Stereographics

Stereographics is not new – It’s been in common use in the movies since the 1950s

And, even longer than that in stills

Binocular Vision

In everyday living, part of our perception of depth comes from the slight difference in how our two eyes see the world around us. This is known as binocular vision.

We care about this, and are discussing it, because stereo computer graphics can be a great help in de-cluttering a complex 3D scene. It can also enhance the feeling of being immersed in a movie.

The Cyclops Model

In the world of computer graphics, the two eye views can be reconstructed using standard projection mathematics. The simplest approach is the Cyclops Model. In this model, the left and right eye views are obtained by rotating the scene plus and minus what a Cyclops at the origin would see.

$\psi$

The left eye view is obtained by rotating the scene an angle $+\psi$ about the Y axis. The right eye view is obtained by rotating the scene an angle $-\psi$ about the Y axis. In practice, if you wanted to do this (and you don't), a good value of $\psi$ would be 1-4˚.

The Vertical Parallax Problem

This seems too simple, and in fact, it is. This works OK if you are doing orthographic projections, but if you use perspective, you will achieve a nasty phenomenon called vertical parallax, as illustrated below.

The fact that the perspective shortening causes the black-dot point to have different vertical positions in the left and right eye views makes it very difficult for the eyes to converge the two images. For perspective projections, we need a better way.
Why not just keep using orthographic projections? Mathematically this is fine, but in practice, the two depth cues, stereo and no-perspective, fight each other. This will bring on an optical illusion. A good example of this is a simple cube, drawn below using an orthographic projection:

Because of the use of stereographics, the binocular cues will say that the Near face is closer to the viewer than the Far face is. However, our visual experience reminds us that the only way a far object can appear the same size as a near object is if it is, in fact, larger. Thus, your visual system will perceive the Far face as being larger than the Near face, when in fact they are the same size.

Diversion #1 – Specifying the Viewing Frustum

The OpenGL `glFrustum` call can be used in place of `gluPerspective`: This is meant to look a lot like the `glOrtho()` call.

In the `glFrustum` case, the values of `left`, `right`, `bottom`, and `top` are now the boundaries of the viewing volume on the face of the near clipping plane. `near` and `far` are the same as used in `glOrtho`.

```
void FrustumZ ( float left, float right, float bottom, float top, float znear, float zfar, float zproj )
{
  if( zproj != 0.0 )
  {
    left        *= ( znear/zproj );
    right      *= ( znear/zproj );
    bottom  *= ( znear/zproj );
    top        *= ( znear/zproj );
  }
  glFrustum( left, right, bottom, top, znear, zfar );
}
```

So, if you wanted to view a car from 30 feet away, you could say:

```
FrustumZ(  -10., 10., -10., 10., .1, 100., 30. );
```

But rather than having to specify the left, right, bottom, and top limits at the face of the near clipping plane (which is what `glFrustum` expects), let's setup a way to specify those limits at some convenient distance in front of us. (This is derived using similar triangles.)

Diversion #2 – Where does a 3D Point Map to in a 2D Window?

Take an arbitrary 3D point in the viewing volume. Place a plane parallel to the near and far clipping planes at its Z value (i.e., depth in the frustum). The location of the point on that plane shows proportionally where the 3D point will be perspective-mapped from left to right in the 2D window.

Two Side-by-side Perspective Viewing Volumes

The best stereographics work is done with perspective projections. To avoid the vertical parallax problem, we keep both the left and right eye looking straight ahead so that, in the vertical parallax example shown before, the block-dot point will project with exactly the same amount of perspective shortening.

We now have a horizontal parallax situation, where the same 3D object projects to a very different horizontal position in the left and right eye views.

Note that this is a situation, not a problem. The difference in the left and right eye views requires at least some horizontal parallax in order to work for stereographics. You can convince yourself of this by alternately opening and closing your left and right eyes. We just need a good way to control the horizontal parallax.
Two Side-by-side Non-symmetric Perspective Viewing Volumes

We do this by moving the sides of each eye’s viewing volume to match the left and right boundaries of the cyclops-eye’s viewing volume. We also define a distance in front of the eye, \( z_{0p} \), to the plane of zero parallax. This is where our 3D dot now projects to the same location for each eye’s display.

To the viewer, the plane of zero parallax will be the glass monitor screen and objects in front of it will appear to live in the air in front of the glass screen and objects behind this plane will appear to live inside the monitor.

Left eye viewing volume:
\[
L_{0p} = -z_{0p} \tan(\phi/2) + E
\]
\[
R_{0p} = z_{0p} \tan(\phi/2) + E
\]

Right eye viewing volume:
\[
L_{0p} = -z_{0p} \tan(\phi/2) - E
\]
\[
R_{0p} = z_{0p} \tan(\phi/2) - E
\]

Looking from the Cyclops eye at the origin, determine the left, right, bottom, and top boundaries of the viewing window on the plane of zero parallax as would be used in a call to `glFrustum( )`. These can be determined by knowing \( z_{0p} \) and the original field-of-view angle \( \phi \).

```
void Stereopersp( float fovy, float aspect, float znear, float zfar, float z0p, float eye ) {
    float left, right; // x boundaries on z0p
    float bottom, top; // y boundaries on z0p
    float tanfovy; // tangent of y fov angle
    // tangent of the y field-of-view angle:
    tanfovy = tan( fovy * (M_PI / 180.) / 2. )
    // top and bottom boundaries:
    top = z0p * tanfovy;
    bottom = -top;
    // left and right boundaries come from the aspect ratio:
    right = aspect * top;
    left = aspect * bottom;
    // take eye translation into account:
    left  -= eye;
    right -= eye;
    // ask for a window in terms of the z0p plane:
    FrustumZ( left, right, bottom, top, znear, zfar, z0p );
    // translate the scene opposite the eye translation:
    glTranslatef( -eye, 0.0, 0.0 );
}
```

An Example

<table>
<thead>
<tr>
<th>Parallel viewing stereo</th>
<th>Cross-eyed viewing stereo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>Right</td>
</tr>
<tr>
<td>Right</td>
<td>Left</td>
</tr>
</tbody>
</table>

Often times, Stereographics Images are printed like this so that both Parallel and Cross-eyed Viewing will Work.

Print this page and cut out the left two images.

Places to mount two cameras

Places to mount this bar to a tripod
Acquiring Stereo Photos Yourself

Two lenses

Acquiring Stereo Video

ESPN's 3D camera

Panasonic's 3D Camcorder

Quad-Buffered OpenGL

Remember double buffering, where you draw into the back buffer and display from the front buffer? OpenGL actually has two back buffers and two front buffers, one for each eye. So, draw the left eye view into GL_BACK_LEFT and the right eye view into GL_BACK_RIGHT. First you need to tell GLUT that you are doing stereo graphics. In InitGraphics():

```
glutInitDisplayMode( GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH | GLUT_STEREO );
```

Then go ahead and create the window as normal. After creating the window, you can also expand it to be the full screen with:

```
glutFullScreen(  );
```

In Display(), you need to clear both buffers:

```
glDrawBuffer( GL_BACK_LEFT );
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
```

```
glDrawBuffer( GL_BACK_RIGHT );
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
```

for( int eye = 0; eye <= 1; eye++ )
{
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    if( eye == 0 )  // left eye view
    {
        glDrawBuffer( GL_BACK_LEFT );
        Sterepersp( fovy, 1.0, znear, zfar, z0p, - eyesep );
    }
    else          // right eye view
    {
        glDrawBuffer( GL_BACK_RIGHT );
        Sterepersp( fovy, 1.0, znear, zfar, z0p,   eyesep );
    }

    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    << draw the 3D scene >>
}
```

```
glutSwapBuffers(  ); // this goes outside the eye loop!
```

Separating the Left and Right-eye Views – Shutterglasses

Uses an infrared transmitter to synchronize the left-right of the glasses to the left-right of the screen refresh

Separating the Left and Right-eye Views – Head-mounted Goggles
Separating the Left and Right-eye Views – the Stereo Mirror

Separating the Left and Right-eye Views – Dual Projectors (“GeoWall”)

Two filters statically provide the polarization.

Separating the Left and Right-eye Views – Stereo Movie Projectors

For movies and sporting events.

Separating the Left and Right-eye Views – Stereo Movie Projectors

One filter dynamically provides the polarization (L-R-L-R-L-R per 1/24 sec frame). These are the projectors and glasses that the Corvallis AMC movie theater uses.

Separating the Left and Right-eye Views – VR Headsets

Uses an accelerometer and a gyroscope to know the head position and orientation.

Separating the Left and Right-eye Views – VR Headsets

Uses shaders to get the correct fisheye lens distortion.

Separating the Left and Right-eye Views – View-Master Viewer for your Cell phone

Uses the phone’s gyroscope to know the head orientation.

Separating the Left and Right-eye Views – View-Master Viewer for your Cell phone

Uses a moving magnet and the phone’s digital compass to perform a “left-click.”
Separating the Left and Right-eye Views

Separating the Left and Right-eye Views – Left-Right 3DTV

Separating the Left and Right-eye Views – Top-Bottom 3DTV

Separating the Left and Right-eye Views – Interlaced 3DTV

Separating the Left and Right-eye Views – Red-Cyan Anaglyphs

Encoding Stereo in a Single Image – ChromaDepth™
Stereographics Rules of Thumb

- Stereographics is especially good for de-cluttering wireframe displays.
- Use perspective, not orthographic, projections to avoid the optical illusion.
- Use an eye separation, E, of approximately: \[ E = Z_{0p} \times \tan(1°-4°) \]
- Use the far clipping plane well. The stereo effects are enhanced when the scene is not complicated by a lot of tiny detail that is far away. The interactive response is improved too.
- Because you are drawing the scene twice, using display lists is especially important.
- It is fun to set \( Z_{0p} = Z_{far} \) so that the image appears to be hanging out in the air in front of the monitor. However, in real life we rarely see anything hanging out in the air that has its sides clipped for no apparent reason, as your scene is likely to have. Perceptually, it is often better to set \( Z_{0p} = Z_{near} \) so that the entire scene looks like it is inside the monitor and that you are viewing it through a rectangular hole cut through the glass. This situation is common in everyday life, so we are used to seeing things that way. Personally, I like putting \( Z_{0p} \) about 1/3 of the way through the scene.
- Intensity depth cueing (glFog) nicely enhances the stereo illusion. This also lets you bring the far clipping plane in closer.
- If you are using texture mapping, be sure to use GL_LINEAR, not GL_NEAREST, for the texture filtering.