display and interaction requires determining the transformations between each tracked object in the same coordinate system. Figure 32.2 follows the model in [Robinett92], which calls this coordinate system *Room* space, a space that is rigidly attached to the physical room or vehicle where the viewer is located. *Room* space is connected to the *Virtual World* space in which the application object lives by a transformation that enables the entire VR system to move as a unit in the world. Depending on the system, displays live in either *Room* space (fishtank VR, CAVE-like systems) or *Head* space (head-mounted displays). The locations of various tracker bases live in room space and they measure sensors that are attached with an offset and rotation to head, hand, and any other tracked space. Because the application may want to draw things in these spaces (hand models, indication of camera location), the VR system must make its spaces available to the application.

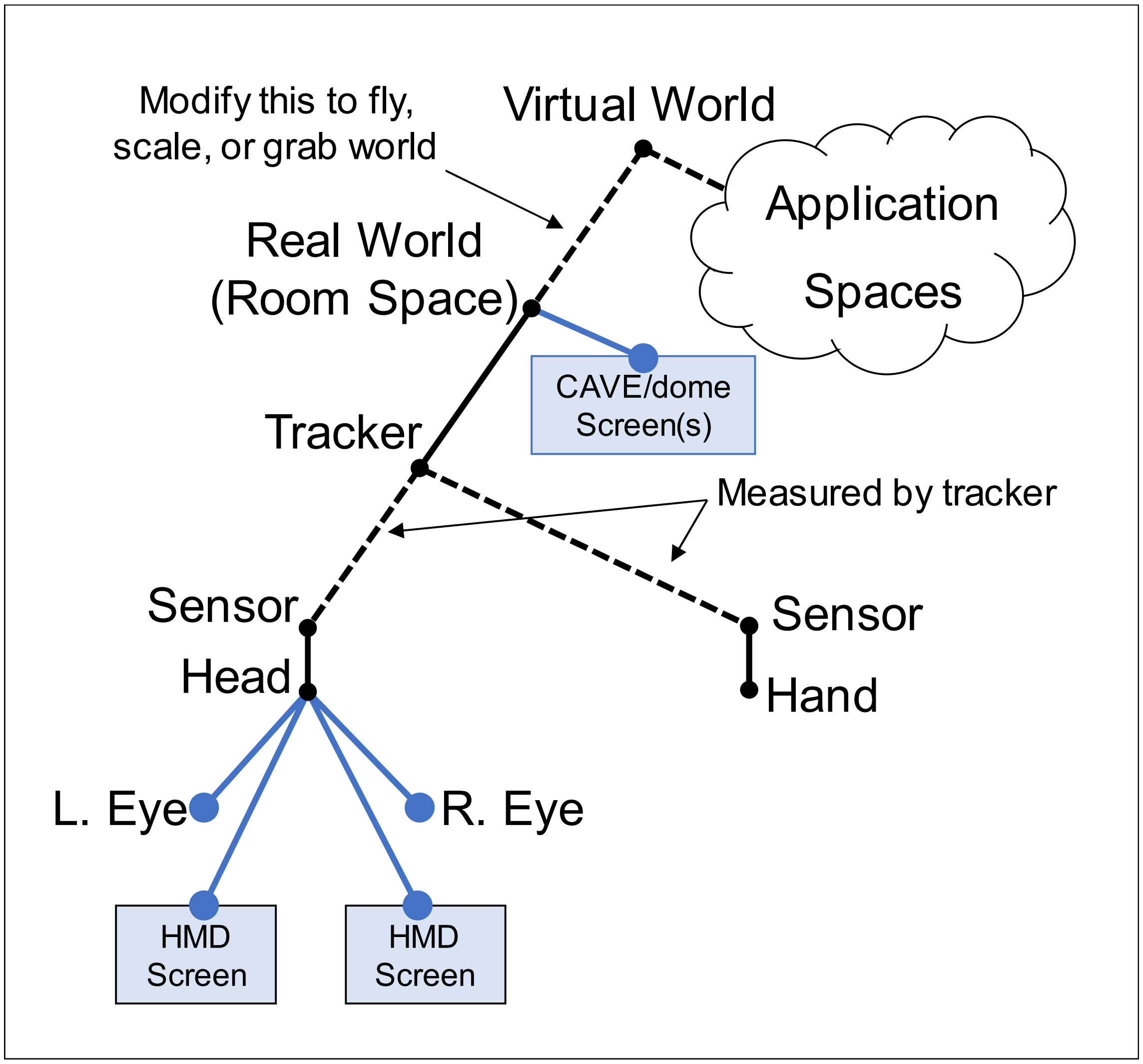


Figure 32.2: This VR space diagram, reworked from [Robinett92], shows the spaces managed by a VR system. Solid lines are determined during system calibration; dashed lines are updated during viewing (with the application managing its own set of spaces, which it can modify at will). The large-dot blue transforms (eyes and screens) are used together to determine the projection and viewing transformations. On some systems, the eyes may also be tracked, but the transformations described in this chapter ignore this tracking.

Implementation of spaces within OSVR

OSVR is separated into *server* and *client* portions, with servers managing devices and keeping track of semantic paths that provide meaningful names (for example, mapping /xbox/button/0 to /me/hands/left/fingerTrigger) and that describe how transformations are nested to provide VR-relevant transformations (for example, mapping /myExternalOculus/tracker/0 to /me/head).

OSVR is also separated into subsystems: *Core* and *RenderManager* are the two that will be referenced here. *Core* manages devices, transformations, configuration, and message passing. *RenderManager* implements advanced rendering techniques for a number of rendering systems (*OpenGL*, *Direct3D*, *GLES*).

OSVR_Core manages the common spaces within OSVR using one or more *osvr_server* processes. Client applications connect to these servers to find out about devices and events. OSVR uses *interfaces* to provide access to spaces and other devices, which are accessed by named paths as shown in Listing 32.1:

Listing 32.1: Getting an interface to head space.

```
OSVR_ClientContext m_context;
m_context = osvrClientInit("com.osvr.renderManager");
std::string headSpaceName = "/me/head";
OSVR_ClientInterface m_roomFromHeadInterface;
OSVR_PoseState m_roomFromHead; ///< Transform to use for head space
osvrClientGetInterface(m_context, headSpaceName.c_str(), &m_roomFromHeadInterface);
```

Once configured, the client context can be updated. Each call to update reads all messages from the server and updates the state of all interfaces. This state is only updated when requested, so that a consistent set of interface states can be used for rendering to all eyes. Once the context has been updated for a frame, the application can read the state of any interfaces along with the time at which the state was valid as shown in Listing 32.2:

Listing 32.2: Reading the head pose state.

```
osvrClientUpdate(m_context);
OSVR_TimeValue timestamp;
osvrGetPoseState(m_roomFromHeadInterface, &timestamp, &m_roomFromHead);
```

For trackers that provide them, velocity and acceleration information is also available in the interface state, supporting predictive tracking.

Projection and Viewing Transformations

This section describes projection transformation and the portions of the viewing transformation required to set the viewpoint. It is adapted from [OSVRView16]. It assumes planar rectangular screens. See the ***Distortion Correction*** section below for how to convert distorted systems (non-ideal lenses, non-planar display surfaces) into the planar model used here.

This discussion ignores the effects of eye tracking, and it also ignores the fact that the center of projection of the eye is not the same as its center of rotation. (The center of projection is always along the viewing direction in front of the center of rotation, so the approximation is slight.) With eye tracking the only change is to move each eye's position from the center of rotation of the eye to the tracked center of its entrance pupil.

Overview

The purpose of the combined projection and viewing transformations are to provide a geometric description of how to properly project 3D points onto one or more rectangular planar screens that the viewer is looking at in such a way that they appear to remain fixed in space as the viewer's head moves and rotates.

When dealing with fixed-screen displays: head-tracked stereo on a monitor, CAVE displays, or VR desk designs, the screen remain fixed in room space and the eyes move around. When dealing with head-mounted displays, each screen moves along with the eyes and remains at the same location in head space. Although this affects how these locations are determined, it does not affect the basic mathematics involved or the approach to determining the viewing transformations.

To make the discussion easier to illustrate and understand, it is presented initially for the 2D case and then extended into 3D.

Without Lenses

Figure 32.3 shows the situation without lenses. This matches the case for fixed-screen displays, but is an approximation for head-mounted configurations. (The case with lenses will be discussed next.) The transformations can be computed in physical-world units, for example meters; they do not depend on the window size in pixels. The location and size of each screen and all eye positions are all that is needed to determine the projection and viewing transformations. Methods for determining these locations are described in the ***Implementation*** section.

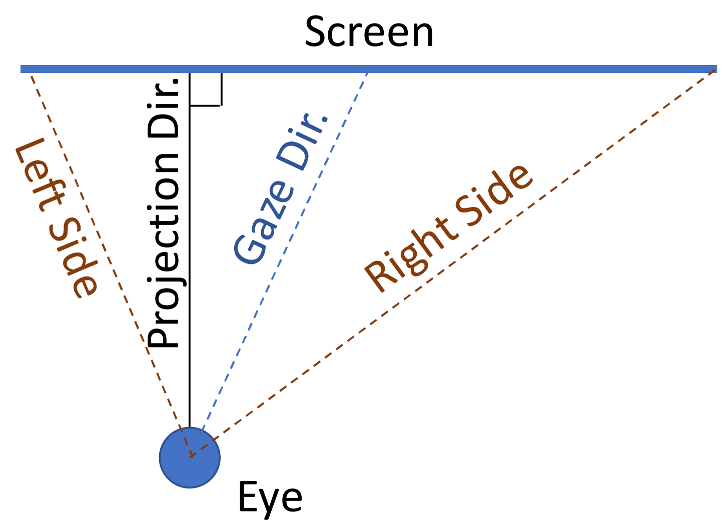


Figure 32.3: Top view of the projection transformation parameters without lenses. In general, the projection direction is not the same as the gaze direction towards the center of the screen (off-axis projection).

Projection Transformation

The result of projection is a 2D image on a planar projection surface. To appear correctly when drawn on the screen, this projection surface needs to be parallel to the physical screen and it must subtend the same region on the retina. It can be moved closer or further to the eye, but then must be scaled so that its edges are at the same projected locations as the real screen. This is easiest to think about in terms of angles and a viewing direction, which are independent of depth.

It is tempting to project along the presumed *gaze direction*, which is towards the center of the screen. However, doing so would project onto a planar surface that is not parallel to the screen. To make the two parallel, the *projection direction* must be perpendicular to the screen, along its normal vector. Note that this will make the center of projection lie outside the screen if the gaze direction is sufficiently off center. (Another term for this case, where the gaze and projection vectors differ, is *off-axis projection*.) Given this unique projection direction, two angles can be specified, one for each edge of the screen. One rotates the projection direction to point towards the left edge of the screen (a slight positive rotation in figure 32.3) and the other to point to the right edge of the screen (a large negative rotation in figure 32.3). Chapter 33 provides details of constructing an off-axis projection matrix. [Kooima18]

Viewing Transformation

The job of the viewing transformation is to place the center of projection at the location of the eye in *Virtual World* space (where the graphical objects to be rendered are defined). This placement requires translating the origin in *Eye* space to its location in world space and rotating it so that the negative Z axis in eye space is looking along the projection direction in world space. Together with the projection transformation, this operation takes 3D points in world space and projects them onto a virtual screen that is consistent with the pose of the physical screen compared to the center of the physical viewer's eye location.

Going to 3D

Going from the 2D-to-1D projection example above to a 3D-to-2D projection requires another translation to set the vertical location of the eye with respect to the screen. It also requires two more rotations. These rotate the world so that the X axis in eye space is parallel to the X axis in screen space (from left to right) and the Y axis is parallel to the Y axis in screen space (from bottom to top). Together with the Z rotation, this aligns the four corners of the virtual screen with the corners of the physical screen.

Implementation

The projection and viewing transformations must ultimately be implemented in the graphics library being used to display the world. These libraries (*OpenGL*, *Direct 3D*, *Unreal*, *Unity*, *Vulkan*, and others) each have their own coordinate systems. Some of the them are right-handed, and some left-handed. Some have the origin at the upper-left corner of the screen and some at the lower left. Some have specified world-space units (meters, centimeters) and some do not.

The coordinate system used in this chapter matches the OSVR internal coordinate system. As with many toolkits, OSVR includes utility adapters to convert its internal representations of viewing and projection matrices to the formats used by various rendering libraries.

Finding Eye Space

In OSVR, *Head* space is defined with its origin halfway between the center of rotation of the viewer's eyes, with its X axis pointing towards the right eye, its Y axis pointing up out the top of the viewer's head, and its Z axis pointing out the back of the head. Getting from room space to head space is the job of the tracking and interaction systems as described elsewhere. Getting from head space to eye space for each eye is a matter of translating along the X axis by half of the inter-pupillary distance (IPD) in +X for the right eye and -X for the left.

Determining Screen Edges

For head-mounted displays without eye tracking, the screen edges are fixed in eye space. This means that the projection transformation remains fixed for a given eye as the viewer's head moves around the environment so that only the viewing transformation requires updating.

For fixed-screen displays, the location of each screen must be measured or constructed such that the locations of its edges are known in room space. The relative position and orientation of each eye relative to each screen changes between each rendered frame, so both the viewing and projection transformations must be updated. This is also required for eye-tracked head-mounted display (HMD) systems.

Because the screens in the HMD are too close for most viewers to focus on, HMD displays employ lenses to turn the physical screen into a virtual image. The impact of this is described in the next section.

With Ideal Lenses

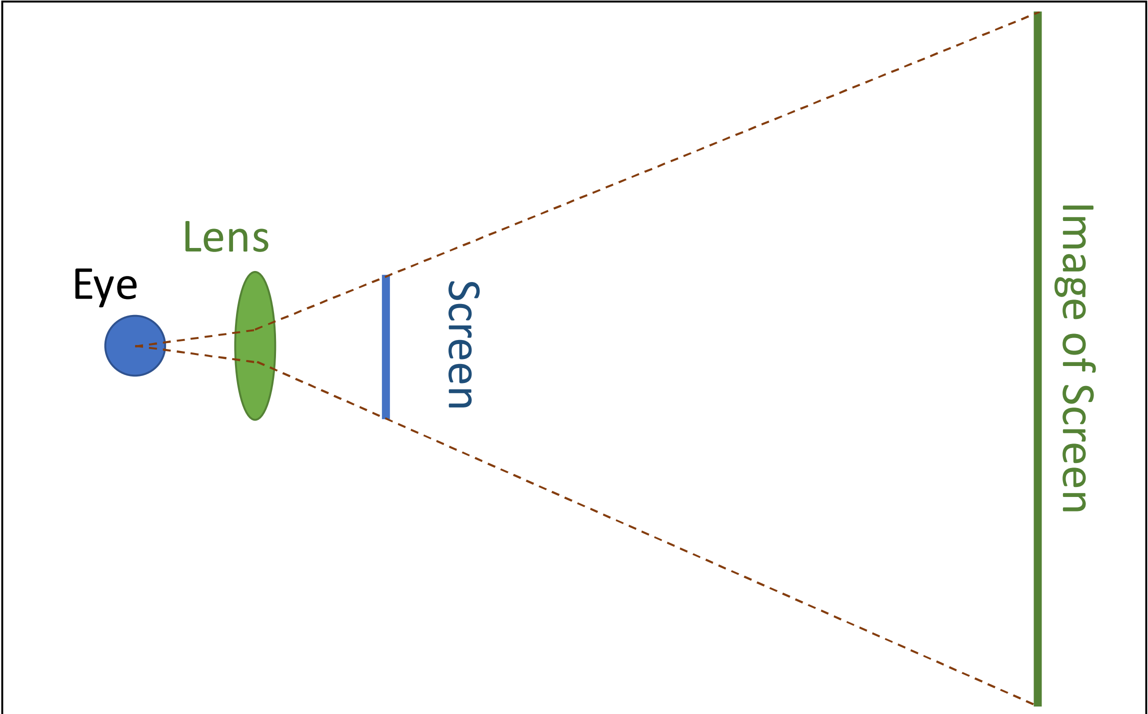


Figure 32.4: An ideal lens magnifies the physical screen and produces a virtual image of the screen that appears to be much further from the eye. An ideal lens produces a flat, undistorted screen.

An ideal lens uniformly magnifies or minifies all objects behind it, scaling their size and distance consistently. As shown in figure 32.4, this enlarges the screen and moves it further away, producing an *image of the screen* that is parallel to the actual screen but located behind it. Note that the rays through the lens bend, causing the image to appear larger than a direct projection would be, yielding a wider field of view.

The algorithm for determining the projection and viewing transforms remains the same as above, but now all measurements are made from the image of the screen rather than the physical screen. This is not theoretically challenging, but it is a practical challenge to determine the when the lens parameters are not known because the edges of the screen may not be visible through the lens. Assuming the lens characteristics and the physical size and location of the screen with respect to the lens are known, the location of the image can be computed. If not, calibration is needed.

Adjustable Lenses

Some head-mounted displays, such as the *OSVR HDK 1.2*, let the user adjust the location of the lenses to make room to insert eyeglasses into the display and to adjust the width so that their pupils stay within the *exit pupil* of the lens (the region where it behaves like an ideal lens, sometimes referred to as the *eye box*).

For an HMD where the lenses can be moved with respect to the screens, the position and size of the image of the screen move with respect to the eye. As the lens is moved closer to the screen (further from the eye), the image of the screen appears to get larger, producing a larger field of view. As the lens is moved to the left, the image of the screen appears to move to the right. When the lens remains stationary and the eye is moved within the exit pupil of the lens, the image of the screen appears to remain the same. (When the eye is moved outside the lens exit pupil, additional distortion is seen.)

So long as the viewer's pupil remains in the exit pupil of the lens, the projection and viewing transformations should be based on the actual center of projection of the eye (based on the IPD) rather than based on the nominal center of the exit pupil for the lens and based on the virtual image of the screen. If the viewer's pupil goes beyond the exit pupil of the lens, causing distortion, then the HMD lens locations should be adjusted and the system recalibrated.

Specification of the screen

The following describes a compact general description of a screen, for a description of how to specify these parameters in OSVR configuration files, see the ***Specifying the screen*** subsection of the ***Distortion Correction*** section below.

Fixed Rectangular Screen: The specification of a rectangular fixed-screen systems can be done by specifying the room-space coordinates of the lower-left, lower-right, and upper-left corners in meters. Because this allows a non-rectangular result, in the case where the vectors from the lower-left corner to the lower-right and upper-left corners are not orthogonal, the projection of the upper-left coordinate onto the plane perpendicular to the vector from the lower-left to lower-right corner will be used as the upper-left corner (which will reduce the screen height).

Head-Mounted Displays (HMDs): The screens in a head-mounted display may be mounted at any angle with respect to each other and with respect to the device and the viewer's relative eye positions. The lack of a standard for fiducials on head-mounted displays and the fact that some can be individually adjusted means that no coordinate system can be defined with respect to the HMD itself that will be correct in all circumstances.

A general solution describes the location of the corners of the image of the screen with respect to *Head* space, which has its origin halfway between the center of rotation of the eyes, its X axis pointing towards the right eye, its Y axis pointing up, and its Z axis pointing towards the back of the head. There is a separate definition for each screen. Although the viewing and projection matrices will depend on the viewer's IPD, the screen location depends only on the lens locations (presuming that the viewer's eyes lie within the exit pupils for each lens). As with the fixed rectangular screen each screen is specified by providing three sets of 3D coordinates: the image of the screen's lower-left corner, its lower-right corner, and its upper-left corner. These corners are the locations of the image of the screen even if those locations are not visible to the viewer through the lenses (this can make calibration challenging). As for fixed rectangular screens, in the case where the vectors from the lower-left corner to the lower-right and upper-left corners are not orthogonal, the projection of the upper-left coordinate onto the plane perpendicular to the vector from the lower-left to lower-right corner will be used as the upper-left corner.

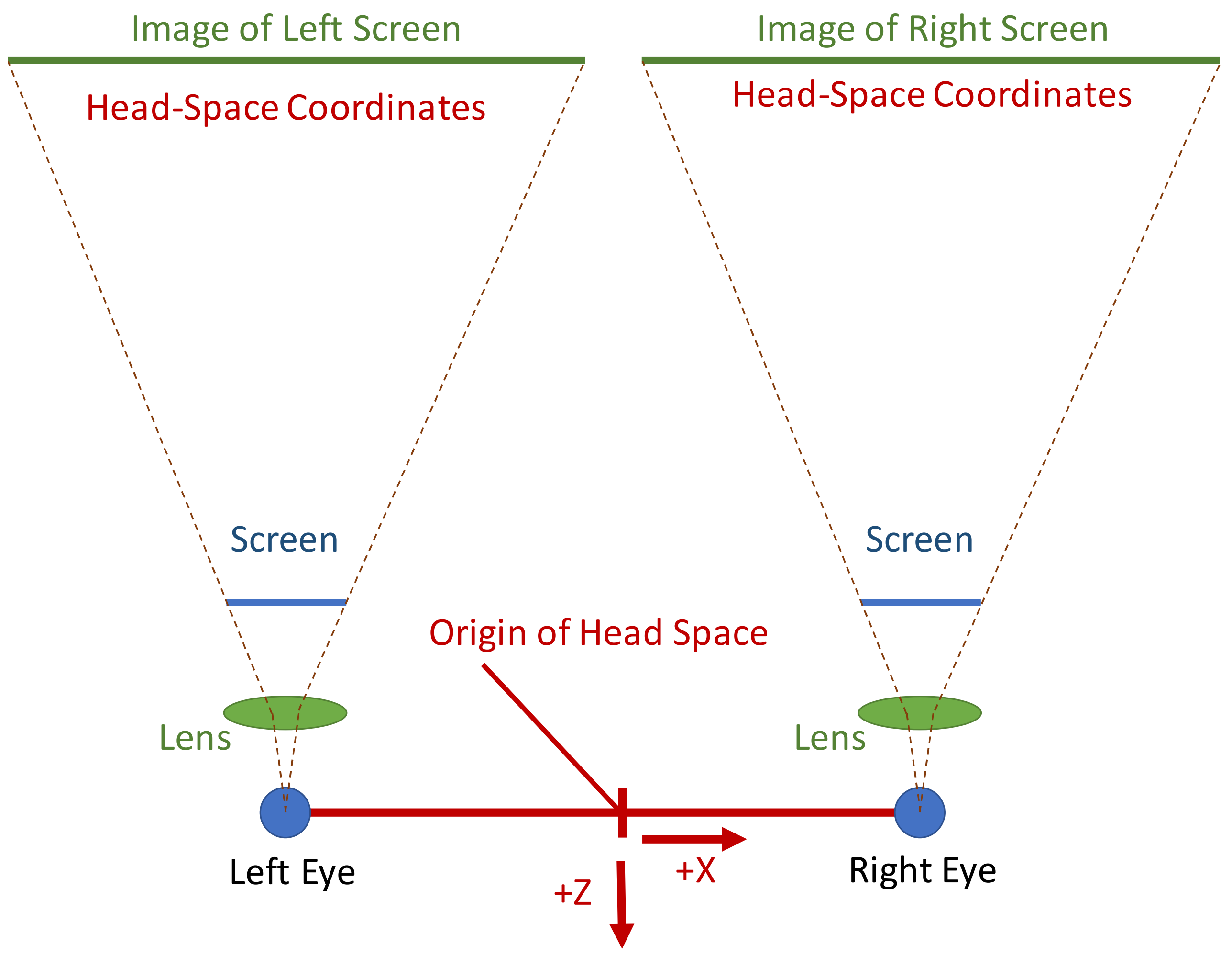
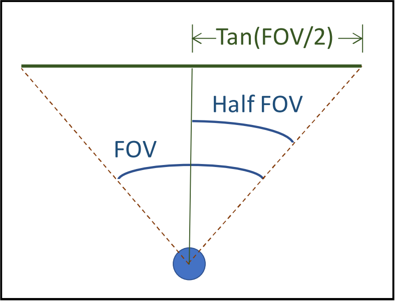


Figure 32.5: The corners of the screens are specified in *Head* space, whose origin is halfway between the two eyes with the -Z axis facing forward and the +X axis pointing from the left eye towards the right.

Specification for HMDs whose lenses introduce distortion, along with fixed curved screens is described in the ***Distortion Correction*** section below. The basic approach is for the distortion correction to map pixels from the physical display onto an appropriately-defined rectangular screen that this chapter will refer to a "*canonical screen*" and then to specify three corners of this canonical screen (whose corners may or may not be visible) as described above.



A note on field of view (FOV) calculations: As seen here, the viewport width is proportional to the tangent of half of the horizontal field of view, and the height to half that of the vertical field of view. This means that for non-square aspect ratios, the ratio of the window width/height is not directly proportional to the ratio of the HFOV/VFOV. This means that *you cannot multiply the horizontal field of view by the ratio of the display size in pixels to compute the vertical field of view*.

For example, the horizontal field of view on the *OSVR HDK 1.2* is 90 degrees and it covers half of the screen (1920/2). It is *incorrect* to compute the vertical field of view using 90 / ((1920/2) / 1080) = 101.25 degrees. The correct calculation is *atanDegrees((tanDegrees(90/2) / ((1920/2) / 1080))) * 2 = 96.73 degrees*. The diagonal field of view uses the screen diagonal size in pixels (1445) rather than 1920/2 to get a diagonal field of view of 112.8 degrees.

Distortion Correction

This section adapts [OSVRDistort16] and describes how to remove the effects of distortion caused by curved screens and lenses. Although it is possible to construct lens systems that do not introduce distortion, weight and cost constraints on HMDs often lead to the use of lenses that do cause distortion. Removing this distortion can be handled as part of rendering.

The basic function of distortion correction is to map locations from a rectangular, planar so-called *canonical screen* that is defined by the distortion-correction algorithm onto coordinates within a physical display being viewed through a lens that causes distortion. This same approach can be used to undistort pixels that are presented on a non-rectangular or non-planar display (such as a curved TV or a projection that includes keystone or that is onto a non-planar surface). Note that this transformation can be specified in fractional screen coordinates in a similar manner to texture coordinates and does not depend on display resolution – the distortion remains the same even when the number of pixels being displayed changes.

In the overall rendering process, the projection and viewing transformations take points in 3D model space and project them onto the rectangular and planar canonical screen, and then distortion correction adjusts the resulting image to undo the nonlinear effects of lenses or curved screens used to view it, mapping each point from its canonical location back into its physical location.

Approach

The distortion correction is free to select any rectangle as the canonical screen to be projected on, so long as it properly undistorts images rendered onto that rectangle. We shall see that the canonical screen should lie in depth within the range of the virtual image of the real screen to reduce shift in distortion as the eye rotates to look in different directions.

Two special cases of distortion correction are presented and then a more general solution is described.

Case 1: Curved screen

The image below shows an example of a curved display (like the currently-available OLED TVs) viewed without a lens from a viewpoint in the middle of the screen along the screen's normal at that location. The center of the image shows a top-down 1D view of the scene and a first-person view of the screen from the eye's point of view. Note that the distortion correction for a curved screen depends on the viewer's eye position. The more curved the screen, then more pronounced the effect. This is also true for other forms of distortion.

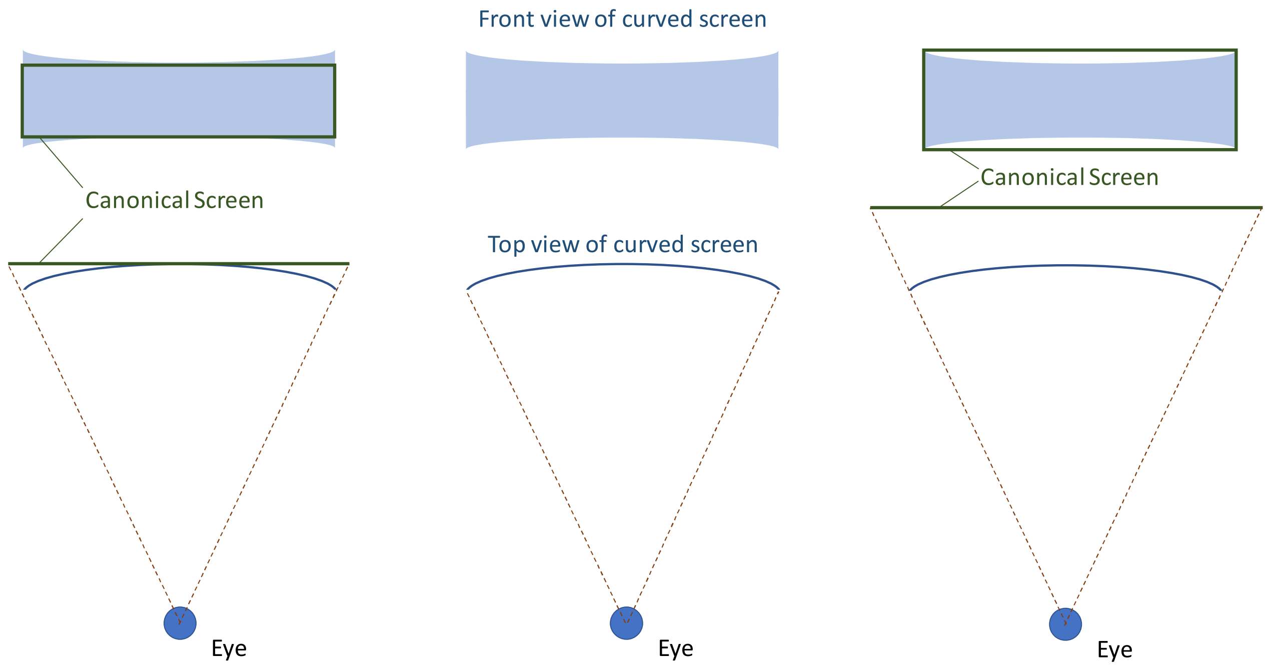


Figure 32.6: Two potential ***canonical*** screens for a curved-screen display viewed straight on. **Left:** The canonical screen is tangent to the physical screen and covers only a subset of the screen's surface (it misses the corner regions). **Right:** The canonical screen is behind the physical screen and covers the entire screen and a region beyond it.

In figure 32.6 left, the blue screen is the (distorted) screen that would be seen and the green rectangle is one possible choice for an undistorted canonical screen. We are free to choose any depth for the canonical screen so long as we adjust its projected size to match the extents seen in the 2D view – it could be brought closer and scaled down or pushed back further and scaled up. This particular canonical screen is smaller than the physical screen – there are locations on the physical screen that are outside the canonical screen.

Overfill

There are some locations on the physical screen shown in figure 32.6 left that do not correspond to any location on the chosen canonical screen. This means that there is no image to be moved to that location. To avoid this, the canonical screen (green) can be selected so that it completely includes the physical screen, which will provide a mapping for every point on the physical display (but will also necessarily provide "wasted" mappings for some points outside the physical display). This example is shown in figure 32.6 right.

This overfill is also required for other distortions, including radially-symmetric distortions. This is because any non-linear distortion will turn the rectangular boundary of the screen into a set of curves.

(In figure 32.6 right, the canonical screen is behind the real screen, which will cause the distortion correction to depend more strongly on eye position. A better solution would place the canonical screen somewhere within the depths covered by the physical screen, rendering into offscreen regions as in the center case.)

Correcting the distortion

Figure 32.7 shows two potential approaches to undistorting cylindrical projection and describes some reasons that might lead to choosing one over the other.

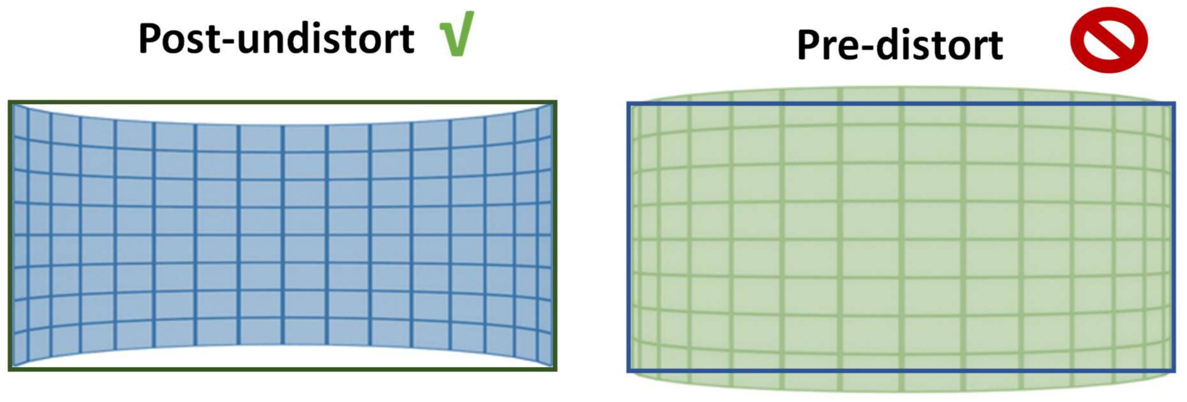


Figure 32.7: Two approaches to distortion correction. **Left:** Construct a map from every point in the real screen (blue) to corresponding locations within the canonical (green) texture and use this to pull the correct pixel in the shader. **Right:** Render pre-distorted geometry texture mapped with the canonical (green) image into the display (blue), pushing each pixel to its correct location.

Pre-distortion and its deficits: The right side of figure 32.7 shows an approach that might be taken, which is to pre-distort the original scene geometry by the inverse of the optical distortion that will be done by the cylindrical projection so that the resulting rendered image can be directly drawn on the (blue) physical screen and have the correct projected image. This requires applying an arbitrary nonlinear mapping to the geometry, which is not easily done and which the graphics hardware would piecewise-linearly interpolate across triangles. Another way to do this same pre-distortion is to do a standard rendering pass and then do a second pass where the original (green) canonical texture generated in the first pass is rendered onto distorted geometry that projects onto the appropriate location on the (blue) screen. Either approach produces an image that includes a non-linear warping of the geometry, resulting in an image that cannot be translated or rotated to handle temporal corrections because of head rotation (see the ***Time Warp*** section below).

Post-undistortion: The left side of the figure 32.7 shows another approach, which is to determine which point on the canonical screen (green) corresponds to each point on the distorted real screen (blue). This makes use of the fact that the graphics system can render into a texture that is handed to the VR system for presentation. During the final render pass, the texture coordinates for each point are adjusted to read each visible pixel from its corresponding location in the green texture. This undistortion can be done in the graphics library's *vertex shader* by producing a dense mesh that has adjusted texture coordinates per color or in the *pixel shader* either by applying a function to the texture coordinates or using a texture map to provide the new texture coordinate per color to map to the proper location on the screen. This approach has the benefit that the image sent to the final rendering pass is still a linear projection, enabling it to work with other techniques described later, such as time warping.

Case 2: Per-color radial distortion

Many lenses have *chromatic aberration* (a different magnification for each wavelength of light), resulting in three different distorted images, one per primary color. This distortion happens in addition to the desired behavior of the lens, which is to magnify the physical display and to move its virtual image further from the eye so that the viewer can focus on it. It is possible to make lens systems that are achromatic and produce the same per-color distortion, but it is also possible to correct for this chromatic distortion within the rendering system.

Although the position of the virtual image of the screen for an ideal lens does not depend on the position of the user's eye (so long as the eye is in the exit pupil for the lens), radial distortion does depend on the location of the viewer's eye. This means that completely correcting for radial distortion requires accounting for the location of each of the viewer's eyes relative to its lens as well as knowing the location of each lens with respect to its screen.

The following parameterization provides one approach (the one used by OSVR) to specifying this type of distortion:

- **Center of projection:** This provides the coordinates for the location on the virtual image of the screen where the ray from the center of the viewer's eye through the center of projection of the lens intersects it. This is a fractional coordinate from 0-1 in each axis, with the lower-left corner of the screen being (0,0) and the upper-right being (1,1).
- **Distance scales:** Because distortion correction depends on both the lens geometry and the screen geometry and may not be directly related to the viewport size or aspect ratio (for lenses that expand more in one direction than the other), one must specify not only the radial distortion polynomial coefficients (which scale powers of the distance from the center of projection to the point), but also the space in which this is measured. This is specified as the number of unit radii in the space the parameters are defined in that span the texture coordinates, which range from 0 to 1. This can be different for X and Y, as the viewport may be non-square and the lens system may make yet a different aspect ratio. There are separate D components for width (D_X) and height (D_Y).
- **Per-color coefficients:** A set of polynomial coefficients can be provided for each color. The coefficients can specify the new radial displacement from the center of projection as a function that scales the original displacement. The first coefficient in each polynomial is a constant factor (multiplied by offset^0, or 1), the second is the linear factor, the third is quadratic, and so forth. There can be as many coefficients as desired.

The coefficients for R, G, and B; the Distances for X and Y; and the center of projection (COP) may be specified in any consistent space that is desired (scaling all of them linearly will have no impact on the result), but the lower-left corner of the space (as viewed on the canonical screen) must be at (0,0) and upper-right must be at (D_X,D_Y).

The parameters for each color specify the new radial displacement from the center of projection as a function of the original displacement. Listing 32.3 shows how to calculate the distorted location based on an original location and the above parameterization:

Listing 32.3: Calculating radial distortion.

```
// Orig is the (x,y) coordinate specified with X in (0..Dx) and Y in (0..Dy)
// COP is the center of projection specified in normalized screen coordinates (0..1) for X and Y
// D is (Dx,Dy) as described above
// Final is the radially-distorted coordinate
Offset = Orig – COP*D;                                    // Vector, component-wise multiplication
OffsetMag = sqrt(Offset.length() * Offset.length());      // Scalar
NormOffset = Offset / OffsetMag;                          // Vector
Final = COP*D + (a0 + a1*OffsetMag + a2*OffsetMag*OffsetMag + ...)  * NormOffset; // Location
```

Examples: (1) For a display 10 pixels wide by 8 pixels high that has square pixels whose center of projection is in the middle of the image, we would get: D = (10, 8); COP = (0.5, 0.5); parameters specified in pixel-unit offsets. (2) For a display that is 6 units wide by 12 units high, but whose optics stretch the view horizontally to produce a square viewing image with pixels that are stretched in X, we could have: D = (12, 12); COP = (0.5, 0.5); parameters specified in vertical pixel-sized units **or** D = (6,6); COP = (0.5, 0.5); parameters specified in horizontal pixel-sized units.

More general solution: Using a screen-point-to-angles table

Suppose that either through direct measurement with a camera or through ray-tracing in the optical design for a head-mounted display, you produce a mapping between physical points on the display screen and angles from the center of the eye, for a given IPD. This mapping can be arbitrary, so long as it is a mathematical function (does not contain folds) and it may be an unordered set of points. Assume that angles are specified in degrees from views looking along the -Z axis in head space (straight forward) and the positions on the display are specified as distances in millimeters from the point on the display that corresponds to the point that would be see at angle (0,0). Further, assume that the focal distance to the virtual image of the real screen is around 2 meters (some portions being closer, and some further). An example of this is shown in figure 32.8.

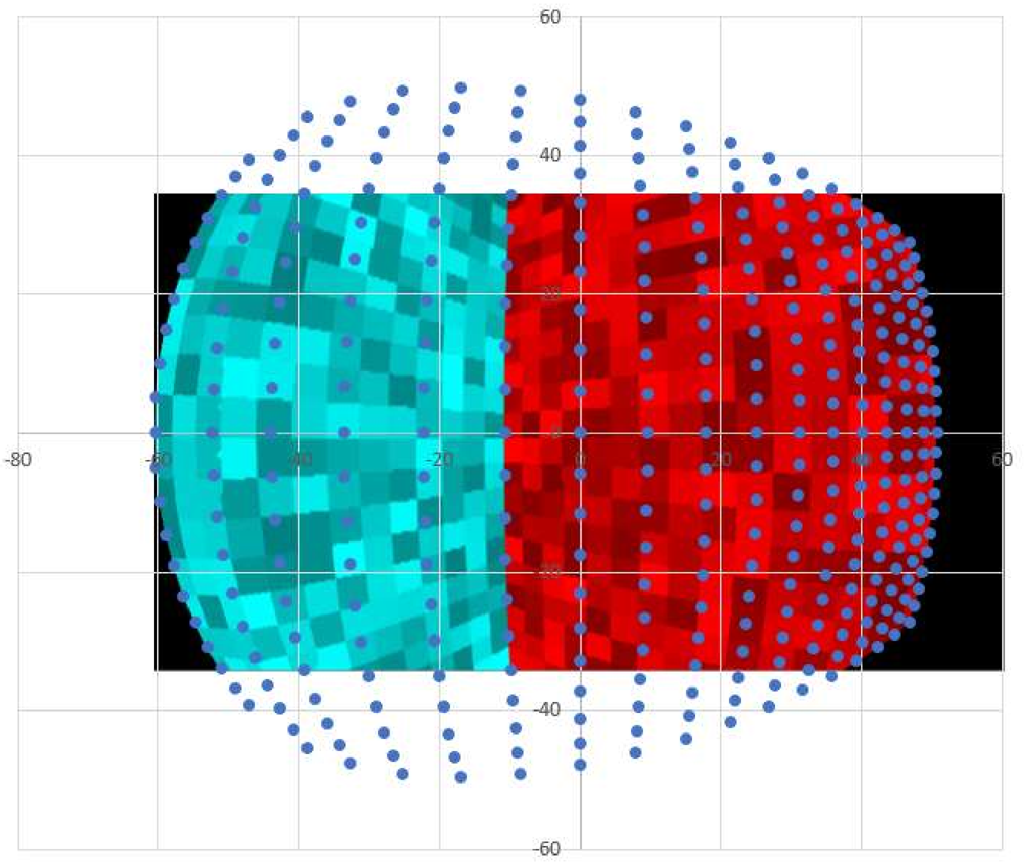


Figure 32.8: A mapping from real-world angles to physical-display locations drawn on top of a rectangular physical display on which is drawn an image distorted by its inverse so that it will look correct when viewed through the lens. Each blue dot represents one sample of the mapping. Dots that lie outside the rectangle will not be visible on the display. Black areas in the display are outside the mapped region so either are not visible through the lenses or must have their distortions extrapolated from dots within the region.

Step 1: Determine a canonical screen that spans the physical screen

The *eye-space location* of each point is computing using polar coordinates, using the 2-meter focus estimate as the radius. The longitudinal angle is assumed to have positive spin around the Y axis with 0 facing forward along the –Z axis and the latitudinal angle is assumed to be positive when rotating up towards the +Y axis.

The *X screen-space extents* are defined by the lines perpendicular to the Y axis passing through:

- **Left:** the point location whose reprojection into the Y=0 plane has the most-positive angle (note that this may not be the point with the largest longitudinal coordinate, because of the impact of changing latitude on X-Z position).
- **Right:** the point location whose reprojection into the Y=0 plane has the most-negative angle (note that this may not be the point with the smallest longitudinal coordinate, because of the impact of changing latitude on X-Z position).

The *Y screen-space extents* are assumed to be symmetric and correspond to the lines parallel to the screen X axis that are within the plane of the X line specifying the axis extents at the largest magnitude angle up or down from the horizontal. This is the point with the largest-magnitude Y value when it is projected into the plane of the screen as determined by the X screen-space extents.

Because the projections of all points in the set will lie within these screen-space extents, no points from outside this region correspond to any point on the physical screen. If the mapping provides angles for each point on the physical screen, there will be a point on the canonical screen to map to. If not all points on the physical screen have mappings, it may be necessary to overfill the render region to provide them (see the ***Overfill*** section below).

Note: The approach described above will only work for displays whose fields of view do not extend 90 degrees from forward in either the nasal or distal orientation. (Planar projection in general will only work for displays whose monocular horizontal field of view is less than 180 degrees. Displays with larger fields of views will need to be rendered using multiple projections that are stitched together.) For displays that have fields of views less than 180 degrees but which extend beyond 90 degrees distal, the reprojection must be done not on the Y=0 plane but on a plane rotated away from the nose such that all displayed angles pass through it. A similar rotation vertically could be used to handle displays that are asymmetric about the X axis.

Step 2: Mapping from physical screen coordinates

Given points in the physical screen, the distortion map provides the coordinates of the corresponding point on the canonical screen. This determines the appropriate point to display at this location on the screen. This is calculated in two steps:

- **Step 2A:** Map from physical-display coordinate to angle using the provided table.
- **Step 2B:** Map from angle to canonical-screen coordinate by projecting the ray from the eye onto the plane of the canonical screen. Then determine the screen-space X and Y coordinates (X = 0 at left and 1 at the right, Y = 0 at the bottom and 1 at the top).

Doing this mapping for points other than those specified in the table requires interpolation for display points between those specified and extrapolation for points outside their convex hull.

Implementation of distortion calibration within OSVR

The above procedure is implemented in the *angles_to_config* program in the OSVR *Distortionizer* project [OSVRAngles17]. Additional details (described below) are needed to describe the general results above in a manner usable by OSVR.

Specifying the screen in the server configuration file

The current OSVR display description includes the specification of a *horizontal field of view*, a *vertical field of view*, a *center of projection* (which is the normalized location on the screen where the line through the eye point perpendicular to the screen pierces the screen) and a *percent overlap* (which is related to the rotation of the screens around the Y axis).

Figure 32.9 shows some of the relevant parameters. Following it, the entries in the OSVR server configuration file are specified, along with a description of how to compute each of their values.

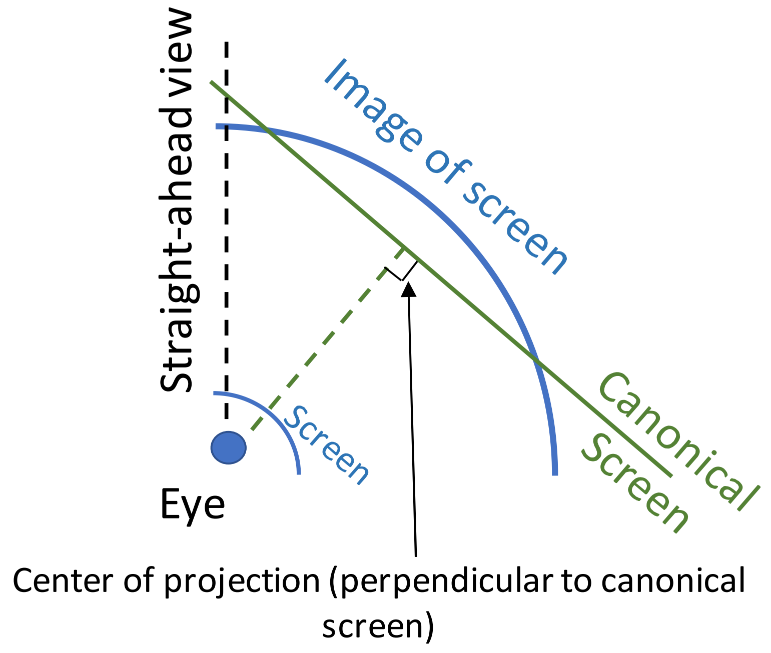


Figure 32.9: Configuration of an HMD with a curved screen that is rotated to provide a larger distal field of view. The eye location is not in the center of the screen, resulting in an off-center projection. The angle between the projection and the straight-ahead view is specified in the *overlap_percent* parameter in the OSVR configuration file.

- **display/hmd/field_of_view/monocular_horizontal:** This value is computed as if the screen is being viewed by an eyepoint located along the line perpendicular to the center of the screen. We determine it using the half-screen width and the perpendicular distance from the origin to the plane of the screen.
- **display/hmd/field_of_view/monocular_vertical:** This value is computed as if the screen is being viewed by an eyepoint located along the line perpendicular to the center of the screen. We determine it using the half-screen height and the perpendicular distance from the origin to the plane of the screen.
- **display/hmd/field_of_view/overlap_percent:** This percentage is computed as if the screen is being viewed by an eyepoint located along the line perpendicular to the center of the screen and as if both eyes were co-located (IPD = 0). (Note; the resulting viewing transform does not make this assumption, just the current algorithm to map from *overlap_percent* to angle.)
- **display/hmd/eyes[0]/center_proj_x:** This location is computed as the fraction of the distance from the left side of the screen to the right side where the line through the eye perpendicular to the screen crosses the screen. This value subtracted from 1 is used in eyes[1]/center_proj_x.
- **display/hmd/eyes[0]/center_proj_y:** Because there is currently no way to specify screens that are tilted up and down with respect to the Y=0 plane, this value is always 0. The value of eyes[1]/center_proj_y is also 0.

Producing the distortion map in the server configuration file

The configuration file format allows the specification of a variety of distortions, identified by the *display/hmd/distortion/type* variable. If the red, green, and blue components of the distortion are all the same, the type *mono_point_samples* can be used. This means that we need to specify just one distortion mesh, which maps from normalized (X,Y) coordinates in a the physical display ([0,0] at the lower-left corner, [1,1] at the upper right) into normalized coordinates in the canonical screen.

We compute the input normalized coordinates for the mesh by normalizing the table's display coordinates to convert them from millimeters to screen fractions, subtracting the coordinates of the lower-left corner of the screen and dividing each axis by the screen dimension. We compute the output coordinates as described in Step 2.

We then store the unordered set of points into the *display/hmd/distortion/mono_point_samples* array, which has a vector of elements, each of which has two elements, the first of which is the 2D coordinates in normalized physical-screen coordinates and the second of which is the 2D coordinates in the canonical-screen coordinates.

An example output, which is a partial description of an HMD, is shown in listing 32.4. It provides the identity mapping.

Listing 32.4: HMD general distortion configuration file example.

```
{
  "display": {
    "hmd": {
      "distortion": {
        "type": "mono_point_samples",
        "mono_point_samples": [
          [ [0,0], [0,0] ],
          [ [1,0], [1,0] ],
          [ [0,1], [0,1] ],
          [ [1,1], [1,1] ]
        ]
      }
    }
  }
}
```

The ***OSVR RenderManager*** uses this set of unordered point samples to compute a mesh by using a bilinear fit to the nearest 3 non-coplanar points to determine each of the coordinates for each point in space that must be sampled to produce a mesh with the specified number of points.

Implementation of distortion correction within OSVR

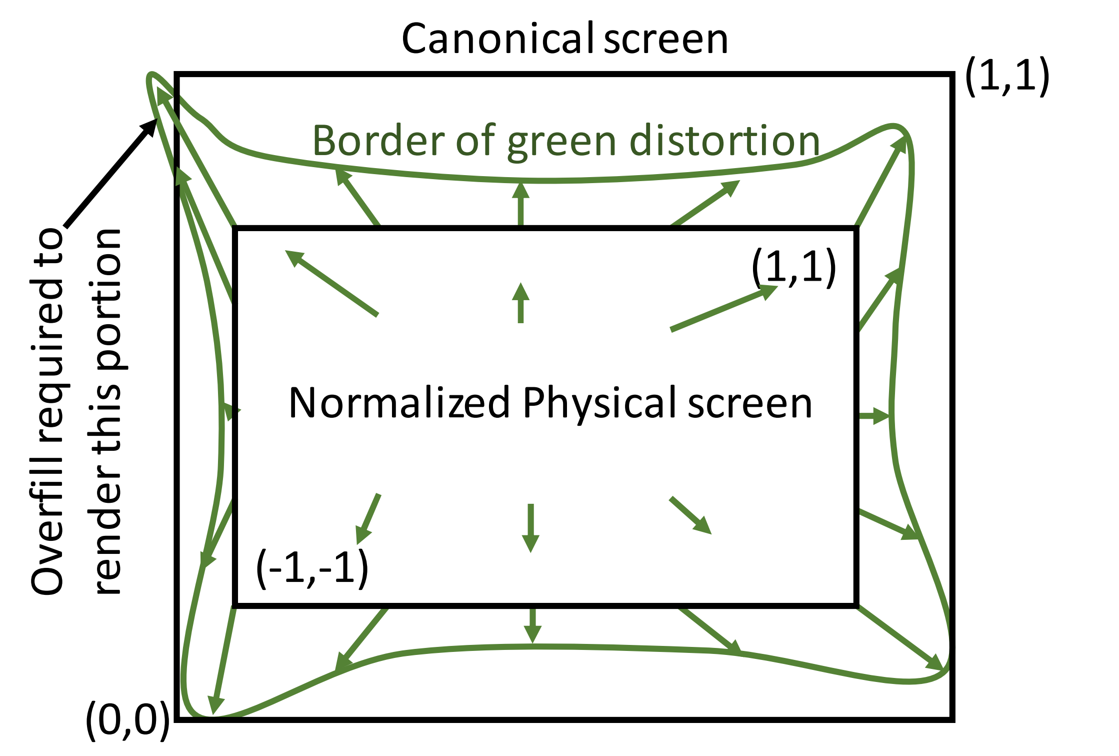

Canonical screen
(1,1)
Border of green distortion
(1,1)
Overfill required to render this portion
Normalized Physical screen
(-1,-1)
(0,0)

Figure 32.10: OSVR uses a regular mesh to describe the mapping from normalized physical-screen coordinates to texture coordinates within or beyond the Canonical screen. The green map is shown here; there are separate maps for red and blue. The green arrows show the mapping from original (base) to distorted (arrow end) for a subset of the points on a mesh; the curved line shows the mapping for the border of the mesh.

As shown in figure 32.10, distortion correction is implemented within the *Sensics OSVR-RenderManager* component [OSVRRenderManager17] by storing a set of texture coordinates for each color with each vertex in the mesh that describes the virtual screen rectangle as shown in Listing 32.5:

Listing 32.5: Distortion mesh structure.

```
/// 2D float data, like a texture coordinate for example.
using Float2 = std::array<float, 2>;
/// Describes a vertex 2D position plus three 2D texture coordinates.
class DistortionMeshVertex {
public:
    DistortionMeshVertex(Float2 const& pos,
                         Float2 const& texRed, Float2 const& texGreen,
                         Float2 const& texBlue)
        : m_pos(pos), m_texRed(texRed), m_texGreen(texGreen),
        m_texBlue(texBlue) {}

    // Flips a texture coordinate that is in the range 0..1 so that
    // it is inverted about 0.5 to be in the range 1..0.  Useful for
    // flipping OpenGL Y coordinates into Direct3D ones.
    static float flipTexCoord(float c) { return 1.0f - c; }
```

```
        Float2 m_pos;              //< X,Y
        Float2 m_texRed;           //< U,V
        Float2 m_texGreen;         //< U,V
        Float2 m_texBlue;          //< U,V
    };

    class DistortionMesh {
    public:
        std::vector<DistortionMeshVertex> vertices;
        std::vector<uint16_t> indices;
    };
```

The (X,Y) coordinates describe the normalized physical-screen-space location of vertices that span the range -1 to 1 in X and Y; four vertices are sufficient to describe a linear transformation but more are needed to describe distortion. The (U,V) texture coordinates describe the relative location within or beyond the canonical screen to look up the color associated with that vertex location in the physical screen and they are linearly interpolated by the graphics library between the vertices. The lower-left corner of the canonical screen is at (0,0) and the upper-right is at (1,1). See the ***Overfill*** section for how points outside this range are handled.

Each rendering library (*OpenGL*, *Direct3D*, *etc.*) implemented in OSVR passes these coordinates to its vertex shader, where they are used to look up the location within the texture map associated with each eye. The OpenGL GLSL vertex shader program to perform this lookup (along with the projections used to handle projection, viewing, and time warp) as shown in Listing 32.6:

Listing 32.6: GLSL Vertex Shader implementing distortion correction and timewarp.

```
#version 100
attribute vec4 position;                //< Homogeneous coordinates for a canonical screen
vertex
attribute vec2 textureCoordinateR;      //< Distorted red texture coordinates for this vertex
attribute vec2 textureCoordinateG;      //< Distorted green texture coordinates for this vertex
attribute vec2 textureCoordinateB;      //< Distorted blue texture coordinate for this vertex
uniform mat4 projectionMatrix;          //< Used to correct for overfill
uniform mat4 modelViewMatrix;           //< Used to handle display scan-out orientation, Y
inversion
uniform mat4 textureMatrix;             //< Used to implement time warp
varying vec2 warpedCoordinateR;         //< Transformed red texture coordinate for fragement
shader
varying vec2 warpedCoordinateG;         //< Transformed green texture coordinate for fragement
shader
varying vec2 warpedCoordinateB;         //< Transformed blue texture coordinate for fragement
shader
void main()
{
  gl_Position = projectionMatrix * modelViewMatrix * position;
  warpedCoordinateR = vec2(textureMatrix * vec4(textureCoordinateR,0,1));
  warpedCoordinateG = vec2(textureMatrix * vec4(textureCoordinateG,0,1));
  warpedCoordinateB = vec2(textureMatrix * vec4(textureCoordinateB,0,1));
}
```

The corresponding fragment shader is shown in Listing 32.7:

Listing 32.7: GLSL Fragment Shader implementing distortion correction and timewarp.

```
#version 100
precision mediump float;           //< Sets floating-point precision used
uniform sampler2D tex;             //< Texture map with image from canonical screen
varying vec2 warpedCoordinateR;    //< Warped texture coordinate for red channel
varying vec2 warpedCoordinateG;    //< Warped texture coordinate for green channel
varying vec2 warpedCoordinateB;    //< Warped texture coordinate for blue channel
void main()
{
   gl_FragColor.r = texture2D(tex, warpedCoordinateR).r;
   gl_FragColor.g = texture2D(tex, warpedCoordinateG).g;
   gl_FragColor.b = texture2D(tex, warpedCoordinateB).b;
}
```

The ***tex*** sampler is the texture passed by the application that represents the eye being rendered. The red, green, and blue coordinates are independently warped by their respective distortion meshes and then reassembled into the fragment color.

Handling Latency and Jitter

Many system latencies combine to produce “motion to photon” delay: tracker sensor delays, tracker finite sampling rates, transmission delays, and synchronization delays on the input side; finite rendering time, O/S and driver buffering delays, reformatting delays, and scan-out delays on the output side. Because of these delays, the poses available to construct the projection and viewing transforms when initiating rendering for a frame differ from the poses that each eye will have when display scan-out happens for that frame. Additionally, for some displays (e.g. HMDs that scan out in portrait mode), the delay for the right eye is different from that for the left eye.

Holloway showed in [Holloway95] that for normal head motions when observing an object of interest at a distance of around 1 meter, each 1ms of total system delay produces about 1mm of offset error in physical space – in a calibrated system errors caused by latency far outweigh all other sources of alignment error. Furthermore, this delay causes motion-dependent “swimming” of the world, which is a major source of discomfort for viewers. During the ~16ms scan out of a screen at 60 Hz, objects move approximately 1.6cm; typical graphics pipelines not designed for VR add up to two additional frames of latency, causing objects to move considerably, and the world to swim uncomfortably, when this is not dealt with.

Furthermore, this delay is not constant: unless steps are taken to synchronize the tracker sampling and rendering to the actual image scan-out, the delays shift over time and cause the scene to appear to jitter back and forth. An extreme form of jitter is when the graphics update rate does not keep up with the display refresh rate. All jitter is perceived as doubled images, which is quite distracting.

VR systems employ several techniques to deal with this latency and jitter, including ***Frame Sync***, ***Predictive Tracking***, ***Time Warp (synchronous and asynchronous)***, and ***Direct Rendering***. Each of these is described in a separate gem below. Not all techniques are employed in every system, but they can be combined to provide a superior experience.

Frame Sync

The underlying rendering and display scan-out circuitry usually runs at a fixed refresh rate, somewhere between 60 and 90 Hz. The currently available frame is scanned out whether or not there is a new image to be displayed, and independent of the rendering initiation or completion time. Thus for long renders an old image may be repeatedly displayed. This section describes how to synchronize rendering with scan-out.

In the case of a single shared buffer between the rendering and scan-out circuitry, so-called *single buffering*, this can result in tearing artifacts when the rendering system clears and then updates the shared buffer while scan-out is occurring – causing neighboring scan lines to be rendered from different frames (a temporal discontinuity – or tear between scanlines). To avoid this tearing in single-buffered mode one must ensure that all buffer clearing and rendering take place during the vertical blanking time at the end of each frame.

A more robust approach to avoiding tearing is to used *double buffering* (or triple buffering), in which case the rendering system is drawing to one buffer while the previously-rendered buffer is being scanned out. Once the renderer completes a frame, it swaps which buffer is to be displayed at the next scan-out and then gets to work rendering the next frame. Double buffering greatly increases the amount of time available for rendering a frame; rather than the small fraction of a frame within the vertical blanking, it can now take an entire frame (or more) to render an image without causing tearing. It also enables seamless decoupling between the rendering loop and the display loop – so long as the rendering does not get more than one frame ahead it will never cause tearing because the frames are swapped out during vertical retrace. The frame-display portions of graphics libraries often provide a way for the application to stall when it would be two frames ahead, waiting until the current frame has finished scanning out before swapping the buffers and returning.

To remove the jitter caused by a variation in the relative timing of render start and the next display scan-out, the application or VR library needs to know when the next scan-out is coming. One approach is to always render ahead so that the graphics library always stalls before returning a new buffer. This approach has the deficit that it starts the new rendering a whole frame before that frame will be scanned out, rendering it with pose information that will be a whole frame behind when scan-out starts. It also does not apply in cases where the application's frame rate cannot keep up with the display frame rate.

Another approach is to use an operating-system-dependent barrier or timing-request function to find out when the next vertical retrace is going to happen, or to find out when the last one has happened (and with the knowledge of the refresh rate compute when the next scan will happen). The application can thus schedule rendering onset such that it will complete just before the next frame is ready to scan out. In this case, double buffering does not add latency because the buffers are being swapped immediately before being scanned out.

There is a subtle remaining issue that is discussed further in the section on ***Time Warp***; different parts of the display scan out at different times. To support intra-frame time warp, the time that the line in the center of the display scans out should be chosen for each eye.

Implementation in OSVR

Frame sync behavior is implemented in the *Sensics OSVR-RenderManager* [OSVRRenderManager17] using different approaches for different situations. The OpenGL and Direct3D11 native code paths are currently implemented using the Simple Directmedia Library (SDL) [SDL17] to obtain windows, and it calls *SDL_GL_SetSwapInterval(1)* to enable vertical sync, which causes frame presentation to block until vertical sync before returning. (The user can also supply their own windowing library in place of SDL2, and an example using Qt is provided in the source code.)

On its *direct rendering* display paths, OSVR uses either vendor-provided routines or observes when vertical syncs happen using OS hardware queries and informs the application of this timing information by providing a timing function that returns the structure shown in Listing 32.8:

Listing 32.8: Render-timing information structure.

```
typedef struct {
    /// Time between refresh of display device
    OSVR_TimeValue hardwareDisplayInterval;

    /// Time since the last retrace ended (the last presentation)
    OSVR_TimeValue timeSincelastVerticalRetrace;

    /// How long until the app must send images to RenderManager
    /// to display before the next frame is presented.
    OSVR_TimeValue timeUntilNextPresentRequired;
} RenderTimingInfo;
```

The application can then busy-wait on this value until it has sufficient time to complete rendering before querying the current tracking pose and initiating the render. At least under the Windows 10 operating system, busy waiting must be performed rather than sleeping because the operating system does not reliably return with a granularity of less than 10 milliseconds. Because each eye may have different timing, the query includes a parameter telling which eye is being rendered. Listing 32.11 in the *time warp* discussion shows the implementation for waiting for render completion.

Predictive Tracking

The *inertial measurement units* included in many VR tracking systems provide direct measurements of positional and (acceleration and rate of rotation). The *Kalman* and other *optimal estimation filters* used to perform *sensor fusion* on the tracking systems can also estimate these derivative estimates along with the location and orientation. The resulting *state vector* can be used to estimate a *pose* (location and orientation) at points in time other than the present, such as the expected future time when the next frame to be rendered will be displayed. Ron Azuma showed that such predictions can improve tracking for delays of up to about 80ms [Azuma95]. This section describes how to harness predictive tracking to reduce perceived latency.

This estimation is done by standard physics-based double integration of acceleration and single integration of orientation changes over the time difference between the start of rendering and the expected scan-out. This estimation should be done separately for each eye because scan-out often does not start at the same time for each. Because of finite display scan-out time, it is advisable to calculate the delay to the center of the scanned-out image rather than to its beginning.

The prediction interval can be made very accurate with respect to the system *input* latencies when all data is properly time-stamped from a consistent, system-wide (cross-component), frame of reference for time. If the system is using frame sync, either by querying for the upcoming scan out time or by always commencing rendering just after a scan out, then the prediction interval can also be made very accurate with respect to the *output* latencies. (Because the rendering latencies usually dominate the end-to-end system latencies, and because only the rendering system has access to up-to-date frame sync information, predictive tracking should be done in the rendering portion of the VR system using state vectors passed from earlier stages when frame sync is being used.)

Because portrait-mode display scan-out (where both eyes are on the same display) sequentially scans one eye out and then the other, the prediction time for the right eye may be half a frame time ahead or behind the left eye.

Implementation of predictive tracking within OSVR

OSVR implements predictive tracking inside the code that provides the application with rendering state information (viewport, modelview & projection matrix). It bases this prediction on the sum of three quantities: 1) the time since the most-recent state vector was constructed (the previous tracker report time) 2) the time until the next vertical retrace; and 3) a per-eye value that depends on the hardware being used and includes the sum of the uncompensated tracker latency with the fixed rendering latencies (O/S and driver buffering delays, reformatting delays, and scan-out delays). See [RMPredictFuturePose17] and [RMPredictiveTracking17] for the complete implementation. This code (shown in Listing 32.9) makes use of the Eigen library:

Listing 32.9: Predictive Tracking.

```
// Function called below that performs dead-reckoning orientation estimation.
inline Eigen::Quaterniond applyQuatDeadReckoning(
   Eigen::Quaterniond const& initialOrientation, double angVelDt,
   Eigen::Quaterniond const& velocityDeltaQuat,
   double predictionDistance) {
     Eigen::Quaterniond ret = initialOrientation;
     // Determine the number of integer multiples of our deltaquat needed.
     int multiples = static_cast<int>(predictionDistance / angVelDt);

     // Determine the fractional (slerp) portion to apply after that.
      auto predictionRemainder = predictionDistance - (multiples * angVelDt);
    auto remainderAsFractionOfDt = predictionRemainder / angVelDt;

     Eigen::Quaterniond fractionalDeltaQuat =
            Eigen::Quaterniond::Identity().slerp(remainderAsFractionOfDt, velocityDeltaQuat);

        // Actually perform the application of the prediction.
        for (int i = 0; i < multiples; ++i) {
            ret = velocityDeltaQuat * ret;
        }
        ret = fractionalDeltaQuat * ret;
        return ret;
    }

    // Function called below that predicts a future position and orientation.
    static void PredictFuturePose(
        const OSVR_PoseState &poseIn,
        const OSVR_VelocityState &vel,
        double predictionIntervalSec,
        OSVR_PoseState &poseOut) {

        // Make a copy of the pose state so that we can handle the
        // case where the out and in pose are the same.
        OSVR_PoseState out = poseIn;

        // If we have a change in orientation, make it.
        if (vel.angularVelocityValid) {
            Eigen::Quaterniond newRotation =
                osvr::util::applyQuatDeadReckoning(
                osvr::util::eigen_interop::map(poseIn.rotation),
                vel.angularVelocity.dt,
                osvr::util::eigen_interop::map(vel.angularVelocity.incrementalRotation),
                predictionIntervalSec);
                osvr::util::eigen_interop::map(out.rotation) = newRotation;
        }

        // If we have a linear velocity, apply it.
        if (vel.linearVelocityValid) {
            out.translation.data[0] += vel.linearVelocity.data[0] * predictionIntervalSec;
            out.translation.data[1] += vel.linearVelocity.data[1] * predictionIntervalSec;
            out.translation.data[2] += vel.linearVelocity.data[2] * predictionIntervalSec;
        }

        // Copy the resulting pose.
        poseOut = out;
    }

    ///==============================================================
    /// Inline code starts here, calling the above functions.
    /// Use the state interface to read the most-recent
    /// location of the head.  It will have been updated
    /// by the most-recent call to update() on the context.
    /// DO NOT update the client here, so that we're using the
    /// same state for all eyes.
    OSVR_TimeValue timestamp;
    if (!m_headPoseCache || !m_headPoseCache->getLastReport(timestamp, m_roomFromHead)) {
        // This is not an error -- they may have put in an invalid
        // state name for the head; we just ignore that case.
    }

    // Do prediction of where this eye will be when it is presented
    // if client-side prediction is enabled.
    if (m_params.m_clientPredictionEnabled) {
        // Get information about how long we have until the next present.
        // If we can't get timing info, we just set its offset to 0.
        float msUntilPresent = 0;
        RenderTimingInfo timing;
        if (GetTimingInfo(whichEye, timing)) {
            msUntilPresent +=
              (timing.timeUntilNextPresentRequired.seconds * 1e3f) +
              (timing.timeUntilNextPresentRequired.microseconds / 1e3f);
        }

        // Find out how long ago this tracker info was found.
        float msSinceTrackerReport = 0;
        OSVR_TimeValue now;
        osvrTimeValueGetNow(&now);
        msSinceTrackerReport = static_cast<float>(
                  osvrTimeValueDurationSeconds(&now, &timestamp) * 1e3);

        // The delay before rendering for each
        // eye will be different because they are at different delays past
        // the next vsync.  The static delay common to both eyes has
        // already been added into their offset.
        float predictionIntervalms = msSinceTrackerReport +
              msUntilPresent;
        if (whichEye < m_params.m_eyeDelaysMS.size()) {
            predictionIntervalms += m_params.m_eyeDelaysMS[whichEye];
        }
        float predictionIntervalSec = predictionIntervalms / 1e3f;

        // Find out the pose velocity information, if available.
        // Set the valid flags to false so that if to call to get
        // velocity fails, we will not try and use the info.
        OSVR_VelocityState vel;
        vel.linearVelocityValid = false;
        vel.angularVelocityValid = false;
        if (osvrGetVelocityState(m_roomFromHeadInterface, &timestamp, &vel) !=
            OSVR_RETURN_SUCCESS) {
            // We're okay with failure here, we just use a zero
            // velocity to predict.
            // Using normal get state calls here because we're effectively
            // throwing away the returned timestamp for this data.
        }

        // Predict the future pose of the head based on the velocity
        // information and how long we should predict.  Check the
        // linear and angular velocity terms to see if we should be
        // using each.  Replace the pose with the predicted pose.
        PredictFuturePose(m_roomFromHead, vel, predictionIntervalSec, m_roomFromHead);
    }
```

Time Warp

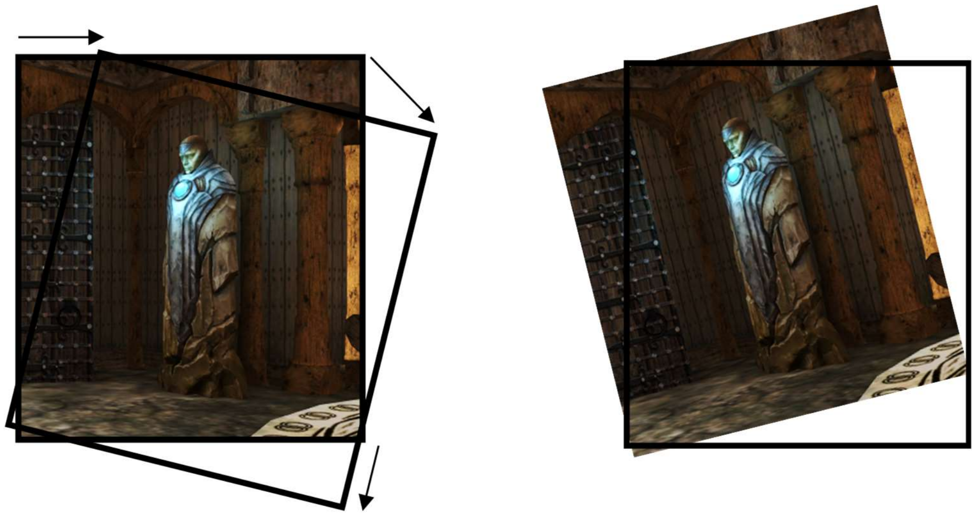


Figure 32.11: Left: *Unity Sun Temple* demo frame drawn for the canonical screen pose at start of rendering and outline of where the canonical screen pose is after head rotation during rendering. **Right:** Canonical screen shown from the viewer's perspective (its motion follows the head, so remains axis aligned) and the rendered image after being time warped by the inverse relative transformation to remain stable in the 3D world.

Because rendering a scene takes time, and because there can be a delay between the end of rendering and the start of display scan-out, the image produced using the head pose that was available when rendering began is not perfectly matched to the pose when that image is presented to the viewer. A solution is to reproject – warp – the original image based on the inverse difference between the original pose and the new pose calculated for the newly estimated time of presentation. The rendered image is thus adjusted to more closely match what should have been rendered had the future pose been known a priori. This section describes how to reduce the impact by re-warping the temporally out-of-date images – time warping.

Fully accurate reprojection of each pixel in the image requires knowledge of its depth because the relative locations of pixels change as the center of projection translates and as the orientation of the projection surface changes. However, much of the viewer's rapid head motion only involves rotation around the center of projection, so a good approximation can be made by projecting the rendered image onto a rectangle in space and then altering that rectangle.

For rotations around the viewing direction, this reprojection is exactly correct. For other rotations and for translations, the quality of the reprojection depends on the distance between the projection plane and the objects in the scene. Reprojections of planar objects aligned with the screen at the same distance used for reprojection will be exactly correct, and objects with other orientations and distances will exhibit some variability; this variability is typically less than the error of the original image, so is still an improvement over using the original, unwarped image. Of course, the less time between rendering start and presentation the less distortion.

To avoid an extra rendering pass, this reprojection can be done by adjusting the transformation used during the distortion correction rendering pass.

Implementation of time warp within OSVR

OSVR adjusts the texture transformation within its vertex shader to enable time warping to be done in the same rendering pass used for the distortion correction (this the reason for inclusion of the separate *textureMatrix* variable in that shader). OSVR keeps track of the rendering poses used to generate each image and reprojects them for each eye using an oversized (see ***Overfill and Oversampling below)*** screen-aligned rectangle projected 2 meters in front of that eye. This transformation is suitable for direct use within OpenGL; for D3D, OSVR adjusts the resulting transformation by inverting Y in two places and transposing the matrix.

The reprojection calculation assumes that it is starting in a texture-coordinate space that has (0,0) at the lower left corner of the image and (1,1) at the upper-right corner of the image, with +Z pointing out of the image. It constructs a transformation from the space used to render into the current-pose space. Next, it moves the points from texture space into world space by scaling and translating them to match a viewport at a given distance in Z from the eyepoint. The points are now in projection space.

The ModelView matrix is then inverted from the last position and applied, moving the points back into world space. The process is then reversed, using the ModelView matrix from the current location (all other matrices are the same) to bring the points back into texture space. It is up to the caller to bring the texture coordinates to and from the space described above (see the ***Overfill and Oversampling*** section for how this is done).

The following code relies on the Eigen library [Eigen17] to do its processing (some error checking has been removed for readability; see [RMATW17] for the complete implementation) as shown in Listing 32.10. This code includes the "*just-in-timewarp"* described below.

Listing 32.10: Computing Time Warps for Each Eye

```
///  @param [in] usedRenderInfo Rendering info used to construct the
/// textures we're going to present.
///  @param [in] currentRenderInfo Rendering info to warp to.
///  @param [in] assumedDepth Depth at which the virtual projected
///  window should be location (defaults to 2 meters)
///  Note that this function is used to compute both synchronous and
///  asynchronous time warps, only the currentRenderInfo changes.
bool RenderManager::ComputeAsynchronousTimeWarps(
    std::vector<RenderInfo> usedRenderInfo,
    std::vector<RenderInfo> currentRenderInfo, float assumedDepth) {

    // See if we're using a D3D11 rendering library.  If so, we need
    // to scale some Y values by -1 and transpose the result. The standard
    // approach works for OpenGL.
    float flipYScale = 1.0f;
    bool doTranspose = false;
    if (dynamic_cast<RenderManagerD3D11Base*>(this)) {
        flipYScale = -1.0f;
        doTranspose = true;
    }

    // Empty out the time warp vector until we fill it again below.
    m_asynchronousTimeWarps.clear();

    size_t numEyes = GetNumEyes();
    if (assumedDepth <= 0) {
        return false;
    }
    if ((currentRenderInfo.size() < numEyes) ||
        (usedRenderInfo.size() < numEyes)) {
        return false;
    }

    for (size_t eye = 0; eye < numEyes; eye++) {
        // Compute the scale to use during forward transform.
        // Scale the coordinates in X and Y so that they match the width and
        // height of a window at the specified distance from the origin.
        // We divide by the near clip distance to make the result match that
        // at a unit distance and then multiply by the assumed depth.
        float xScale = static_cast<float>(
            (usedRenderInfo[eye].projection.right -
             usedRenderInfo[eye].projection.left) /
            usedRenderInfo[eye].projection.nearClip * assumedDepth);
        float yScale = static_cast<float>(
            (usedRenderInfo[eye].projection.top -
             usedRenderInfo[eye].projection.bottom) /
            usedRenderInfo[eye].projection.nearClip * assumedDepth);

        // Compute the translation to use during forward transform.
        // Translate the points so that their center lies in the middle of
        // the view frustum pushed out to the specified distance from the
        // origin.
        // We take the mean coordinate of the two edges as the center that
        // is to be moved to, and we move the space origin to there.
        // We divide by the near clip distance to make the result match that
        // at a unit distance and then multiply by the assumed depth.
        // This assumes the default r texture coordinate of 0.
        float xTrans = static_cast<float>(
            (usedRenderInfo[eye].projection.right +
             usedRenderInfo[eye].projection.left) /
            2.0 / usedRenderInfo[eye].projection.nearClip * assumedDepth);
        float yTrans = static_cast<float>(
            (usedRenderInfo[eye].projection.top +
             usedRenderInfo[eye].projection.bottom) /
            2.0 / usedRenderInfo[eye].projection.nearClip * assumedDepth);
        float zTrans = static_cast<float>(-assumedDepth);

        // NOTE: These operations occur from the right to the left, so later
        // actions on the list actually occur first because we're
        // post-multiplying.

        // Translate the points back to a coordinate system with the
        // center at (0,0);
        const Eigen::Isometry3f postTranslation(
            Eigen::Translation3f(0.5f, 0.5f, 0.0f));

        // Determine the impact of just-in-timewarp in the coordinate system
```

```
// with the center of the screen at the origin and unit width and
// height.  We only do this if just-in-timewarp is enabled; otherwise,
// we set this to the identity matrix.
Eigen::Matrix<float, 4, 4> justInTimeWarp;
justInTimeWarp.setIdentity();
if (m_params.m_justInTimeWarp) {
    std::array<float, 4> coeffs = ComputeJustInTimeWarp();
    const float &xScale = coeffs[0];
    const float &yScale = coeffs[1];
    const float &xShearWithY = coeffs[2];
    const float &yShearWithX = coeffs[3];
    justInTimeWarp(0, 0) = xScale;
    justInTimeWarp(1, 1) = yScale;
    if (doTranspose) {
        justInTimeWarp(0, 1) = xShearWithY;
        justInTimeWarp(1, 0) = yShearWithX;
    } else {
        justInTimeWarp(1, 0) = xShearWithY;
        justInTimeWarp(0, 1) = yShearWithX;
    }
}

/// Scale the points so that they will fit into the range
/// (-0.5,-0.5)
/// to (0.5,0.5) (the inverse of the scale below).
const Eigen::Affine3f postScale(
    Eigen::Scaling(1.0f / xScale, flipYScale / yScale, 1.0f));

/// Translate the points so that the projection center will lie on
/// the -Z axis (inverse of the translation below).
const Eigen::Isometry3f postProjectionTranslate(
    Eigen::Translation3f(-xTrans, -yTrans, -zTrans));

/// Compute the forward last ModelView matrix.
const Eigen::Isometry3f lastModelView = osvr::util::eigen_interop::map(
    usedRenderInfo[eye].pose).transform().cast<float>();
Eigen::Isometry3f lastModelViewTransform(lastModelView);

/// Compute the inverse of the current ModelView matrix.
const Eigen::Isometry3f currentModelViewInverseTransform =
    osvr::util::eigen_interop::map(
    currentRenderInfo[eye].pose).transform().cast<float>().inverse();

/// Translate the origin to the center of the projected rectangle
Eigen::Isometry3f preProjectionTranslate(
    Eigen::Translation3f(xTrans, yTrans, zTrans));

/// Scale from (-0.5,-0.5)/(0.5,0.5) to the actual frustum size
Eigen::Affine3f preScale(Eigen::Scaling(xScale, flipYScale * yScale, 1.0f));

// Translate the points from a coordinate system that has (0.5,0.5)
// as the origin to one that has (0,0) as the origin.
Eigen::Isometry3f preTranslation(
    Eigen::Translation3f(-0.5f, -0.5f, 0.0f));

/// Compute the full matrix by multiplying the parts.
Eigen::Projective3f full =
    postTranslation * justInTimeWarp * postScale * postProjectionTranslate *
    lastModelViewTransform * currentModelViewInverseTransform *
    preProjectionTranslate * preScale * preTranslation;
```

```
        // Store the result, transposing if we're using D3D.
        matrix16 timeWarp;
        if (doTranspose) {
            Eigen::Matrix4f::Map(timeWarp.data) = full.matrix().transpose();
        } else {
            Eigen::Matrix4f::Map(timeWarp.data) = full.matrix();
        }
        m_asynchronousTimeWarps.push_back(timeWarp);
    }
    return true;
}
```

Asynchronous Time Warp (ATW)

Due to scene complexity, O/S interrupts, or other causes the rendering process sometimes takes more time than a single scan out interval. For non-immersive displays, this can introduce jerkiness in playback; in immersive VR it also introduces a doubled image when the viewer's head pose is changing. To avoid these artifacts, the VR system can re-warp and re-display the previously presented image based on updated tracking information at the time the next frame needs to be displayed.

The warping function is the same as for standard time warp and does the same adjustment based on the difference between the viewer's pose at the time the image began rendering and the current pose.

To ensure that a new warped image is available each frame, asynchronous time warp must use *frame sync* and it must launch a separate rewarping thread that keeps the most-recently-presented image and use that to warp and present just ahead of display scan-out. This thread should have real-time priority in both the operating system and on the GPU. (To enable interruption of long-running renders, it must make use of vendor-specific APIs to enable pre-emptive rendering.) On operating systems such as Windows 10 with coarse sleep-return temporal granularity (e.g. 10ms or more), it may be necessary to busy-wait on the time before refresh to avoid missing updates.

To ensure that basic scene rendering has fully completed prior to attempting the last-millisecond final rendering pass, the application thread must use a rendering-library-specific call to ensure that all operations are complete and the texture is ready for use before handing it to the rewarping thread.

Because the rewarping thread must always have a texture containing the basic scene ready, it must either make a copy of the texture presented by the application or the application must use double buffering and not modify the texture that was most recently presented; it must only modify a texture after presenting a different texture for display.

(Note that asynchronous time warp is only correct for non-moving objects in the scene. Moving objects will have shifted positions between the beginning and end of rendering, and applying this time warp to them will produce artifacts like those produced by jitter to those objects, similar to how this happens to the entire scene when rendered without frame sync.)

Implementation of asynchronous time warp within OSVR

Asynchronous time warp is implemented in the *Sensics OSVR-RenderManager's* ATW renderer [OSVRRenderManager17]. As of March 2018, it is implemented only for *direct mode* interfaces using the *Direct 3D* graphics library because these are the only ones that currently support frame sync but it on Windows it is wrapped using the OpenGL Interop libraries to provide ATW for OpenGL as well.

OSVR constructs a completion-query event when the renderer is opened and uses it to ensure that rendering to the texture completes before passing it to the rewarping thread (see [OSVRRMD3DBase17] for the complete implementation) as shown in Listing 32.11:

Listing 32.11: Ensuring Rendering Completion

```
// Constructed during initialization and re-used during rendering
D3D11_QUERY_DESC desc = {};
desc.Query = D3D11_QUERY_EVENT;
m_D3D11device->CreateQuery(&desc, &m_completionQuery);

// Using the query each time through the rendering loop
m_D3D11Context->End(m_completionQuery);
m_D3D11Context->Flush();
while (S_FALSE == m_D3D11Context->GetData(m_completionQuery, nullptr, 0, 0)) {
   // We don't want to miss the completion because Windows has
   // swapped us out, so we busy-wait here on the completion
   // event.
}
```

A rewarping thread in the ATW RenderManager class uses a second RenderManager to do the actual rendering. It internally keeps track of either copying buffers or locking shared buffers before handing them to the rendering thread.

As of March 2018, pre-emptive GPU scheduling is only available within OSVR on nVidia Pascal-series cards (eg. GeForce 1080), and only with GeForce driver version 372.54 or later. In other cases, ATW cannot pre-empt a long-running render thread. This means that a long-running rendering thread will block access to the GPU and prevent the rendering thread from gaining access, causing it to miss frames.

Just In Time Warp (AKA Beam Racing, Just-In-Time Pixels, Intra-Frame Warp)

Many head-mounted displays scan the visible pixels from one end of the display to the other, thus pixels at the bottom line are rendered almost a full cycle behind the pixels at the top. Because the images produced by the application are rendered at a single point in time, head motion during the frame causes spatial misalignment between what should be seen and the rendered scene. For example, the image of a cube rendered in a frame where the viewer's head is rapidly rotating from the left to the right should show the lower portions of the cube to the left of the higher portions because the head has moved since the upper pixels were displayed, yet with standard rendering are directly below them. This makes the perceived object seem to be slanted towards the left.

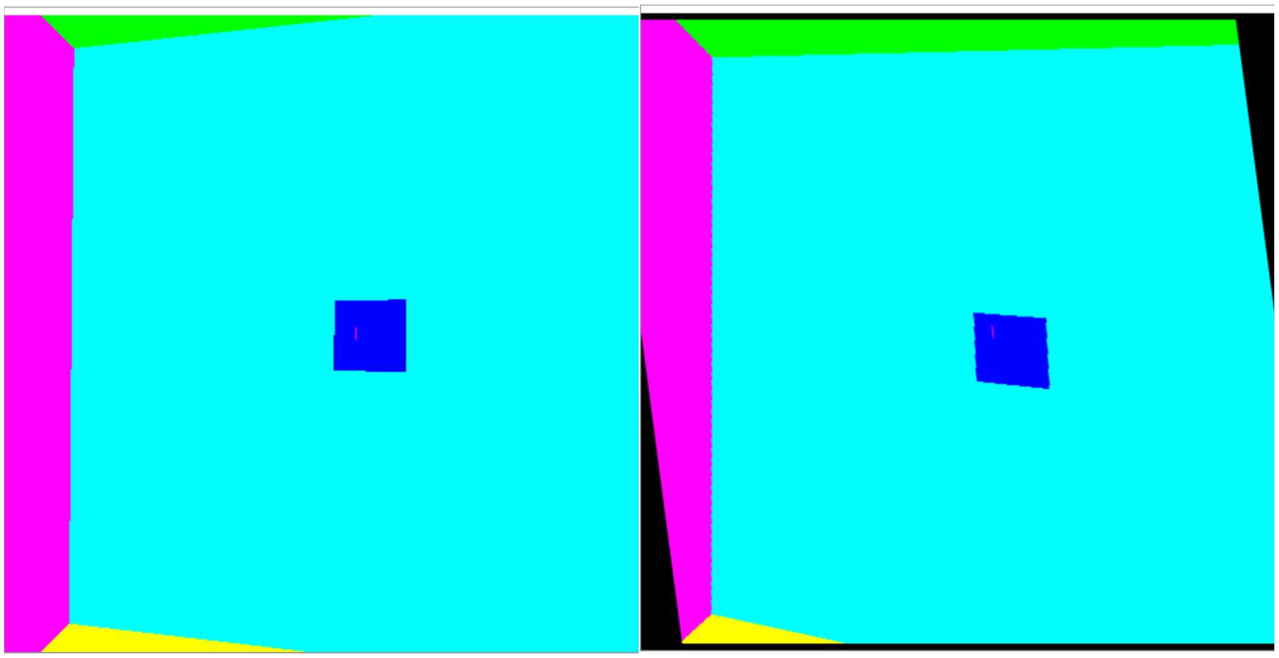


Figure 32.12: Left: An image of a small dark blue cube within a colored cube room taken with the head held still shows the cube to be square. Right: A just-in-time rendering of this same scene with the head rapidly moving shows shearing in X and a scale reduction in Y to compensate for changes in location of the scan lines during scan-out. Note that the still image is deceiving because when you look at it on paper it's entirely "rendered" at the same time and your head isn't moving – when viewed from within the moving HMD, the scene appears normal; without just-in-time rendering, the cube appears to be stretched in Y and sheared in the opposite direction. (As for the black regions, see the section on **Overfill and Oversampling** below.)

Noting that "The ideal way to generate an image […] would be to recalculate for each pixel the position and orientation of the camera and the position and orientation of the scene's objects, based upon the time of display of that pixel" Olano et al. propose "*Just-in-time-pixels*" [Olano95]. Because of the expense of re-rendering each scene, they propose an approximation of determining the correct transformation for the first and last scan lines in an image and using linear interpolation for object locations in the scan lines between them. Figure 12 shows this implementation in action on a simple test scene.

Implementation of intra-frame time warp within OSVR

Observing that the largest distortion due to head motion is often caused by rotation of the head in the vertical or horizontal planes and further noting that affine transformations can be readily applied during the rendering pass (the time warp implementation already includes a general 4x4 matrix multiplication), OSVR-RenderManager can easily approximate the impact of these transformations at negligible increased rendering cost by adding anisotropic scaling and shearing to the time-warp texture reprojection matrix. OSVR predicts the viewing time for each eye in the center of the viewing area and distorts other image regions based on linear horizontal and vertical rotational velocity estimates. As is the case with regular time warp of planar-projected images, these transformations are approximations that work better for small temporal differences.

The case where the display scans out from top to bottom and the viewer's head is rotating from right to left and slightly downwards matches the case shown in figure 32.12, where the lower portion of the square must be offset to the right with respect to the center of the image so that it will be drawn at a location in physical space that is directly below its top (it is drawn later, and the head has rotated to the left). The distortion is compensated for by adding shearing to the texture reprojection matrix which causes it to sample texture locations increasingly to the right as the image is scanned out from top to bottom. The amount of shear is selected that causes a vertical line drawn at the center of the image to appear to remain vertical in the presence of the estimated rotational speed.

Figure 12 also shows the case where the display scans out from top to bottom and the viewer's head is rotating downward causes the bottom of the cube to appear to be drawn lower in physical space, causing it to appear to stretch vertically. To offset this, an anisotropic scaling is performed in the vertical direction, where X coordinates are left unchanged but Y coordinates are adjusted to compensate for the perceived stretching by shrinking the cube vertically. For upwards head rotation, the Y coordinates are stretched. The scaling factor is selected that results in no vertical stretching or squashing for a pixel located at the center of the image.

The two transformations are orthogonal for small motions, so can be safely applied independently of one another.

For an HMD whose screen is mounted upside down (at a display rotation of 180 degrees), the distortions described above are inverted – downward head motion causes apparent squashing and upward motion causes apparent stretching. This case can be handled by inverting the change in X and Y positions. For the case of displays that scan out from right to left and left to right, the shearing and stretching operations are swapped. The general case can be treated as a rotation about the +Z axis (which comes out of the image), transforming from the (X,Y) coordinate system to a (shear, scale) coordinate system with the signs of the scaling and shearing factors determined by the rotation. Because the display orientation in the operating system and the display scan-out may not be related, a configuration-file entry declares from which border of the HMD screen scan-out commences.

Listing 32.10 included the construction of the shearing and stretching transformations, which rely on the function shown in Listing 32.12 to compute the appropriate amount and orientation of the shear and anisotropic scaling.

Listing 32.12: Determining Just In Time Warp coefficients

```
/// This function computes the coefficients of nonuniform scaling
/// and shearing required to implement just-in-timewarp in a space
/// where (0,0) is the center of the screen and the screen width and
/// height are both 1 (dimensions go from -0.5 to 0.5 in each axis).
/// It first computes the velocity, then based on that and the rotation
/// of the scan-out with respect to the image produces the four values.
/// It does not check to see whether just-in-timewarp is enabled.
/// @return Four doubles, indicating: 0th = the scaling
```

```
    ///         of the image in the X direction, 1st = the scaling
    ///         of the image in the Y direction, 2nd = the shear in
    ///         X coordinate as Y varies, 3rd = the shear in the Y
    ///         coordinate as X varies.  At most one of the scalings and
    ///         at most one of the shear transformations will be active
    ///         at a time; which ones are active depends on the orientation.
    std::array<float, 4> RenderManager::ComputeJustInTimeWarp() {
        // We initialize the values with ones that won't cause any
        // change.  We will override them as we find reason.
        std::array<float, 4> ret = { 1, 1, 0, 0 };

        // Figure out which edge of the display scan-out starts at based
        // on the just-in-time rotation.  This describes which edge will
        // be rotated to point "up" when the display is rotated about the
        // +Z axis (out of the screen) and it starts at the canonical orientation
        // with X to the right and Y up.  The four results are 0 = top, 1 = right,
        // 2 = bottom, 3 = left.  The code rounds to the nearest one.
        int edgeUp = static_cast<int>(
            floor((m_params.m_justInTimeWarpRotation + 44.9999) / 90));
        if (edgeUp < 0) { edgeUp += 4 * static_cast<int>(1 - edgeUp / 4); }
        edgeUp = edgeUp % 4;

        // Find out the timing information, which will let us know the
        // duration of a full-screen scan-out.  If we are scanning out
        // from left to right or right to left, divide this by the number
        // of eyes per display to find the per-eye scan-out duration.
        RenderTimingInfo timing;
        if (!GetTimingInfo(0, timing)) {
            // If we have no timing information, then we have nothing to use
            // to predict so we return the do-nothing result.
            return ret;
        }
        double screenScanTime = (timing.hardwareDisplayInterval.seconds +
            timing.hardwareDisplayInterval.microseconds / 1e6);
        if (edgeUp % 2 == 1) {
            screenScanTime /= GetNumEyesPerDisplay();
        }

        // Find out the pose velocity information, if available.
        // Set the valid flags to false so that if to call to get
        // velocity fails, we will not try and use the info.
        OSVR_TimeValue timestamp;
        OSVR_VelocityState vel;
        vel.linearVelocityValid = false;
        vel.angularVelocityValid = false;
        if (osvrGetVelocityState(m_roomFromHeadInterface, &timestamp, &vel) !=
              OSVR_RETURN_SUCCESS) {
            // No velocity information available, so we return the do-nothing result.
            return ret;
        }

        // Convert the incremental orientation change in world space back
        // into (local) head space by transforming it by the inverse of the
        // current head pose.
        //  Handle a non-Identity room-from-world transform in the OSVR-Core
        // room-to-world transform (as opposed to the RenderManager one, which is
        // already handled because we apply that transformation ourselves).  We
        // do this by getting and applying the room-to-world transform from Core
        // here.  Again, we can ignore the RenderManager room-to-world that was
        // passed in as RenderParam because all of our differential transform
        // work here takes place below it.
        OSVR_PoseState pose;
        if (osvrGetPoseState(m_roomFromHeadInterface, &timestamp, &pose) != OSVR_RETURN_SUCCESS)
{
            // No pose information available, so we return the do-nothing result.
            return ret;
        }
        osvr::common::Transform xform(ei::map(pose).matrix(), ei::map(pose).matrix().inverse());
        xform.transform(m_context->getRoomToWorldTransform());
        Eigen::Quaterniond localRot = xform.transformDerivative(
            ei::map(vel.angularVelocity.incrementalRotation));

        // Turn incremental rotation into Euler rotation rates in radians/second.
        // Do this by converting the Quaternion into Euler and then dividing by the
        // delta time.  We do this twice, once with the X axis being defined as the
        // last axis to be rotated around and once with the Y axis as the last. (The
        // last axis is the first listed in right-to-left multiplication.)  If we
        // use the same Euler set for more than one angle, sometimes we get flips by
        // Pi around the axes.
        const double &dt = vel.angularVelocity.dt;
        Eigen::Vector3d euler = localRot.toRotationMatrix().eulerAngles(0, 1, 2);
        if (euler[0] > boost::math::double_constants::pi / 2) {
            // Rotation around first axis is always positive when returned from eulerAngles;
switch
            // the second quadrant into the fourth so that we get symmetry around 0.
            euler[0] -= boost::math::double_constants::pi;
        }
        double rX = euler[0] / dt;
        euler = localRot.toRotationMatrix().eulerAngles(1, 2, 0);
        if (euler[0] > boost::math::double_constants::pi / 2) {
            // Rotation around first axis is always positive when returned from eulerAngles;
switch
            // the second quadrant into the fourth so that we get symmetry around 0.
            euler[0] -= boost::math::double_constants::pi;
        }
        double rY = euler[0] / dt;

        // Determine the amount of rotation around the X axis in degrees that takes
        // place during the eye scan-out time.  Do the same for Y.
        double xRotationDegrees = screenScanTime * osvr::common::radiansToDegrees(rX);
        double yRotationDegrees = screenScanTime * osvr::common::radiansToDegrees(rY);

        /// Determine the fraction of the display width in angles in X that will be
        /// covered by this rotation around Y over the course of the frame.  Do the same for
        /// the fraction of the height in angles in Y that will be covered by rotation
        /// about X.  Leave these signed, so that we know whether to rotate in the positive
        /// or negative direction.
        float xRotationNormalized = static_cast<float>(xRotationDegrees /
              osvr::util::getDegrees(m_params.m_displayConfiguration->getHorizontalFOV())
          );
        float yRotationNormalized = static_cast<float>(yRotationDegrees /
              osvr::util::getDegrees(m_params.m_displayConfiguration->getVerticalFOV())
          );

        // Based on the scan-out direction, adjust the relevant output parameters
        // to indicate the amount of scaling and shearing that will take place over
        // an eye scan-out time.
        switch (edgeUp) {
        case 0: // Top up.
            // As the head rotates around +X, we get stretching in Y.
            // To compensate, we need to scale Y down when rotating in +X.
            ret[1] = 1 - xRotationNormalized;

            // As the head rotates around +Y, we get shearing in +X with increasing Y.
            // To compensate, we need to shear in X based on -Y.
            ret[2] = -yRotationNormalized;
            break;

        case 1: // Right up
            // As the head rotates around -Y, we get stretching in X.
            // To compensate, we need to scale X down when rotating in +Y.
            ret[0] = 1 + yRotationNormalized;

            // As the head rotates around +X, we get shearing in -Y with increasing X.
            // To compensate, we need to shear in Y based on X.
            ret[3] = xRotationNormalized;
            break;

        case 2: // Bottom up
            // As the head rotates around +X, we get compression in Y.
            // To compensate, we need to scale Y up when rotating in +X.
            ret[1] = 1 + xRotationNormalized;

            // As the head rotates around +Y, we get shearing in -X with increasing Y.
            // To compensate, we need to shear in X based on Y.
            ret[2] = yRotationNormalized;
            break;

        case 3: // Left up
            // As the head rotates around Y, we get stretching in X.
            // To compensate, we need to scale X up when rotating in +Y.
            ret[0] = 1 - yRotationNormalized;

            // As the head rotates around +X, we get shearing in Y with increasing X.
            // To compensate, we need to shear in Y based on -X.
            ret[3] = -xRotationNormalized;
            break;
        }

        return ret;
    }
```

Direct Rendering (aka Direct Mode, Direct-to-Display)

To support transparent borders and other user-interface effects, some operating systems store each rendered frame before compositing it onto the display, which adds a frame of latency. To improve throughput, some graphics-card drivers keep two or more frames in the pipeline, with CPU rendering completing more than a frame sooner than the image will be presented to the display. Both approaches add to the end-to-end latency for VR systems. This section describes how to avoid these delays using direct rendering.

Vendor-specific APIs have been provided by nVidia and AMD to bypass the operating system and render directly to the display device. (A vendor-independent approach is being implemented within a new Microsoft API as well.) Each of these approaches also offers control over the number of buffers and their presentation to the display surface, enabling either frame asynchronous or frame synchronous swapping of buffers and determination of the time at which vertical retrace happens. This enables front-buffer rendering, but also double-buffer rendering where the buffers are swapped just before vertical retrace, thus providing the combined benefit of extended render times together with low-latency presentation.

Within-Display Buffering

A similar effect can happen inside the display devices themselves. Many devices will support taking images in either landscape or portrait mode and support flipping the scan-out upside down in either mode. Of course, these displays natively scan out in only one particular direction (often portrait mode, starting at the right side of the display as mounted in an HMD) so to flip the image they must first internally buffer a whole frame before starting scan out.

Determining which orientation is preferred requires reading manufacturer specifications or careful testing with a sensitive latency meter. Once determined, best performance is achieved by driving the display in the native mode and doing any required frame flipping within the VR system's final rendering pass.

Graphics-Language Interoperability

On Windows 10, Direct Rendering is only available for the Direct3D graphics library and not for OpenGL. On upcoming Linux interfaces, it may be available only on Vulkan. Accessing these capabilities from OpenGL or other languages requires sharing image buffers between graphics libraries, either using the nVidia NV_DX_interop interface [nVidia10], using shared handles or using Khronos EGL buffers [Khronos17].

In these cases, rendering is performed in one rendering library and then the buffers are shared with the Direct-Rendering-capable library and it presents them to the display. These approaches require an additional flushing of the graphics commands to GPU before passing control of the buffers between libraries to ensure that all rendering dependencies are met.

Note that different graphics libraries have different coordinate systems (or different defaults that are used by their communities). For example, OpenGL and Direct3D use different origins for texture coordinates, with OpenGL using the lower left corner and Direct3D the upper left. This requires adjustments to be made when sharing buffers between libraries.

Implementation of direct rendering within OSVR

Because the individual vendor APIs are only available under nondisclosure agreements, the *Sensics OSVR-RenderManager* library implements *RenderManager* interfaces for them and distributes them with OSVR-built DLLs but cannot release the source code for these drivers. To maximize the portion of the code using open source, all techniques using DirectMode: *Asynchronous Time Warp*, *OpenGL*

Interoperability, *Frame Sync*, and even the interface that applications use to control *Direct Rendering* are implemented by either harnessing a Direct Rendering RenderManager or are implemented within it using the same RenderManager interface used by the open-source drivers.

OSVR-RenderManager uses OpenGL Interop to share buffers between an application OpenGL context (Legacy or Core) and the Direct3D context used for display. It handles the buffer flipping and coordinate transformations needed to translate images, distortion correction, and time warps between the systems. It does this by providing a *RenderManagerD3DOpenGL* class [RMD3DOpenGL16].

OSVR-RenderManager also handles the image flipping required to avoid ***Within-Display Buffering***, as well as providing the option to drive portrait or landscape displays mounted at any orientation. It provides transformations to the application so that it can render the images as if they were right-side-up (enabling text, sprites, and other pixel-aligned techniques to work properly) and then re-orienting the image as needed to meet display needs. [RMRotateViewport17] [RMConstructModelView17]

Overfill and Oversampling

Time warp reprojects an image from a different viewpoint. Normally, the original image could be rendered to exactly cover the canonical screen; however, reprojection causes the new viewpoint to see past those original borders. This produces black borders creeping in from the edges. Distortion correction can produce a similar effect when the canonical screen does not completely fill the display, resulting in similar borders. Both issues can be addressed using *Overfill — i.e.* rendering an image that goes beyond the edges of the canonical screen. This section describes how to use these approaches to remove rendering artifacts.

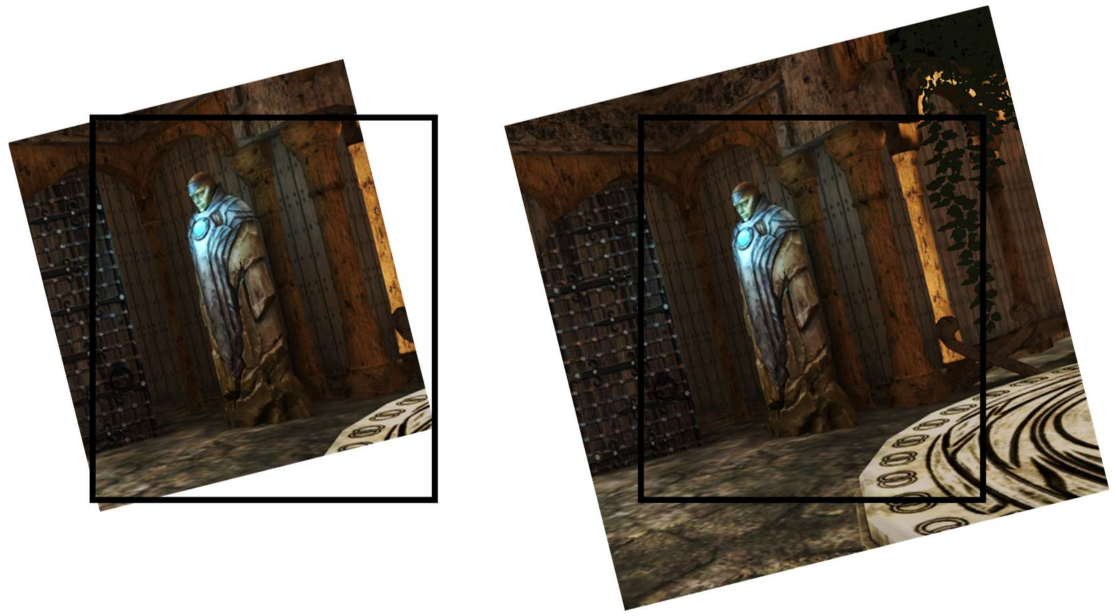


Figure 32.13: Left: Time-warped frame from *Unity Sun Temple* demo reveals borders around the image that were not rendered. **Right:** By rendering past the borders of the canonical screen, there is now enough image to cover the entire new viewport.

Overfill requires adjustment of both the projection transformation (making the projection region wider) and the graphics viewport size in pixels (providing a place to store the extra pixels); the viewing transformation remains the same. The size of overfill needed to hide the borders depends on the distortion correction being done, on the length of time between rendering and warping, and on the speed of rotation of the viewer's head: faster rotation reveals more border per unit time.

Distortion correction will, by definition, increase the visible size of some regions on the canonical screen and decrease the size of others. If the rendered image has as many texture elements as there are pixels on the display, then some regions will be expanded such that there are more physical display pixels than available texture elements, producing images that sacrifice the potential sharpness of the display. This can be addressed using *Oversampling — i.e.* rendering an image that has more texture elements than the display has pixels.

Oversampling requires adjustment of only the pixel size of the graphics viewport; the viewing and projection transformations remain the same. The amount of oversampling required depends on the largest magnification caused by distortion correction.

Oversampling can also be used in the opposite direction, reducing the number of texture elements compared to the number of display pixels, to increase the rendering rate for applications that have large amount of per-pixel processing. This trades off reduced image resolution for increased rendering speed.

Implementation of overfill and oversampling within OSVR

The *Sensics OSVR-RenderManager* library implements both overfill and oversampling, taking them into account when generating the projection transformation and when generating the viewport description. Overfill is handled in the projection transformation by specifying a fractional increase in size, which is then used to expand the projection as shown in Listing 32.13:

Listing 32.13: Overfill handling in projection transformation calculation.

```
double xMargin = width / 2 * (m_params.m_renderOverfillFactor - 1);
double yMargin = height / 2 * (m_params.m_renderOverfillFactor - 1);
left -= xMargin;
right += xMargin;
top += yMargin;
bottom -= yMargin;
```

This expansion must be inverted in the code that renders to the graphics library so that only the correct fraction of the image is visible within the resulting displayed frame. This is handled in the OpenGL code path by adjusting the *projectionMatrix* entry in the vertex shader shown in Listing 32.14:

Listing 32.14: Overfill handling in vertex shader projection.

```
m_projectionUniformId = glGetUniformLocation(m_programId, "projectionMatrix");

GLfloat myScale = m_params.m_renderOverfillFactor;
GLfloat scaleProj[16] = { myScale,0,0,0,   0,myScale,0,0,   0,0,1,0,   0,0,0,1 };
glUniformMatrix4fv(m_projectionUniformId, 1, GL_FALSE, scaleProj);
```

Along with oversampling, overfill is used to expand the viewport size as shown in Listing 32.15:

Listing 32.15: Overfill handling in viewport calculation.

```
viewport.width = xFactor * m_displayWidth * m_params.m_renderOverfillFactor * m_params.m_renderOversampleFactor;
viewport.height = yFactor * m_displayHeight * m_params.m_renderOverfillFactor * m_params.m_renderOversampleFactor;
```

The expansion in viewport must be taken into account in the code that handles time warp and distortion correction so that it maps the standard texture coordinate range (0,0)-(1,1) into the portion of the texture that represents the canonical screen. This approach supports the expansion of the range within the overfilled viewport [RMCorrectCoord17] as shown in Listing 32.16:

Listing 32.16: Overfill Support in Distortion Correction.

```
/// Takes a texture coordinate that is specified in the coordinate system of
/// a Presented texture for a given eye, which has (0,0) at the lower left
/// and (1,1) at the upper right.  The lower left and upper right are at the
/// boundaries specified by the overfill rectangle, which are not visible
/// for overfill factors > 1.
/// @param eye eye to get coordinates for
/// @param inCoords coordinates to modify
/// @param distort distortion parameters
/// @param color red=0, green=1, blue=2
/// @param overfillFactor scaling factor to allow for timewarp
/// @param interpolators list of unstructured mesh interpolators
using Float2 = std::array<float, 2>;
inline Float2 OSVR_RENDERMANAGER_EXPORT DistortionCorrectTextureCoordinate(
  const size_t eye, Float2 const& inCoords,
  const DistortionParameters& distort, const size_t color,
  const float overfillFactor,
  const std::vector< std::unique_ptr<UnstructuredMeshInterpolator> >& interpolators) {
  // Check for invalid parameters
  if (color > 2) {
    return inCoords;
  }

  // Convert from coordinates in the overfilled texture to coordinates
  // that will cover the range (0,0) to (1,1) on the screen.  This is
  // done by scaling around (0.5,0.5) to push the edges of the screen
  // out to the (0,0) and (1,1) boundaries.
  using Eigen::Vector2f;
  using Eigen::Map;
  const auto inMap = Map<const Vector2f>(inCoords.data());

  Vector2f xyN = (inMap - Vector2f::Constant(0.5f)) * overfillFactor +
               Vector2f::Constant(0.5f);
  const float xN = xyN.x();
  const float yN = xyN.y();

  const auto normalized_inCoords = Float2{xN, yN};
```

```
    Float2 ret = DistortionCorrectNormalizedTextureCoordinate(
      eye, normalized_inCoords, distort, color, interpolators);

    // Convert from unit (normalized) space back into overfill space.
    ret[0] = (ret[0] - 0.5f) / overfillFactor + 0.5f;
    ret[1] = (ret[1] - 0.5f) / overfillFactor + 0.5f;

    return ret;
  }
```

Rendering State

VR systems take time-varying, linear and nonlinear geometric descriptions of the relative locations and orientations of objects in space and produce descriptions suitable for implementation in the linear geometric operations available in graphics libraries. This section describes how to manage this state across graphics libraries.

The resulting linear operations can be implemented in various rendering systems, including basic graphics libraries (OpenGL, Direct3D, GLES, Vulkan, etc.), game engines (Unreal, Unity, Blender, etc.) and others (VTK, OpenCV, etc.). These systems have a variety of distance units (meters, mm, pixels, etc.) and coordinate systems (right-handed vs. left-handed, screen lower-left vs. upper-left, etc.). This means that no single internal representation can be used within a VR system that is to be implemented across multiple rendering systems. It also means that all aspects of the coordinate system must be carefully described because they will be unfamiliar to users of some of the rendering systems.

The variety of coordinate systems requires that for a VR system to be easily adapted between rendering systems it must either provides adapters for each rendering system or else use conditional compilation or wrappers to behave differently when used with different systems.

Time

The proper spatial alignment of rendered viewpoints with objects that remain stationary in the real world is required to prevent “swimming” of the virtual world around the viewer. This is even more important in augmented reality systems, where overlaid virtual objects must remain aligned with their real-world counterparts.

This alignment requires a level of timing accuracy that is beyond the needs of most non-immersive 3D graphics displays. Combining multiple devices, and sometimes multiple computers, in the collection of tracking data (and sometimes video data to integrate the real world) can make accurate timing difficult. The Network Time Protocol (NTP) [NTP17] can be tuned to achieve submillisecond agreement among a small number of computers on the same network. Properly-aligned, submillisecond-precise clocks between processes have recently become common on some operating systems and compilers.

USB interfaces, video cameras, network drivers, and other drivers within an operating system often enable high throughput and offload work from the CPU by providing buffering and a separation of fast kernel-level drivers from slower user-level drivers. This can introduce both latency and jitter in the time between a physical measurement on a device and the presentation of that measurement to the system. Reducing this can require adjustment of system scheduling intervals, tuning parameters on network connections, raising the priority of processes that handle devices, and busy-waiting on inputs rather than letting the operating system release data to the system at its usual intervals [VRPN01]. It can also require back-dating the time associated with measurements based on the known capture and transmission time [VRPN01].

To enable consistency between all portions of a VR system, each event and measurement should be time stamped. This enables comparison and proper relative dating of all measurements within the system, producing a common frame of reference.

Implementation of rendering state within OSVR

OSVR-Core associates timing information with all system events and measurements and uses busy-waiting on actively-used devices to ensure low-latency measurement and data transport. Its internal end-to-end latencies for device measurement, estimation, and reporting are considerably submillisecond. When compiled using Visual Studio 2015 or higher on Windows, and on all other operating systems, it provides submillisecond-accurate consistent clocks across processes on a single computer; it relies on NTP to maintain accuracy between computers.

The *Sensics OSVR-RenderManager* provides graphics-language-specific (OpenGL, Direct3D, Unity, Unreal) conversion functions to describe the number and size of required textures, viewports, projection and ModelView matrices needed to configure rendering for scenes [RMGLD3D16] [RMGLGL16]. The RenderManager receives all viewports and textures in their canonical (viewer up is texture up) orientation and internally maps everything to the correct orientation, enabling the use of bitmap fonts and other rendering effects that require canonical orientation. An optional, callback-based rendering path provides these transformations for arbitrary OSVR spaces (head space, hand space, room space, etc.).

The *Sensics OSVR-RenderManager* manages the display-orientation remapping using *Modelview* matrices within the vertex shaders for each of its rendering paths. It internally keeps track of any rotation required by the display and any flipping required by the rendering library compared to the OSVR internal coordinate system. The following routine uses this information to produce a generic matrix that each rendering path then copies into the matrix used by its shader as shown in Listing 32.17:

Listing 32.17: Display-Orientation Remapping.

```
bool RenderManager::ComputeDisplayOrientationMatrix(
    float rotateDegrees, //< Rotation in degrees around Z
    bool flipInY,//< Flip in Y after rotating?
    matrix16& outMatrix //< Matrix to use.
    ) {

    /// Scale the points to flip the Y axis if that is called for.
    float yScale = 1;
    if (flipInY) { yScale = -1; }
    Eigen::Affine3f preScale(Eigen::Scaling(1.0f, yScale, 1.0f));
```

```
        // Rotate by the specified number of degrees.
        Eigen::Vector3f zAxis(0, 0, 1);
        float rotateRadians = static_cast<float>(rotateDegrees * M_PI / 180.0);
        Eigen::Affine3f rotate(Eigen::AngleAxisf(rotateRadians, zAxis));

        /// Compute the full matrix by multiplying the parts.
        Eigen::Projective3f full = rotate * preScale;

        // Store the result.
        memcpy(outMatrix.data, full.matrix().data(), sizeof(outMatrix.data));

        return true;
    }
```

Conclusion

The geometry-critical and time-critical rendering needs in virtual reality require the concerted use of a suite of techniques beyond those applied in non-immersive interactive computer graphics systems. This chapter describes each of those needs and provides example code to implement them based on the OSVR system, which itself is an open-source solution that implements all of them working together.

References

[Robinett92] Robinett, W. and R. Holloway (1992). Implementation of Flying, Scaling, and Grabbing in Virtual Worlds. Proceedings of the ACM Symposium on Interactive 3D Graphics, Cambridge, MA, ACM SIGGRAPH.

[OSVRView16] https://github.com/OSVR/OSVR-Docs/blob/master/Configuring/projectionAndViewMatrices.md

[OSVRDistort16] https://github.com/OSVR/OSVR-Docs/blob/master/Configuring/distortion.md

[OSVRAngles17] https://github.com/OSVR/distortionizer/tree/master/angles_to_config

[OSVRRenderManager17] https://github.com/sensics/OSVR-RenderManager

[OSVRRMD3DBase17] https://github.com/sensics/OSVR-RenderManager/blob/master/osvr/RenderKit/RenderManagerD3DBase.cpp

[Holloway95] Holloway, R. L. (1995). Registration Errors in Augmented Reality Systems. Computer Science. Chapel Hill, The University of North Carolina. http://www.cs.unc.edu/techreports/95-016.pdf

[SDL17] https://www.libsdl.org

[Azuma95] Azuma, R. (1995). Predictive Tracking for Augmented Reality. Computer Science. Chapel Hill, The University of North Carolina. http://www.cs.unc.edu/techreports/95-007.pdf

[RMPredictFuturePose17] https://github.com/sensics/OSVR-RenderManager/blob/3458fb7ac8948215026ac416a3aa6cec4320e6af/osvr/RenderKit/RenderManagerBase.cpp#L118

[RMPredictiveTracking17] https://github.com/sensics/OSVR-RenderManager/blob/3458fb7ac8948215026ac416a3aa6cec4320e6af/osvr/RenderKit/RenderManagerBase.cpp#L1503

[RMD3DOpenGL16] https://github.com/sensics/OSVR-RenderManager/blob/master/osvr/RenderKit/RenderManagerD3DOpenGL.cpp

[RMRotateViewport17] https://github.com/sensics/OSVR-RenderManager/blob/3458fb7ac8948215026ac416a3aa6cec4320e6af/osvr/RenderKit/RenderManagerBase.cpp#L1317

[RMConstructModelView17] https://github.com/sensics/OSVR-RenderManager/blob/397e4374ca7a04f7113edef680b39241bb3e0101/osvr/RenderKit/RenderManager.h#L975

[RMCorrectCoord17] https://github.com/sensics/OSVR-RenderManager/blob/75318eabd698bf1c42f64fd5ded77587215e1eb0/osvr/RenderKit/DistortionCorrectTextureCoordinate.h

[RMGLD3D16] https://github.com/sensics/OSVR-RenderManager/blob/75318eabd698bf1c42f64fd5ded77587215e1eb0/osvr/RenderKit/GraphicsLibraryD3D11.h

[RMGLGL16] https://github.com/sensics/OSVR-RenderManager/blob/75318eabd698bf1c42f64fd5ded77587215e1eb0/osvr/RenderKit/GraphicsLibraryOpenGL.h

[Eigen17] http://eigen.tuxfamily.org

[RMATW17] https://github.com/sensics/OSVR-RenderManager/blob/3458fb7ac8948215026ac416a3aa6cec4320e6af/osvr/RenderKit/RenderManagerBase.cpp#L1630

[nVidia10] http://developer.download.nvidia.com/opengl/specs/WGL_NV_DX_interop.txt

[Khronos17] https://www.khronos.org/egl/

[NTP17] http://www.ntp.org/

[VRPN01] Taylor II, R. M., et al. (2001). VRPN: A Device-Independent, Network-Transparent VR Peripheral System. Proceedings of the ACM Symposium on Virtual Reality Software & Technology, VRST, Banff Centre, Canada.

[Taylor93] R.M. Taylor II, W. Robinett, V.L. Chi, F.P. Brooks, Jr., W.V. Wright, R.S. Williams, and E.J. Snyder, "The Nanomanipulator: A Virtual-Reality Interface for a Scanning Tunneling Microscope," *Computer Graphics: Proceedings of SIGGRAPH '93*, (August 1993) pp. 127-134.

[Olano95] Marc Olano, Jon Cohen, Gary Bishop, "Combatting Rendering Latency," Proceedings of the 1995 Symposium on Interactive 3D Graphics (I3D '95), Monterey, California. pp. 19-24. 1995.

[Kooima18] "VR Developer Gems", CRC Press, Robert Kooima, Chapter 33, Ed: William R. Sherman, 2018