Cubemap Projection Onto Cylindrical Screens

By Terri Lim
Carnegie Mellon University – Entertainment Technology Center, 2025

Physical Space

The Cavern is a cylindrical *CAVE* system located at the *Carnegie Mellon University (CMU) Entertainment Technology Center (ETC)*. It features a 270° screen with a radius of 3m, and a height of 2.3m. Each of 3 ceiling projectors in the middle of the ceiling handles projecting onto a 90° section of the screen. The screen resolution is 5760×1080.

The Cavern at CMU ETC.

Accessories

The Cavern features *HTC VIVE Trackers*, and *ORBBEC Femto Bolts* which can be used for positional, rotational and body pose tracking. The Cavern also has active 3D glasses and specialised software for stereoscopic rendering.

HTC Vive Tracker

ORBBEC Femto Bolt

Active 3D Glasses

Assumptions

This document assumes that the reader is familiar with the rendering pipeline, including 2D and cubemap texture sampling. The shader code snippet in this document will be using *Unity 6 HLSL* syntax, as that was what we used for the actual implementation.

Problem Statement

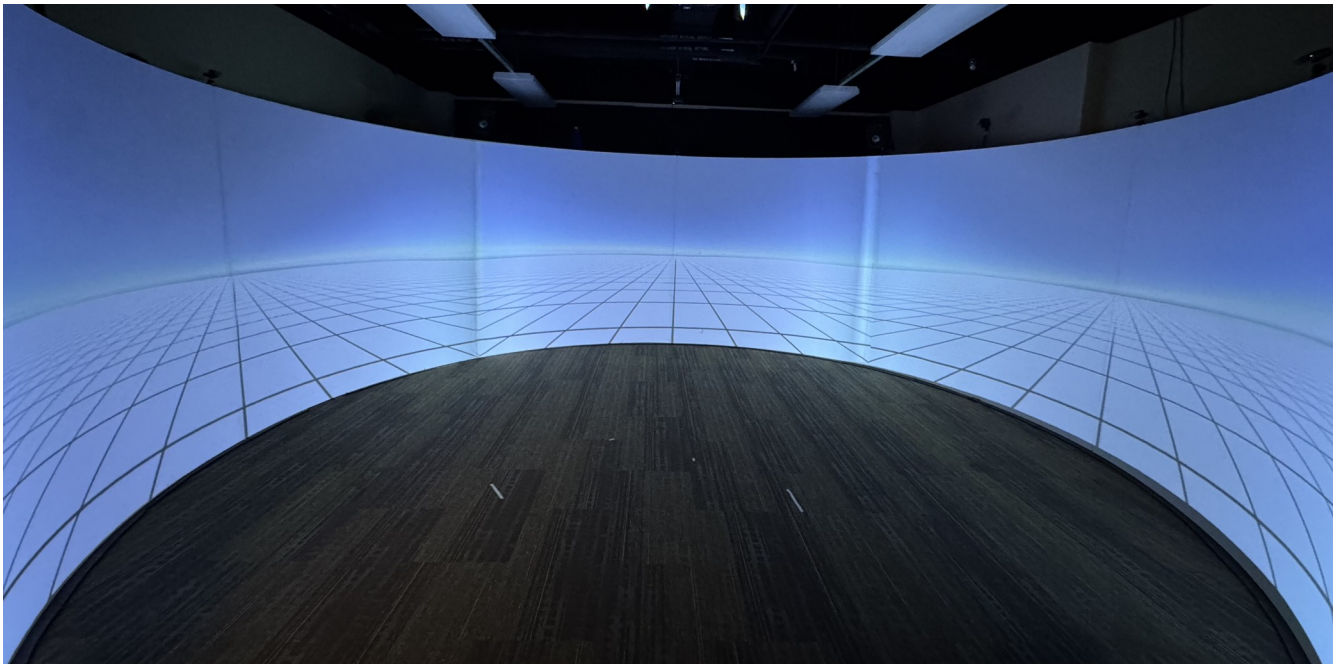
When rendering a 3D game scene onto a flat screen, the camera's view frustum is projected onto a flat image via the Model-View-Projection matrix. Cameras in games typically have a maximum field of view of 180°, and thus a single camera is insufficient for the 270° screen of the Cavern.

Furthermore, as the Cavern is a physical space which players can interact with, the dimensions of the screen should also be accounted for, so that objects in the game are rendered to scale with the physical world. That is to say, an object with a width of 1 unit in game should appear as having 1m in physical space.

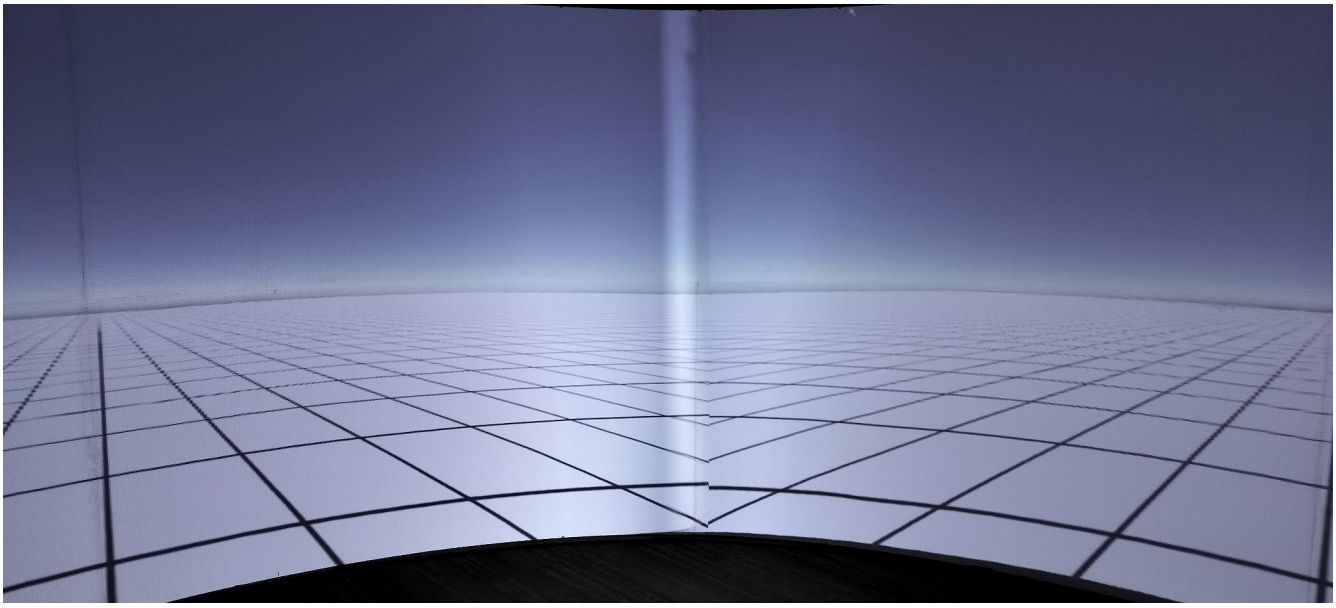
Intuitively, one might perhaps consider using three cameras, each with a 90° field of view. However, since the Cavern's screen is cylindrical, the camera has to additionally account for the curvature of the screen.

This document explains the method used to render the game scene onto the Cavern's cylindrical screen.

Not Accounting For Curvature



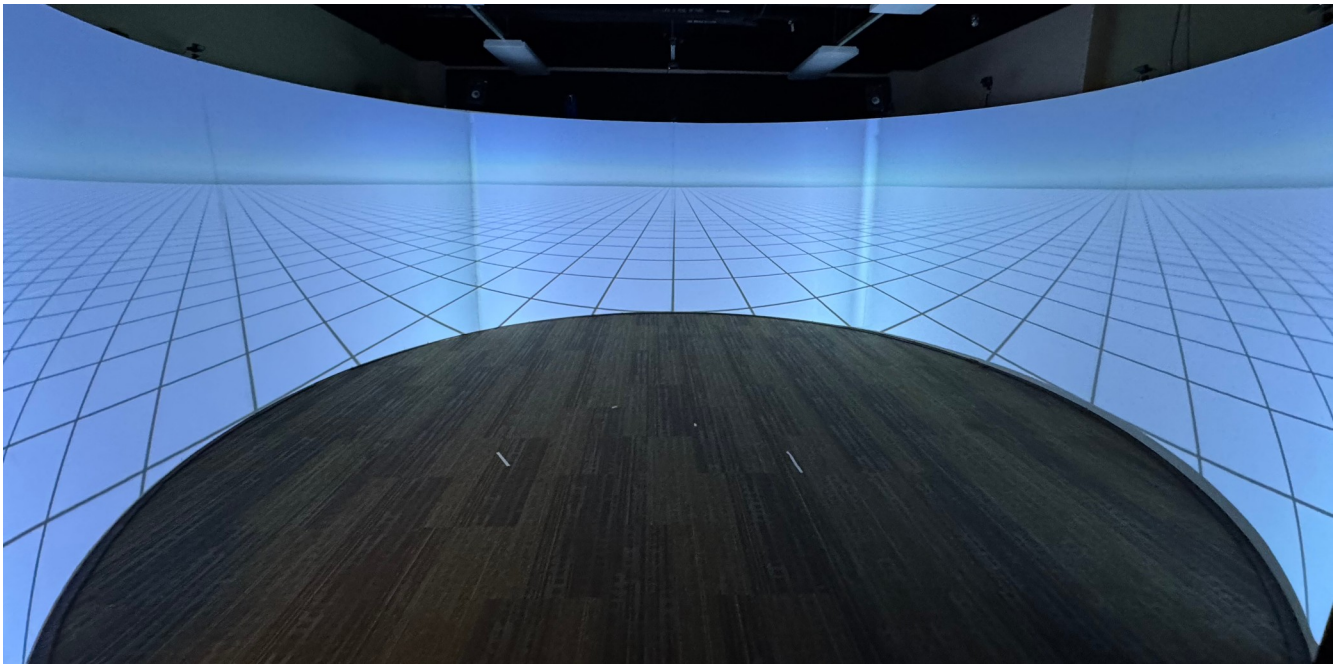
Lines appear bent on a curve screen.



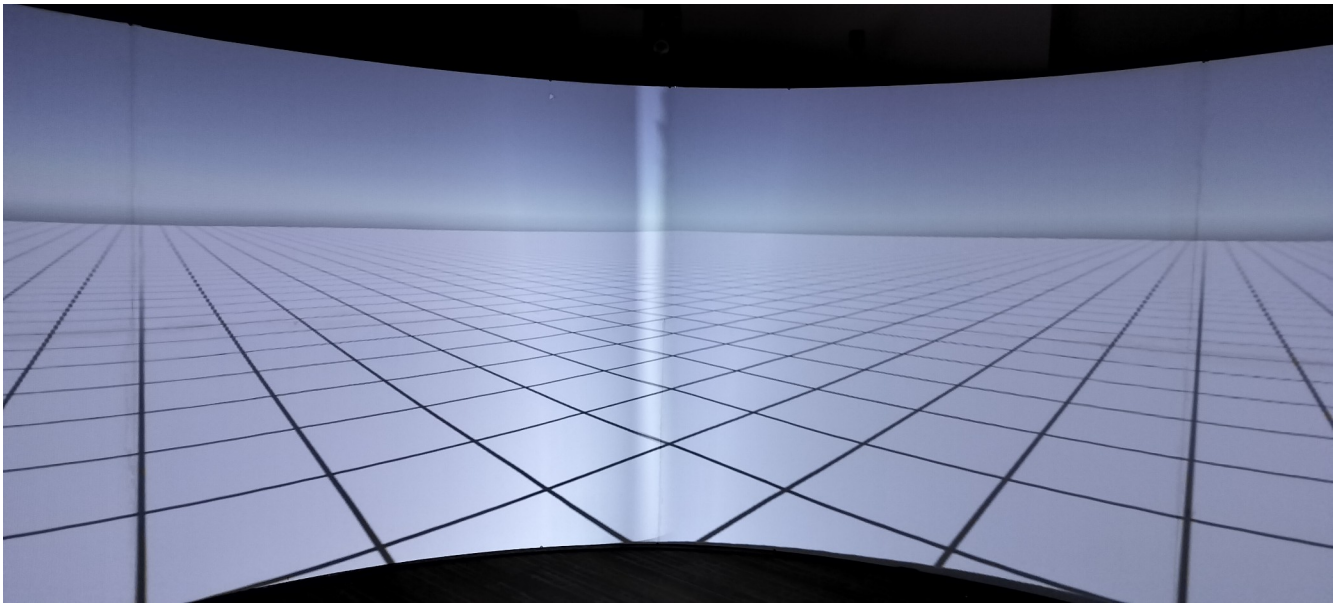
Up close view from centre of the screen.

Note that at the bottom of the screen, every tile in the grid is “bending” along with the screen, rather than remaining straight.

Accounting For Curvature



Lines appear straight.



Up close view from centre of the screen.

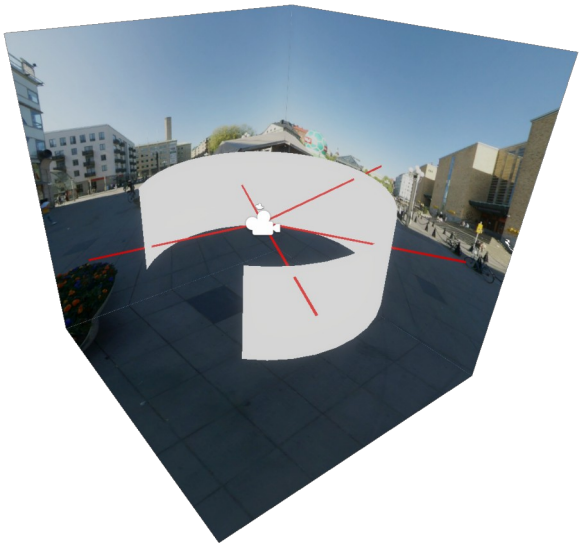
Note that at the bottom of the screen, rather then “bending” the tiles, the screen “carves out” a circle on the tiles.

Approach Overview

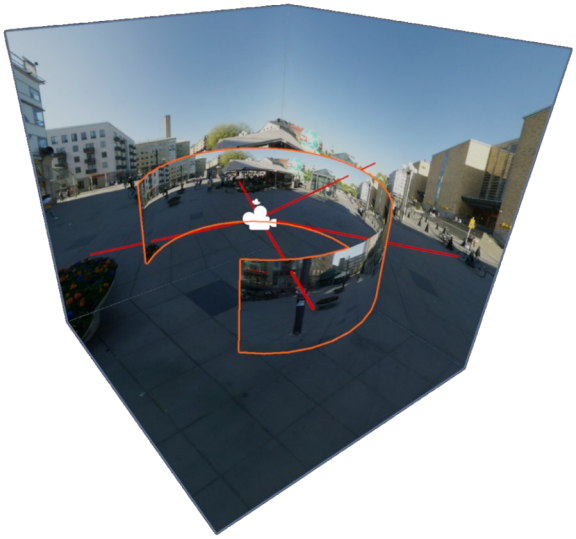
The general approach is to render the scene onto a cubemap from position of the camera. Then, use a custom shader to sample the cubemap into the final 2D render buffer that will be projected onto the screen.

We sample the cubemap by finding the direction from the camera to each fragment in phyiscal space, and using the direction as our cubemap coordinates.

Note that we should only render the faces of the cubemap that can be seen by the final 2D render buffer for performance reasons.



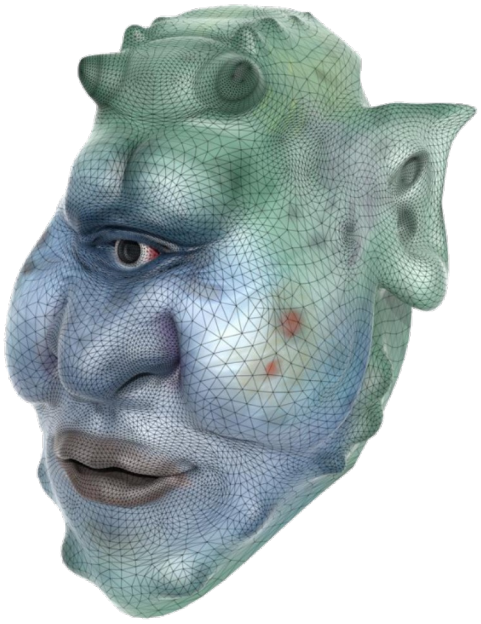
Step 1: Render a cubemap from the camera's position.



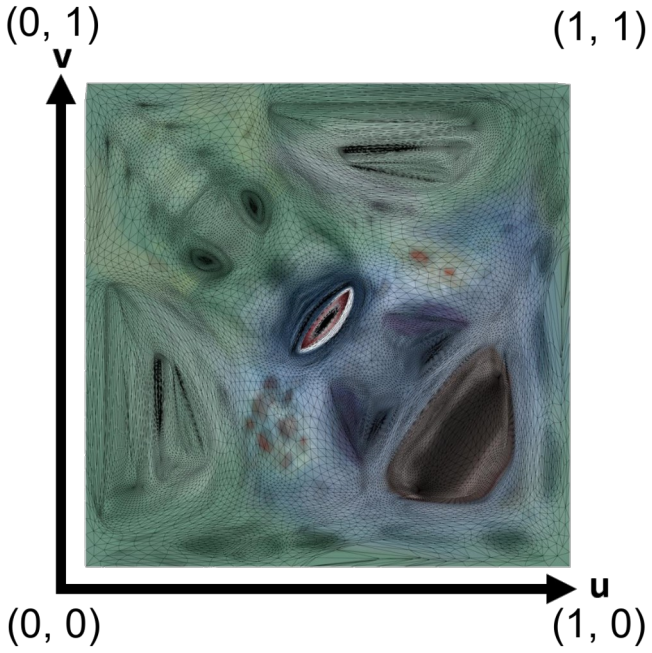
Step 2: Sample the cubemap by finding the direction from the camera to each fragment.

2D Texture Recap

UV mapping is the process of projecting a 3D model's surface to a 2D image for texture mapping. Colour is sampled from the texture using 2D texture coordinates known as UV coordinates. Texture coordinates are typically in the $[0,1]$ range for both axes.



3D Model



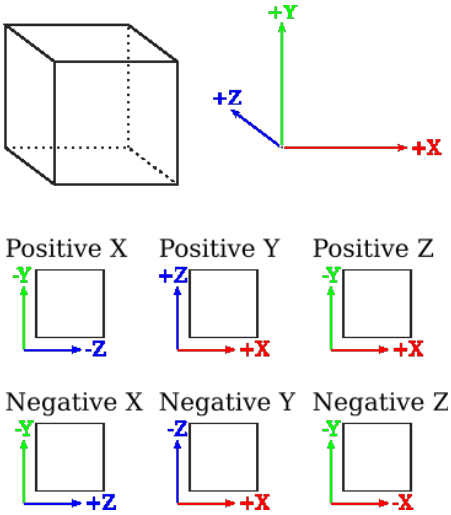
Texture Data

Cubemap Recap

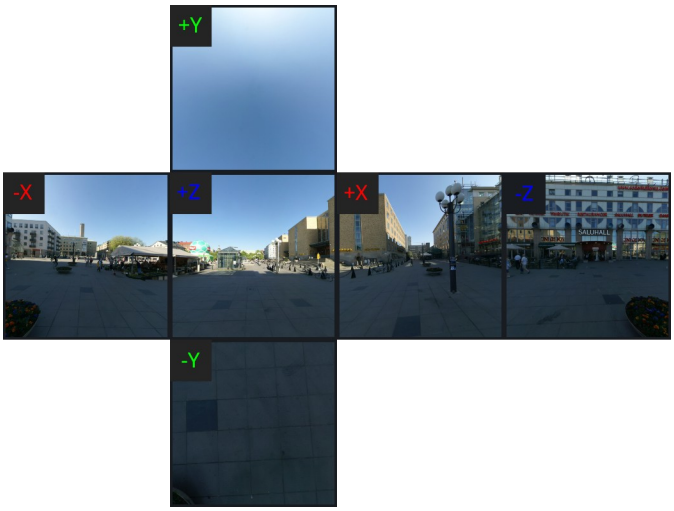
Cubemaps are 3D textures in the shape of a cube, and unlike 2D textures, colour is sampled from it using 3D coordinates, typically in the $[-1,1]$ range for all three axes. Cubemap coordinates are directional vectors, with $(0,0,0)$ being the centre of the cube. And although each axis has the range of $[-1,1]$, cubemap coordinates do not have to be normalised.

Note that by convention, cubemap coordinates are usually left-handed, even for right-handed graphics APIs such as OpenGL.

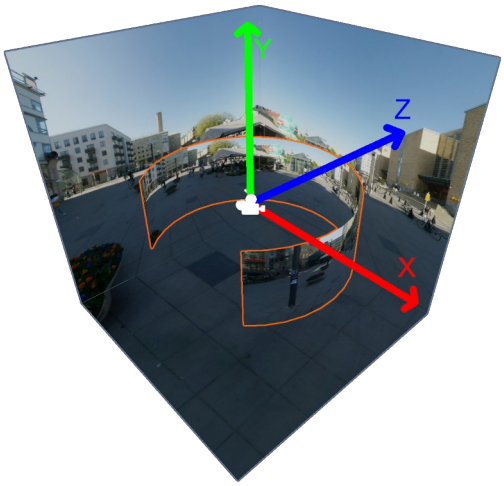
In this document, we will follow the left-handed orientation used by Unity, where $+Z$ points forward, $+X$ points right, and $+Y$ points upwards.



Cubemap Orientation

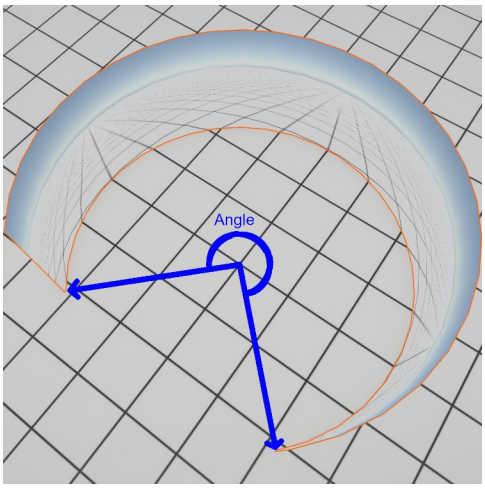


Cubemap Layout

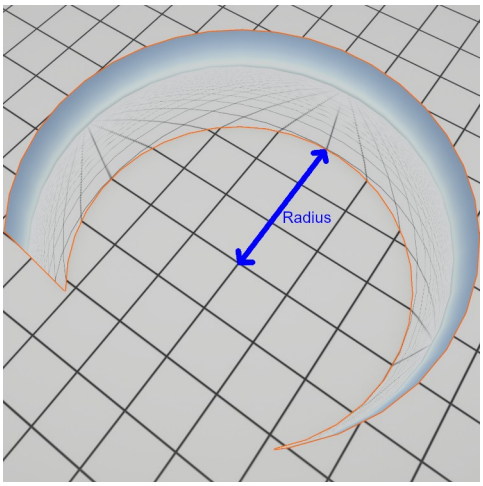


Cavern Screen Orientation

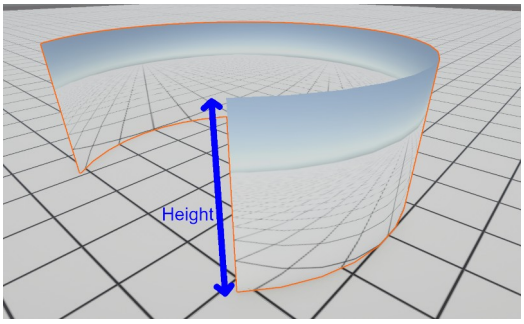
Determining Fragment Position In Physical Space



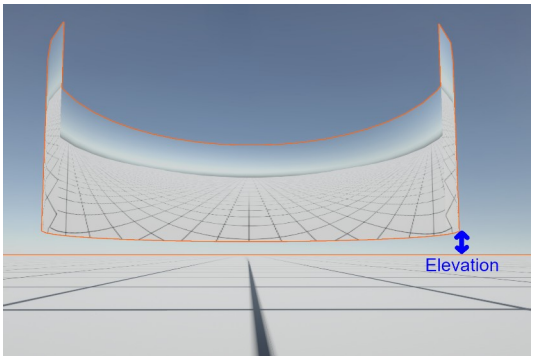
Screen Angle



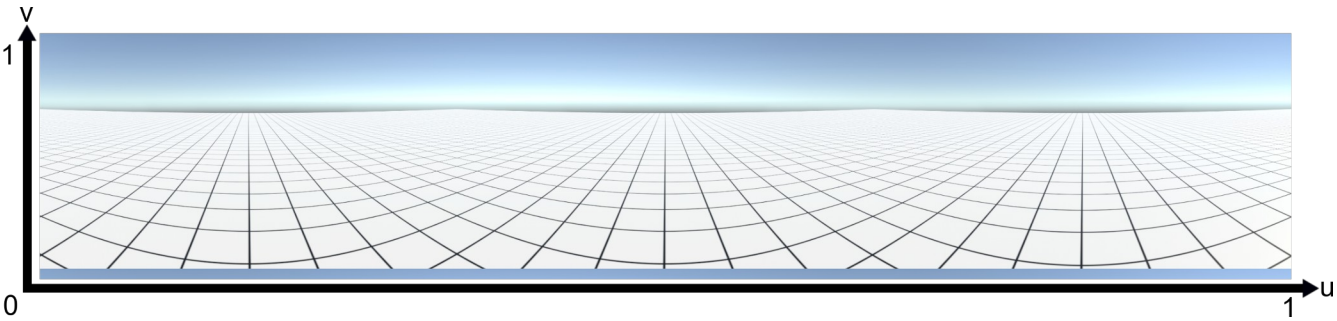
Screen Radius



Screen Height



Screen Elevation



Screen "unrolled" into 2D texture.

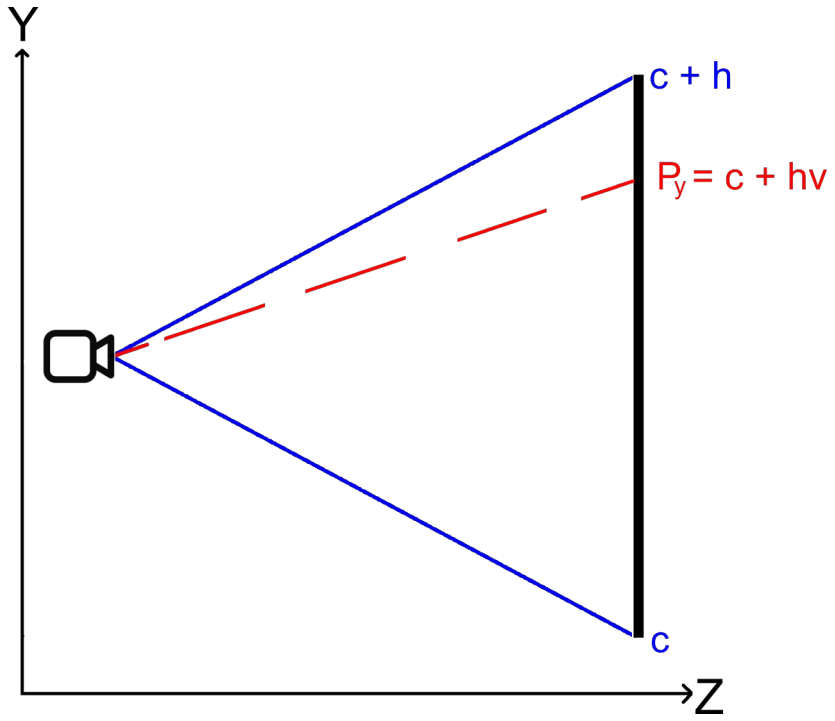
To determine the 3D position of a fragment, “unroll” our 3D screen into a flat 2D texture. Then, using the UV coordinates of each fragment on the 2D texture, find the position of the fragment in physical space when it is “rolled” up again.

Define known variables:

- h is the screen height in metres.
- r is the screen radius in metres.
- θ is screen angle radians.
- c is screen elevation off the floor in metres.
- $(u,\ v)$ is the 2D texture coordinates of the fragment on the 2D render texture.

Define unknown variables:

- P is the position of the fragment on the screen in physical space.
- ϕ is the angle between P and the $+Z$ axis.

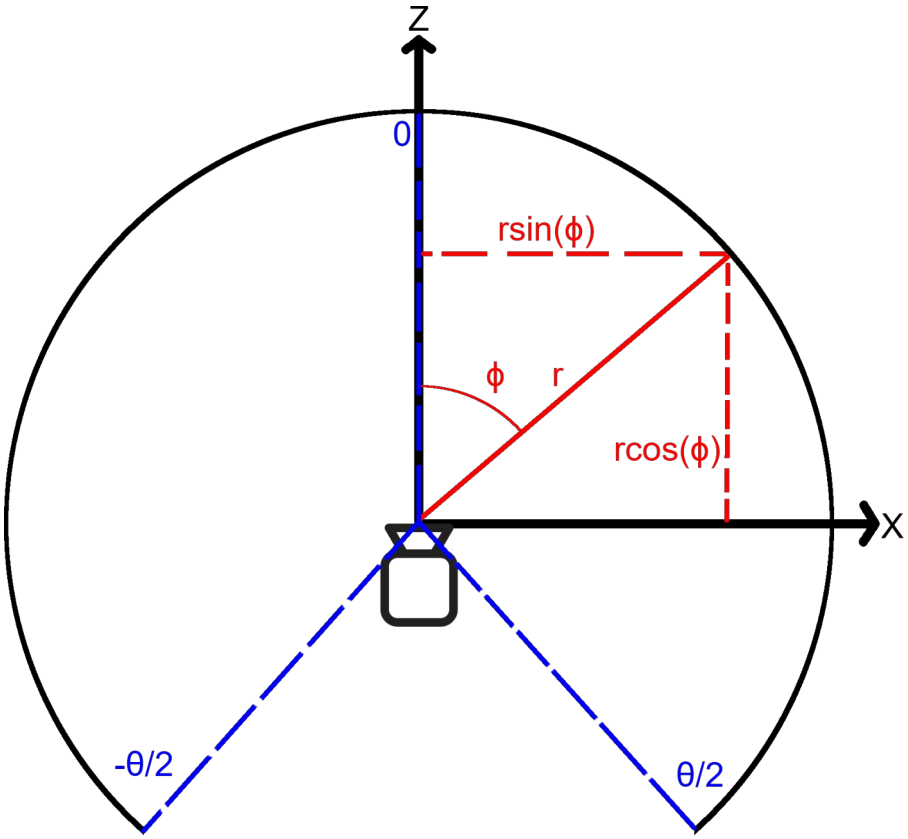


Side View of Screen

When $v=0$ then $P_y=c$, and when $v=1$ then $P_y=c+h$.

Therefore,

$$P_y=c+hv$$



Top View of Screen

When $u=0$ then $\phi=-\frac{\theta}{2}$ and when $u=1$ then $\phi=\frac{\theta}{2}$.

Therefore,

$$\phi=(2u-1)(\frac{\theta}{2})$$

and consequently,

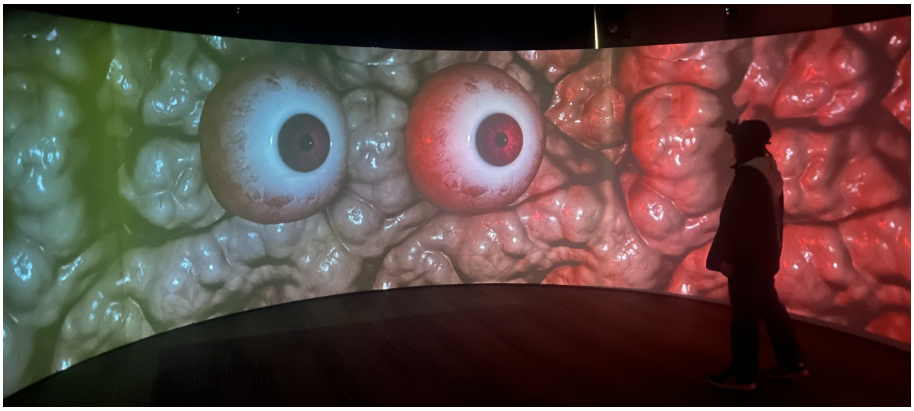
$$P_x=rsin(\phi),\ P_z=r\cos(\phi)$$

Thus we have our cubemap coordinates,

$$P=(r\sin\phi, c+hv, r\cos\phi)$$

Off-Axis Rendering

Being a physical space, players are encouraged to move about in the Cavern. The player's position can also be tracked using accessories such as the VIVE Tracker. An example would be a game where a creepy set of eyes follow a player as they move about.



Spooky eyes following the player.

However, since the camera is currently assumed to be in the centre of the screen when calculating P, this creates the issue where the perspective of the rendered image is incorrect when the player moves away from the centre. In the case of the above game example, the eyes might look like they are looking at the player when they are standing in the centre of the space, but appear to be looking past the player when they walk around.

Define player head position as H.

Then, the new direction from the camera to a fragment is

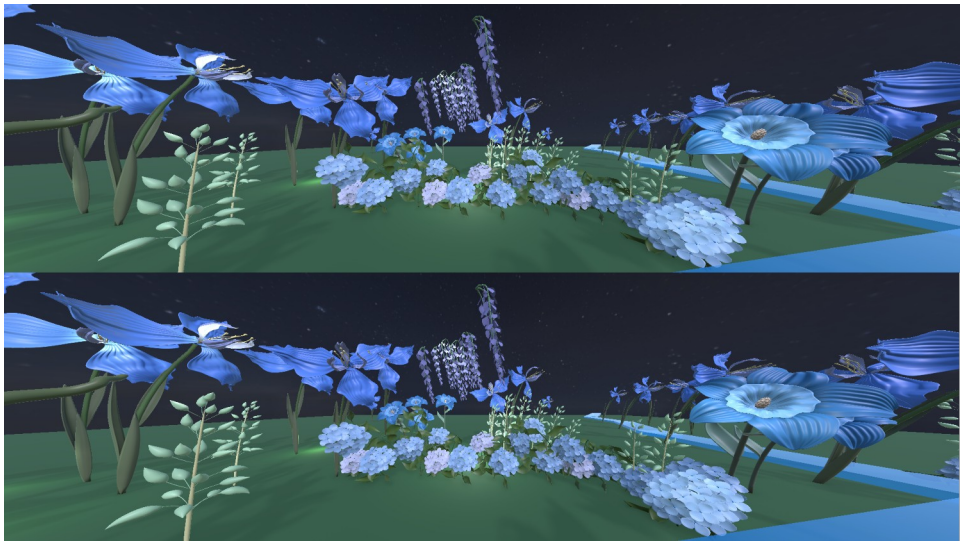
$$P' = P - H.$$

Stereoscopic Rendering

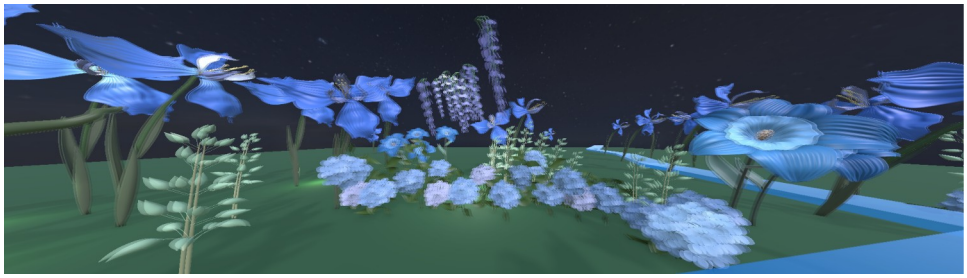
Stereoscopic rendering allows us to achieve the “popping out” effect that we see when watching a 3D movie. To render a stereoscopic view on a flat screen, we render the scene twice, each with a slight offset for each eye, and overlay the images on top of one each other. When the player puts on 3D glasses, it filters out the images such that each eye only sees one image, and the brain combines them to perceive depth.

The offset is known as the interpupillary distance (IPD), which is the distance between our eyes. On average, adult humans have an interpupillary distance of 63mm.

In the case of the Cavern, the output for each eye is vertically stacked in the output render buffer, and specialised software is used to overlay them when projecting onto the screen.

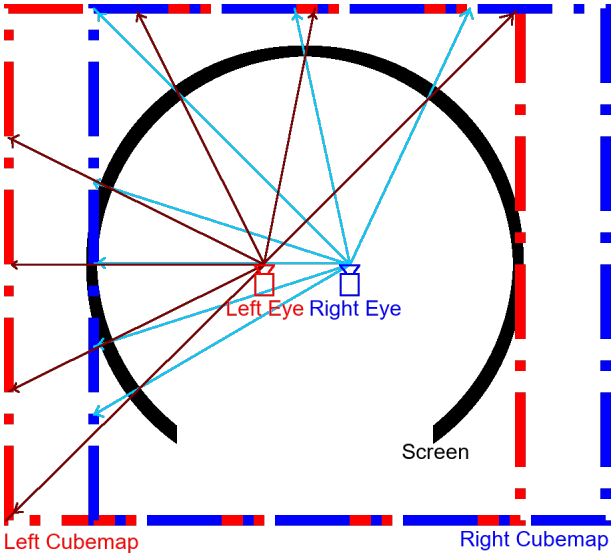


Left eye view is stacked on top of the right eye view.



Overlaying both images on top of one another.

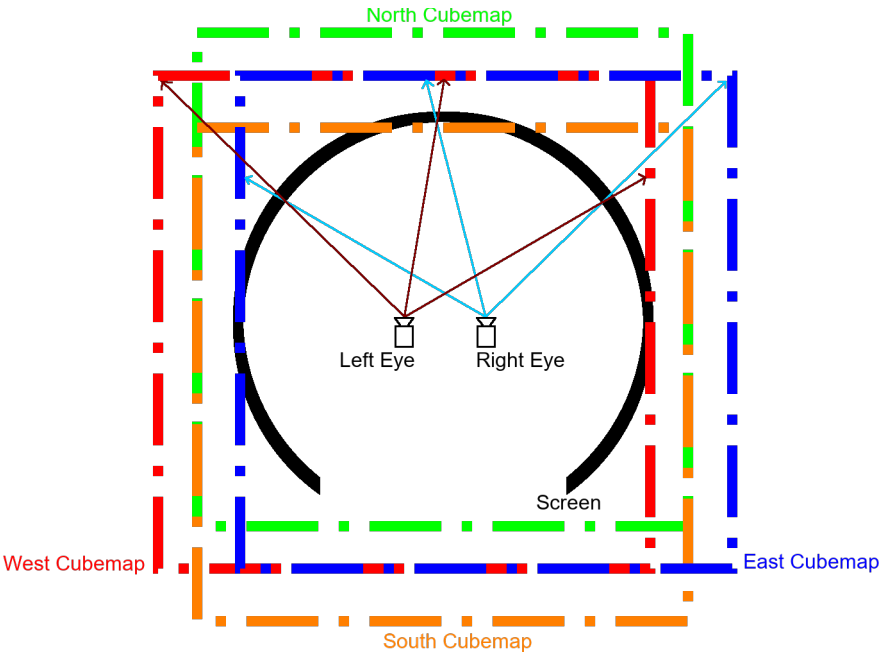
We can also approximate the effect using multiple cubemaps.



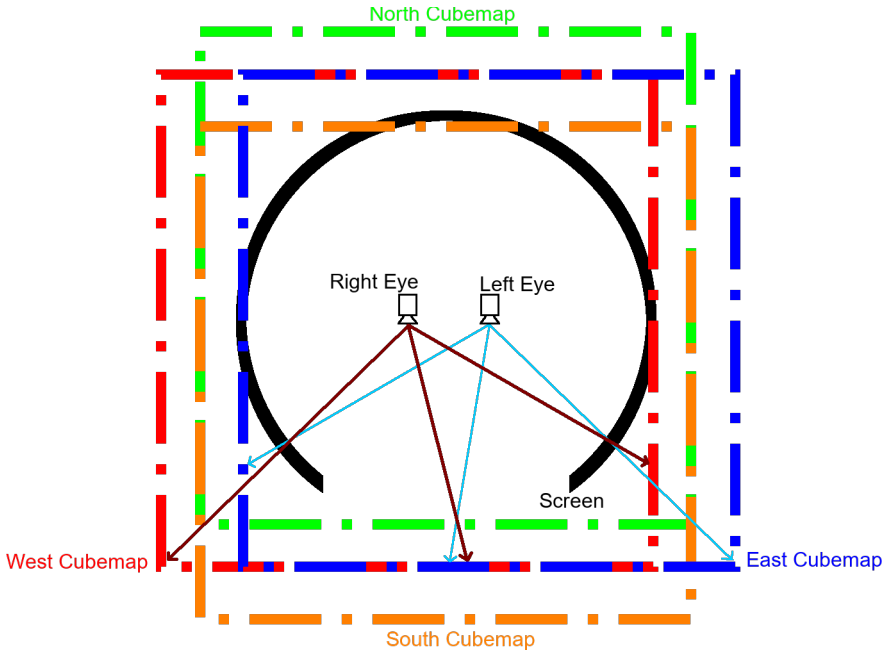
Top view of Cavern rendering with two cubemaps.

Notice that as the screen angle approaches 90°, the stereoscopic effect decreases as the IPD between the eyes approaches zero. Furthermore, going beyond 90°, the view from the left eye is now to the right of the right eye, resulting in the depth perception of objects being reversed.

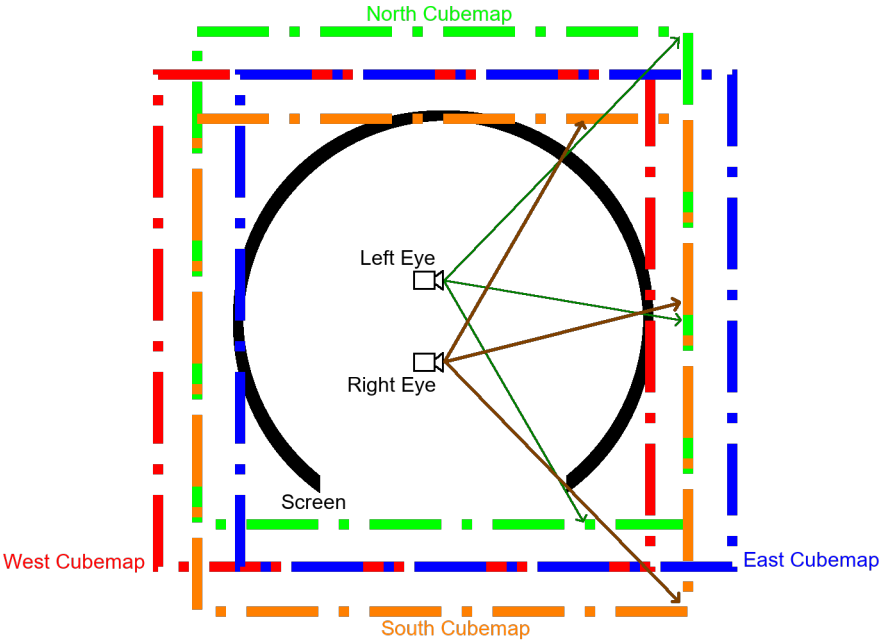
To solve these issues, we will use four cubemaps, each with an offset in each cardinal direction. Then, split the screen into four 90° quadrants, one for each cardinal direction. Finally for each quadrant, the left and right eye will select a different cubemap to sample from.



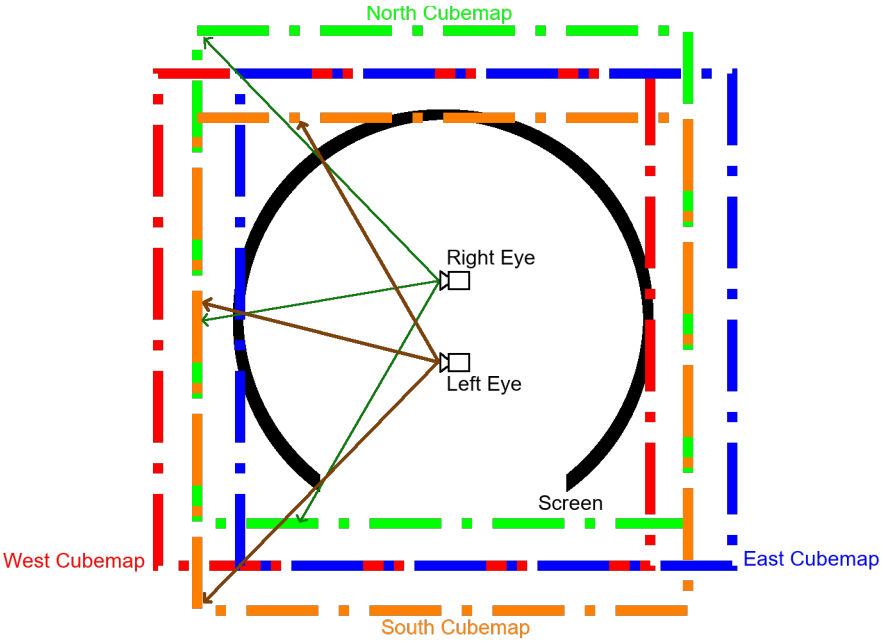
When facing northwards, the left and right eyes sample from the west and east cubemaps respectively.



When facing southwards, the left and right eyes sample from the east and west cubemaps respectively.

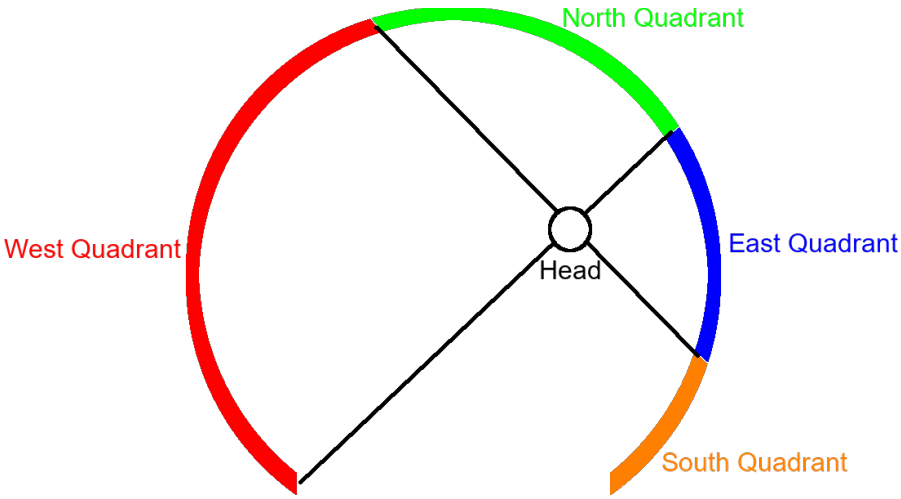


When facing eastwards, the left and right eyes sample from the north and south cubemaps respectively.



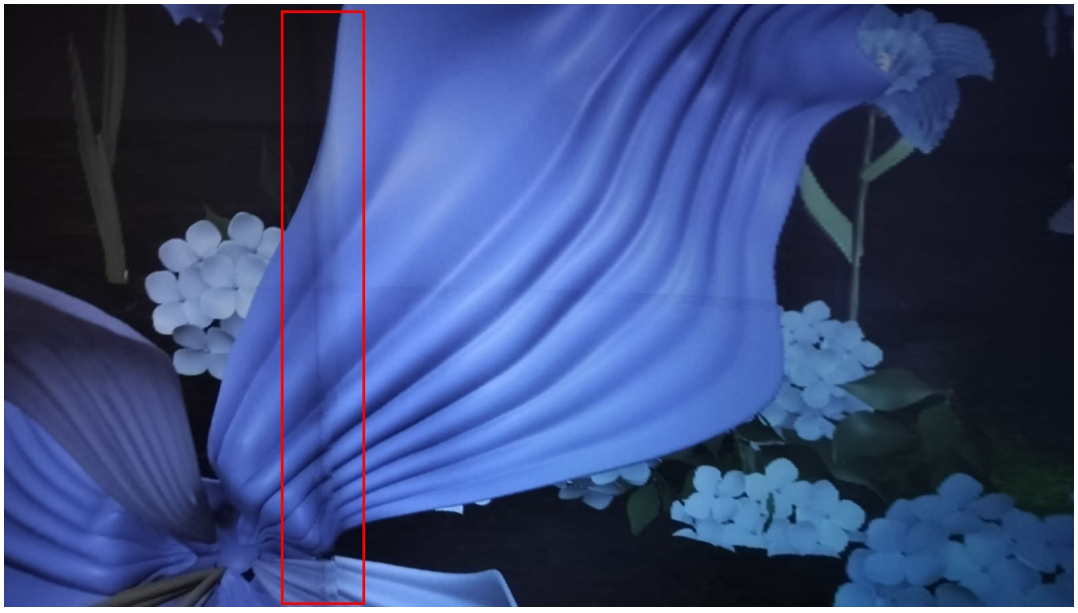
When facing eastwards, the left and right eyes sample from the south and north cubemaps respectively.

As off-axis rendering is supported, the quadrants should always be divided relative to the player's position instead of at the centre of the screen. That is easily done by simply finding the angle between $(P_x, 0, P_z)$ and the forward vector $(0, 0, 1)$ via their dot product.

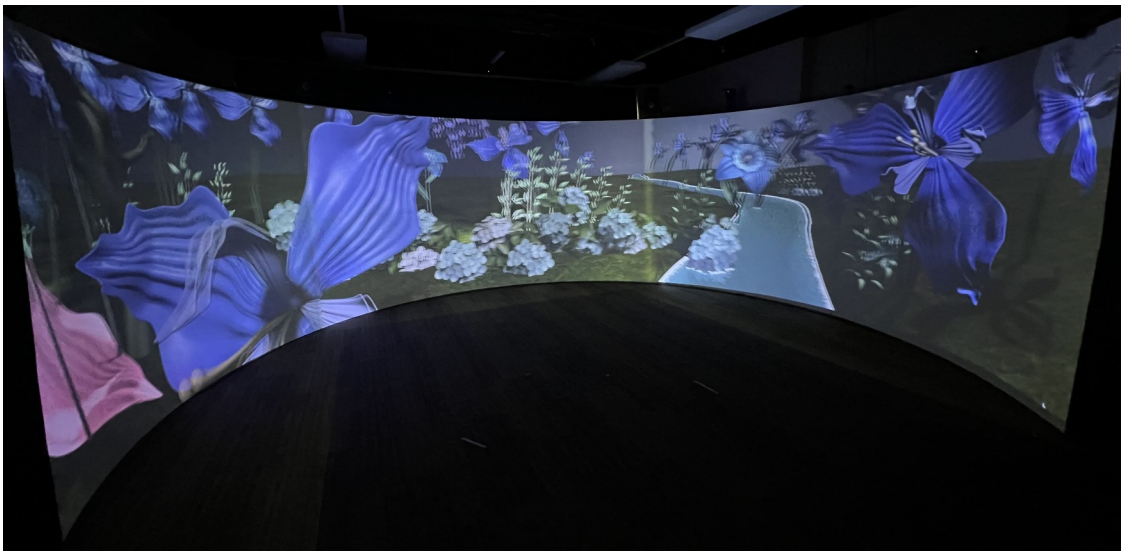


Divide the screen into quadrants based on the head's position.

A flaw of this method is that it tend to create a vertical seam along the lines where the quadrants. Despite that, during testing a majority of our users who spent about 15 minutes in the Cavern, the average time of an experience, did not notice it unless it was specifically pointed out to them.



Vertical seam where quadrants connect.



Stereoscopic Rendering Photo

Shader Code Snippet

```
// Cavern Dimensions Uniforms
float _CavernHeight;
float _CavernRadius;
float _CavernAngle;
float _CavernElevation;

// Off-Axis Rendering Uniforms
float3 _HeadPosition;

// Stereoscopic Rendering Uniforms
int _EnableStereoscopic;
float _InterpupillaryDistance;

// The vertex function, runs once per vertex.
Vert2Frag Vertex(Attributes input) {
    VertexPositionInputs positionInputs = GetVertexPositionInputs(input.positionOS);
    Vert2Frag output;
    output.positionCS = positionInputs.positionCS; // Set the clip space position.
    output.uv = input.uv; // Set the texture UV coordinates.
    return output;
}

// Helper function to sample colour for left eye.
float4 SampleLeftEye(float3 headToScreen, float fragmentRelativeAngle, float3 ipdOffsetZ, float3 ipdOffsetX) {
    // Physcial screen rear quadrant relative to head position.
    if (fragmentRelativeAngle > 135.0f || fragmentRelativeAngle < -135.0f) {
        return SAMPLE_TEXTURECUBE(_CubemapEast, sampler_CubemapEast, headToScreen - ipdOffsetX);
    }
    // Physcial screen left quadrant relative to head position.
    if (fragmentRelativeAngle < -45.0f) {
        return SAMPLE_TEXTURECUBE(_CubemapSouth, sampler_CubemapSouth, headToScreen + ipdOffsetZ);
```

```
    }
    // Physcial screen right quadrant relative to head position.
    if (fragmentRelativeAngle > 45.0f)  {
        return SAMPLE_TEXTURECUBE(_CubemapNorth, sampler_CubemapNorth, headToScreen - ipdOffsetZ);
    }
    // Physcial screen front quadrant relative to head position.
    return SAMPLE_TEXTURECUBE(_CubemapWest, sampler_CubemapWest, headToScreen + ipdOffsetX);
}

// Helper function to sample colour for right eye.
float4 SampleRightEye(float3 headToScreen, float fragmentRelativeAngle, float3 ipdOffsetZ, float3 ipdOffsetX) {
    // Physcial screen rear quadrant relative to head position.
    if (fragmentRelativeAngle > 135.0f || fragmentRelativeAngle < -135.0f) {
        return SAMPLE_TEXTURECUBE(_CubemapWest, sampler_CubemapWest, headToScreen + ipdOffsetX);
    }
    // Physcial screen left quadrant relative to head position.
    if (fragmentRelativeAngle < -45.0f) {
        return SAMPLE_TEXTURECUBE(_CubemapNorth, sampler_CubemapNorth, headToScreen - ipdOffsetZ);
    }
    // Physcial screen right quadrant relative to head position.
    if (fragmentRelativeAngle > 45.0f) {
        return SAMPLE_TEXTURECUBE(_CubemapSouth, sampler_CubemapSouth, headToScreen + ipdOffsetZ);
    }
    // Physcial screen front quadrant relative to head position.
    return SAMPLE_TEXTURECUBE(_CubemapEast, sampler_CubemapEast, headToScreen - ipdOffsetX);
}

// The fragment function, runs once per pixel on the screen.
float4 Fragment(Vert2Frag input) : SV_TARGET {
    // Split the screen into 2 halves, top and bottom.
    // For stereoscopic rendering, the top will render the left eye, the bottom will render the right eye.
    // For monoscopic rendering, both halves will render the same thing.
    const bool isLeftEye = 0.5f < input.uv.y;

    float2 uv = input.uv;
    // For the left eye, convert the UV's y component from the [0.5, 1] range to the [0, 1] range.
    // For the right eye, convert the UV's y component from the [0, 0.5] range to the [0, 1] range.
    uv.y = isLeftEye ? (uv.y - 0.5) * 2.0f : uv.y * 2.0f;

    // Find the angle of the fragment on screen. Take note that angle 0 points down the Z-axis, not the X-axis.
    const float fragmentAngle = (uv.x * 2.0f - 1.0f) * _CavernAngle * 0.5f;
    // Find the direction from the head to the fragment.
    const float3 headToScreen = float3(_CavernRadius * sin(radians(fragmentAngle)),
                                       _CavernElevation + _CavernHeight * uv.y,
                                       _CavernRadius * cos(radians(fragmentAngle))) - _HeadPosition;

    // Monoscopic mode.
    if (!_EnableStereoscopic) {
        return SAMPLE_TEXTURECUBE(_CubemapNorth, sampler_CubemapNorth, headToScreen);
    }

    // Stereoscopic mode.
    const float3 ipdOffsetZ = float3(0.0f, 0.0f, _InterpupillaryDistance * 0.5f);
```

```
    const float3 ipdOffsetX = float3(_InterpupillaryDistance * 0.5f, 0.0f, 0.0f);

    // Because the head might not be in the centre of the screen, we have to find the angle of the fragment relative to the head.
    const float3 forwardDir = float3(0.0f, 0.0f, 1.0f);
    const float3 headToScreenXZ = normalize(float3(headToScreen.x, 0.0f, headToScreen.z));
    const float fragmentRelativeAngle = degrees(acos(dot(forwardDir, headToScreenXZ))) * ((0.0f < headToScreenXZ.x) ? 1.0f : -1.0f);

    return isLeftEye
               ? SampleLeftEye(headToScreen, fragmentRelativeAngle, ipdOffsetZ, ipdOffsetX)
               : SampleRightEye(headToScreen, fragmentRelativeAngle, ipdOffsetZ, ipdOffsetX);
}
```

3rd Party Textures & Images Used In This Document

- CMU Computer Graphics, Nancy Pollard – http://15362.courses.cs.cmu.edu/spring2025/lecture/raster
- OpenGameArt Urban Skyboxes, Emil Persson – https://opengameart.org/content/urban-skyboxes