Rendering Omni-directional Stereo Content

Google Inc.

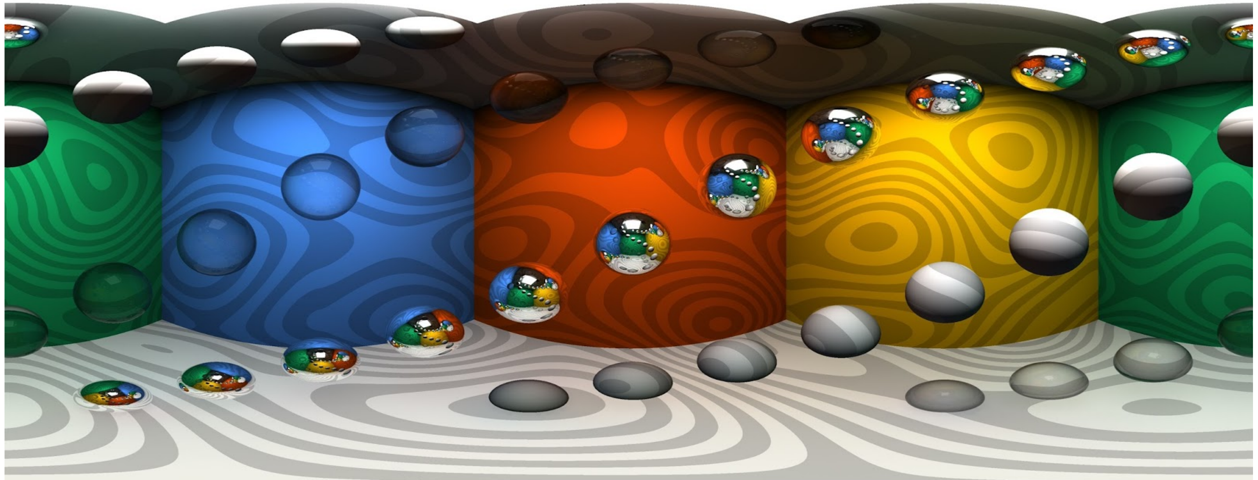
Introduction

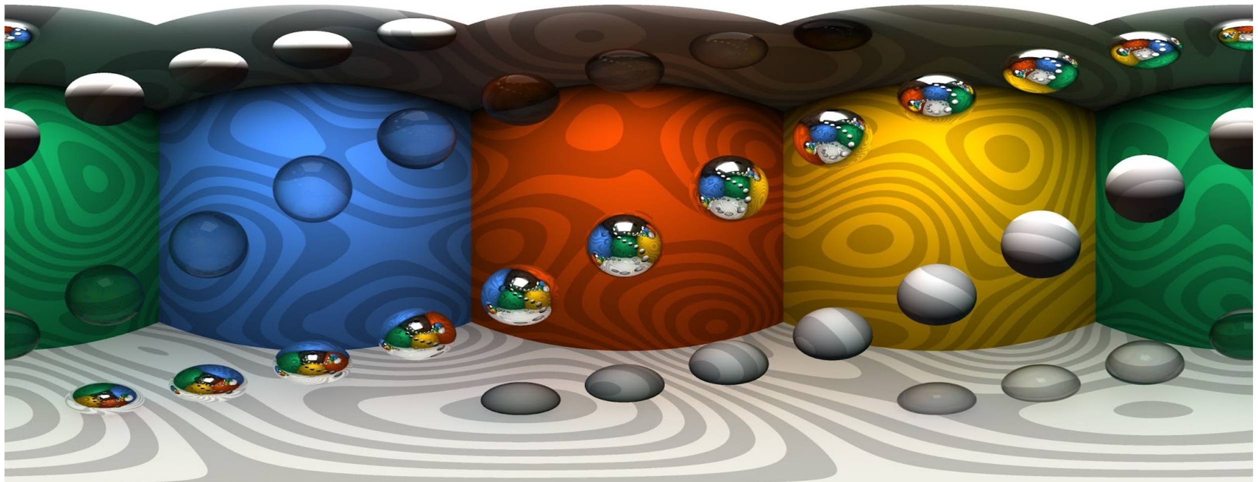
Omni-directional stereo (ODS) is a projection model for stereo 360 degree videos. It's designed for VR viewing with a head-mounted display (HMD). ODS uses a special projection format which has the following advantages:

- It is **panoramic and stereo everywhere**—there are no bad seams or dead zones (except directly above and below the camera, which we will discuss later).
- It's **pre-rendered**—like film and video, it plays well on all devices.
- It's encoded as two video streams—it can be stored, edited, and transmitted **using conventional tools**.

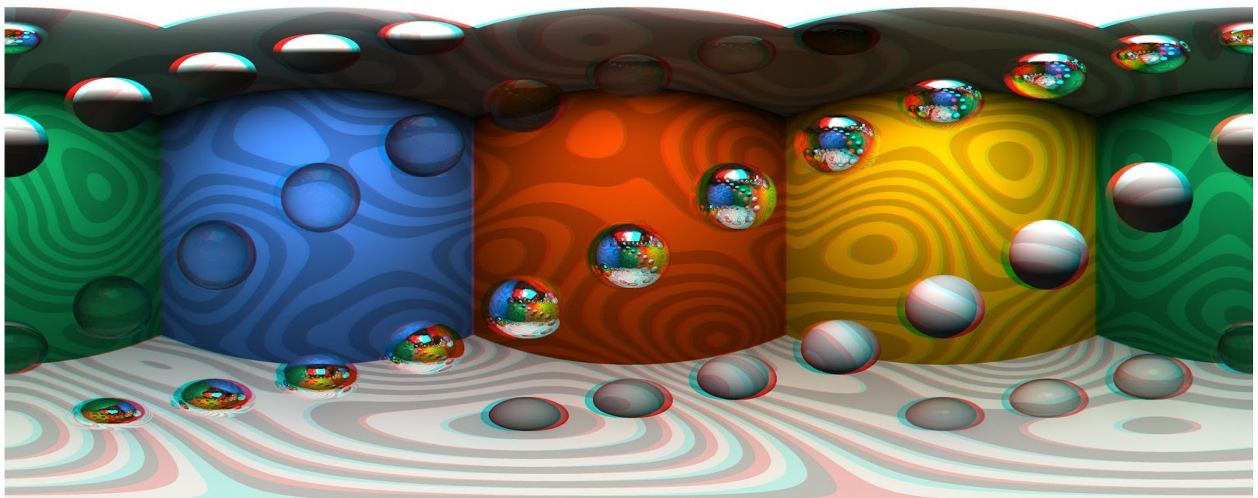
This document describes the theory behind ODS and provides instructions and pseudocode to facilitate implementing a virtual ODS camera in your favorite rendering software.

> Google recently announced Jump which is a cutting-edge camera plus video processing software solution that delivers live-action ODS content. You can use the technique described in this document to render text and title overlays or other CG effects in the ODS format, which can then be composited with Jump content. If you're interested in using Jump to capture live action content, see http://g.co/jump.





Above: A rendered ODS image. The left eye is on top, the right eye is on the bottom.

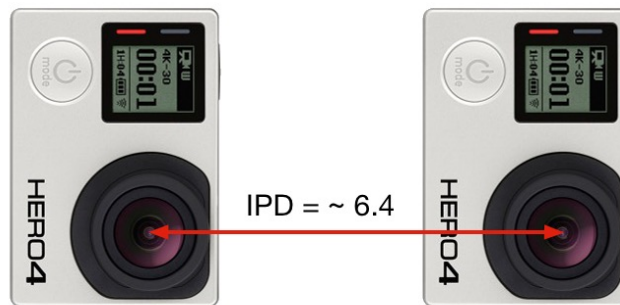


Above: An anaglyph image to make the left-right eye difference more apparent.

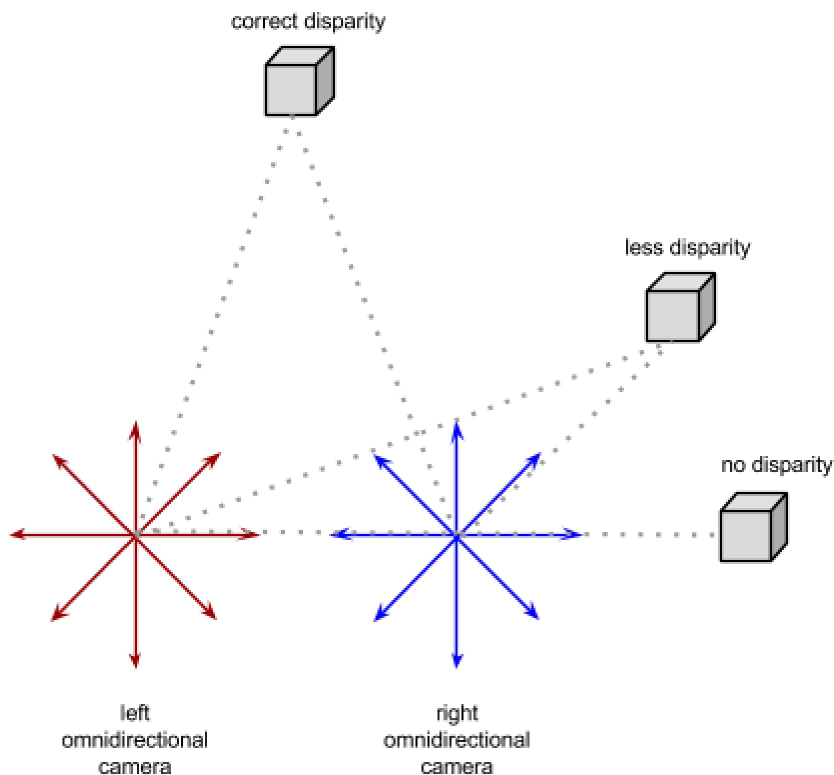
Note: The images on this page were made using RenderMan.

What’s so hard about stereo 360° video?

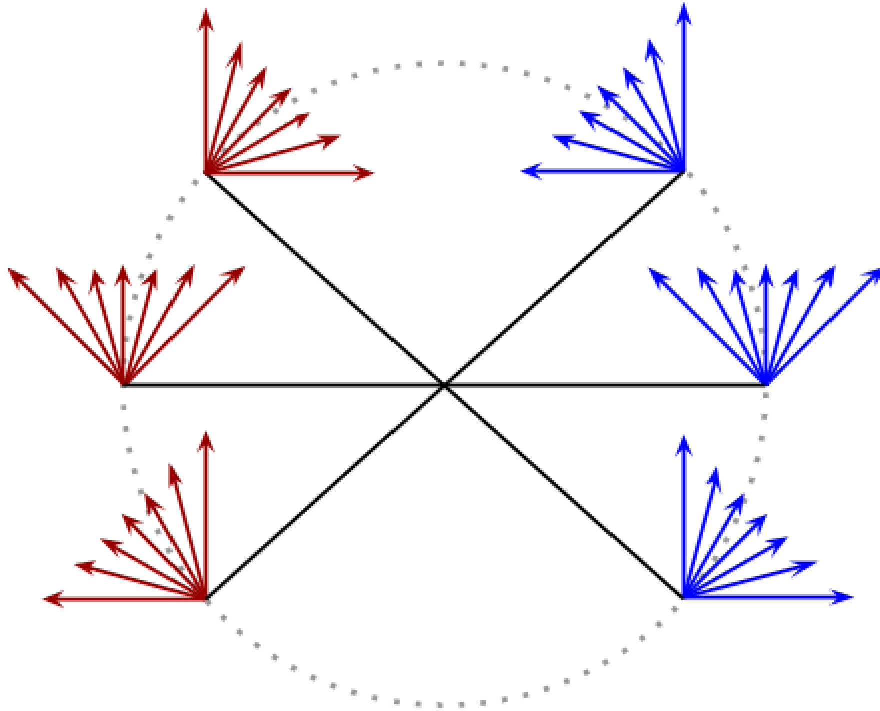
One of the ways our brains achieve stereo vision is by fusing the two images from our eyes. Nearby objects have greater disparity (shift in image position between left and right eyes) than far away objects. Correspondingly, to capture stereo video, we need to capture synchronized video from two cameras set apart at interpupillary distance (IPD), on average about 6.4 cm.



For VR, we need omnidirectional (360 degree) video. Doing this in stereo is *not* as simple as placing two omnidirectional cameras set apart at IPD. Aside from the issue that the cameras will see each other (in the real world anyway), the problem is that objects lying on the line between the two camera centers will have no disparity. The camera ray geometry would look like the following diagram.



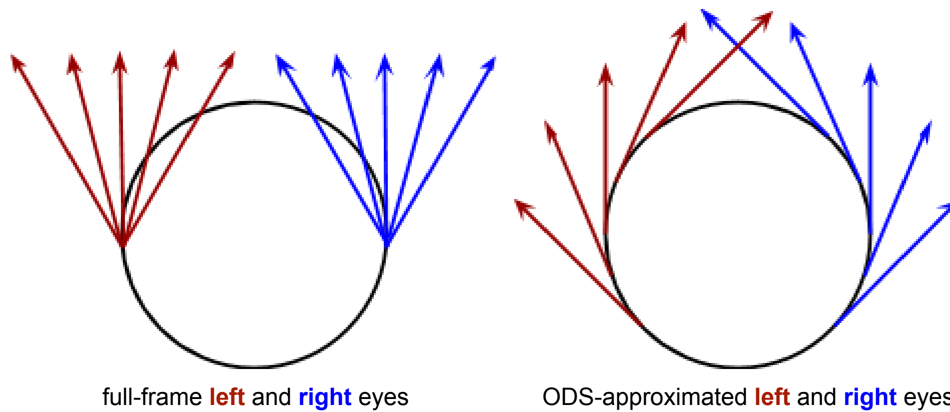
However, in reality, what happens is that your eyes *move* as you turn your head. This is how real-time CG stereo rendering works. Now imagine you could capture a stereo image for every head orientation, say one pair for every degree.



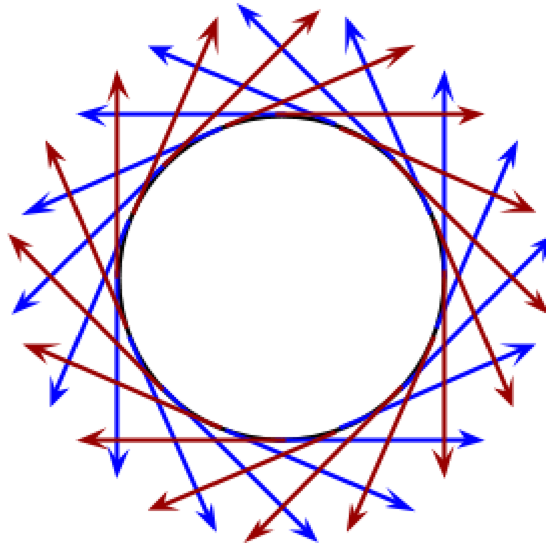
That would be a lot of images! But if you're willing to tolerate a small amount of distortion, you can actually create a pair of 2D omnidirectional images which achieve nearly the same effect.

ODS projection

Instead of capturing a full image of rays at each camera position, imagine capturing only the central ray of each camera and borrowing the other ray directions from the other cameras. The ray for your left pixel is actually the central ray captured by a camera counter-clockwise along the circle. In this case, the ray geometry for left and right eyes looks like this:



…which, when extended to be omnidirectional, looks like this:



Building a physical camera with this ray geometry is a hard problem (see g.co/jump), but fortunately, for CG content it should be as simple as changing the ray equations in your ray tracer.

You have probably noticed that the ODS images at the beginning of this document look similar to a conventional cylindrical or spherical (equirectangular) panorama. Indeed, when the IPD goes to zero or if the scene is at infinity, ODS becomes equirectangular. Consequently, it can be used just like an equirectangular image, that is, projected onto a sphere and re-rendered, and it's compatible with existing panorama viewers.

Limits and distortions

Of course, ODS does not exactly reproduce the image seen by the eye. This is because instead of all rays originating at a single point, each ray originates from a slightly different point on the capture circle. While ODS produces excellent results in general, very close-up objects can cause two problems:

- **Line bending**—straight lines in the real world may appear slightly curved in the image.
- **Vertical disparity**—after projecting the ODS image to perspective for viewing, some points may project to slightly different *y* coordinates in left and right eyes. If this effect is severe, the two images will not be fused by our visual system.

The magnitude of the distortion depends on the *distance* of objects to the camera as well as their *angle* to the horizon. The distortion is worse for nearby objects and may be quite severe directly above and below the camera. To avoid any issues with distortion, we recommend the following:

- In general, objects should remain at least 60cm from the camera (relative to an IPD of 6.5cm).
- Objects appearing directly above or below the camera should remain at least 3.5m from the camera (relative to an IPD of 6.5cm).

How to render ODS

ODS is a new medium and there are artistic and technical challenges to creating great content for it. On the technical side, you will need to implement the ODS camera. This means extending your favorite ray tracer to produce ODS images directly, or if that's not an option, rendering hundreds of perspective images from various points on the capture circle and assembling the images into ODS. On the artistic side, you want content that looks great in the ODS format, showcasing both the stereoscopic and omnidirectional aspects of the format.

Ray tracing the ODS projection

One of the appealing qualities of the ODS projection is that it can easily be represented in current video formats. The left and right images are stacked vertically. Resolution and frame rate are adjustable, but we have found that 4096x4096 at 30fps provides a good experience on today's HMDs.

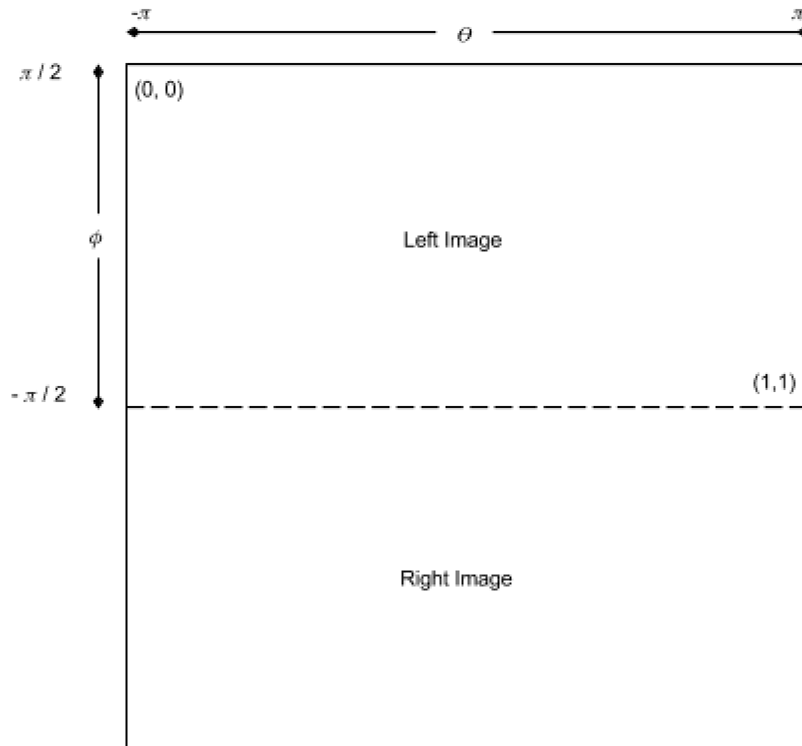
Image pixels map to viewing rays as follows: we divide the video frame into left and right images and define normalized image coordinates so that the top left corner of the top left pixel is (0,0) and the bottom right corner of the bottom right pixel is (1,1). Therefore, for a 4096x4096 video frame:

Left image: `[0,4096] x [0,2048] -> [0,1] x [0,1]`
Right image: `[0,4096] x [2048,4096] -> [0,1] x [0,1]`

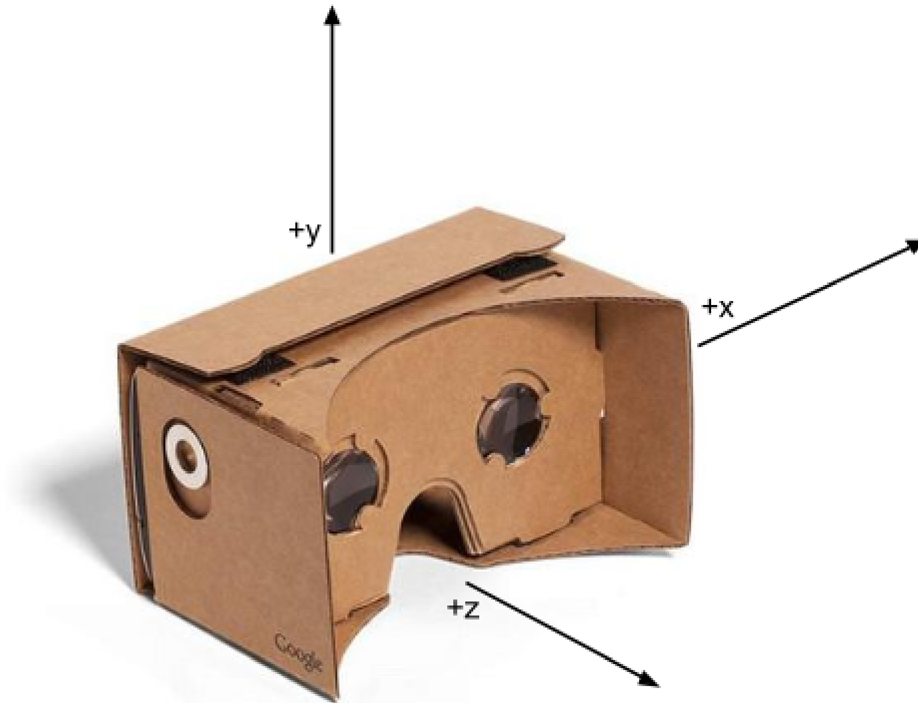
Some care should be taken during sampling and filtering operations to avoid bleeding across the seam between the left and right images. We recommend that you compose left and right images separately and stack them only as the last step.

Next we map from normalized image coordinates to equirectangular coordinates (θ, ϕ). (The ODS projection is very similar to the equirectangular projection, except of course that the rays emanate from a circle of IPD diameter instead of a single point.) We define θ (`theta`) and ϕ (`phi`) so that (0, 0) corresponds to the center of the image, which is (0.5, 0.5) in normalized image coordinates. This is the default viewing direction for the HMD, meaning the viewer is looking forward, not having turned their head.

```
theta = x * 2 * pi - pi
phi = pi / 2 - y * pi
```



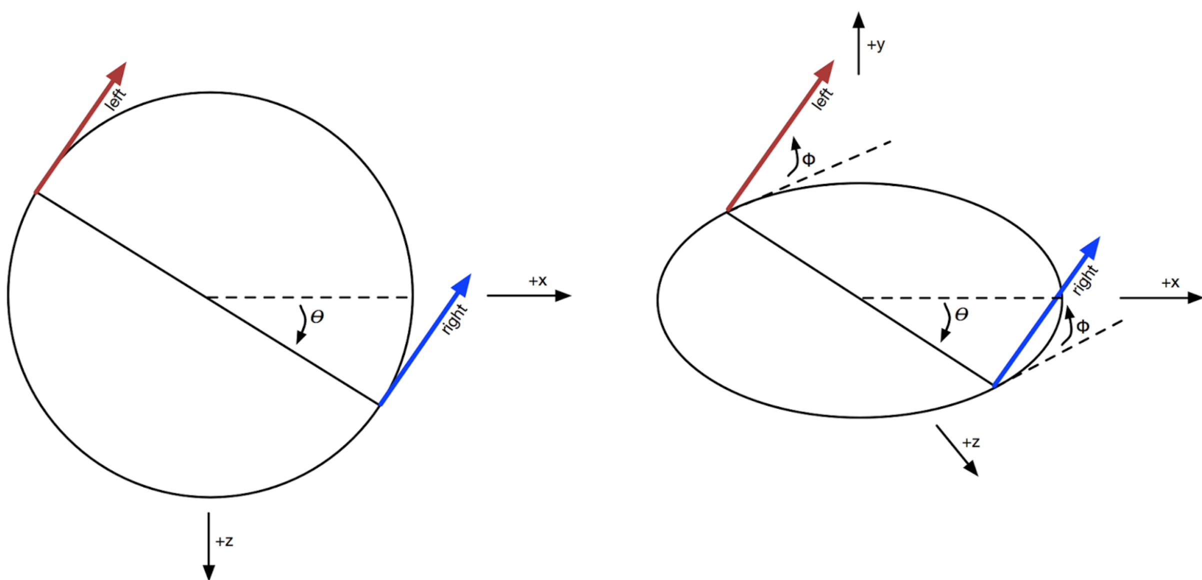
Now, we define camera coordinates. Our camera coordinate system is a right-handed 3D Cartesian system where positive *x* points to the right, positive *y* points up, and positive *z* points backward.



In this system, ODS viewing rays are defined as:

```
ray_origin = [cos(theta), 0, sin(theta)] * IPD / 2 * (is_left ? -1 : 1)
ray_direction = [sin(theta) * cos(phi), sin(phi), -cos(theta) * cos(phi)]
```

where "`[x, y, z]`" is a vector, "`IPD`" is the interpupillary distance (which you may need to scale to match your world coordinates), and "`( ? : )`" is the C++ ternary operator.



For ray tracing pseudocode, see the ray tracing example near the end of this document.

ODS by stitching perspective images

If implementing a new camera isn't an option you can use perspective cameras in a *rotating slit* approach to achieve the same result, albeit with more effort. Do this by moving the camera in small increments along the capture circle while pointing the camera tangential to the circle (-90 degrees for the right eye and +90 degrees for the left eye). At each position, render a *vertical slit* image (e.g., 1 pixel wide), and then *stitch* those images together to form the ODS image. Each of these renders must exactly reproduce the scene for the same point in time in the animation. Any random animation effects must be made deterministic.

Stitching may be as simple as stacking the images horizontally. However, to reduce any seam artifacts, you might render each slit with additional padding and blend together the neighboring overlapping slits. Note that some post-render image effects (such as bloom, depth of field) may require slits to be large enough to accommodate that effect's *kernel* size.

For highest quality, use (at least) as many steps as columns in the output ODS image. This requires rendering the scene thousands of times per frame, which may be quite slow. You can use larger steps and thicker slits to achieve faster results at the expense of quality.

The vertical field of view should be a full 180 degrees. Spherical and fisheye cameras can produce this directly, but you can also use several perspective cameras to cover the field of view.

See the rasterization example at the end of this document for pseudocode implementing this approach.

Artistic considerations

ODS is a new medium, and we recommend that artists pay special attention to the following aspects of this medium:

- **Content should be omnidirectional**—even if the action is happening in one particular direction, any direction should be viewable. This means the scene must be fully modeled.
- **Scale matters**—if an object is 1 unit tall in one scene and 10 in the next, this will be noticed (unless you also change the IPD to match). The lessons learned from 3D movies apply here; however, the stereo effect in VR is much more compelling, so pay extra attention to it.
- **Avoid nausea**—camera motion, especially acceleration, can cause considerable nausea. The experience is so immersive that if your vision "feels" something that doesn't match the rest of your senses, you get an unpleasant physiological response.

Ray tracing example

For ray tracing, you can create a virtual ODS camera where rays are generated as shown in the following pseudocode example:

```
// Returns the viewing ray for a given pixel.
// x: ranges from 0 (left side of the image) to 1 (right side of the image)
// y: ranges from 0 (top of the image) to 1 (bottom of the image)
// which_camera: either ‘left’ or ‘right’
// IPD: Interpupillary distance, the distance between the eyes.

function [ray_origin, ray_direction] = GetRayForODSCamera(x, y, which_eye, IPD)
  // First convert the pixel coordinates to theta, phi.
  theta = x * 2 * pi - pi;
  phi = pi / 2 - y * pi;

  // Shift the ray origin onto the circle, either to the left or to the right.
  if (which_eye == LEFT_EYE)
    scale = -IPD / 2;
  else
    scale = IPD / 2;
  ray_origin = [cos(theta), 0, sin(theta)] * scale;
  // Ray directions are tangent to the circle.
  ray_direction = [sin(theta) * cos(phi), sin(phi), -cos(theta) * cos(phi)];
end_function
```

Rasterization example

The following pseudocode example shows how to create a virtual ODS camera using a rasterization-based renderer such as OpenGL.

```
// Renders an ODS image using a rasterization-based renderer (e.g. OpenGL).
// This function assumes that ‘renderer.Render()’ will apply a model/view
// transformation to render the scene so that the camera is at the origin
// and the y axis is vertical.

function ods_image = RenderODSImage(renderer, which_eye, IPD, output_width)
  // The ODS image is equirectangular (spherical), with x ranging from (-180,180)
  // degrees and y ranging from (-90,90) degrees.
  output_height = output_width / 2;
  ods_image.resize(output_width, output_height);
  // We will render the ODS image one column at a time, since the ray origin
  // is different for each column. To cover a 180 degree vertical field of view,
  // each column will be rendered from two 90 degree perspective cameras, one
  // looking up at 45 degrees, and one looking down at 45 degrees, and then project
```

```
  // back to equirect using the ProjectToODS function below.
  renderer.SetOutputImageSize(1, output_height / 2);
  renderer.SetVerticalFOVDegrees(90);
  renderer.SetHorizontalFOVDegrees(360 / output_width);

  for column in [0 : output_width - 1]
    x = (column + 0.5) / output_width;  // Add 0.5 to sample the center of the pixel.
    // First render the top 90 degrees.
    [ray_origin, ray_direction] = GetRayForODSCamera(x, 0.25, which_eye, IPD);
    renderer.LookAt(ray_origin, ray_origin + ray_direction);
    renderer.Render();
    ods_image.SubImage(column, 0) =
       ProjectToODS(renderer.ReadImage());
    // Now render the bottom 90 degrees.
    [ray_origin, ray_direction] = GetRayForODSCamera(x, 0.75, which_eye, IPD);
    renderer.LookAt(ray_origin, ray_origin + ray_direction);
    renderer.Render();
    ods_image.SubImage(column, output_height / 2) =
        ProjectToODS(renderer.ReadImage());
  end
end_function

// Projects (warps) a perspective single-column image to an equirect (spherical)
// image. Note that this could also be done in the fragment shader.
function equirect_image = ProjectToODS(perspective_image)
  height = perspective_image.Height();
  equirect_image.resize(1, height);
  for equirect_y in [0 : height - 1]
    // Our camera FOV is pi/2, so map y to (-pi/4, pi/4).
    phi = (equirect_y + 0.5) / height * pi / 2 - pi / 4;
    // Map from angle to perspective y coordinate.
    perspective_y = (tan(phi) * 0.5 + 0.5) * height;
    // We can ignore the projection in x because the image is only one column.
    equirect_image(0, equirect_y) =
        BilinearSample(perspective_image, 0.5, perspective_y);
  end
end_function
```