Proceedings of IDETC/CIE 2006
ASME 2006 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference
September 10-13, 2006, Philadelphia, Pennsylvania, USA

DETC2006-99155

Realtime Dome Imaging and Interaction: Towards Immersive Design Environments

Mike Bailey
Matt Clothier
Oregon State University

Nick Gebbie
University of California San Diego

Abstract

As engineering design becomes more and more complex, we predict that the field will look to immersive environments as a way to create more natural interactions with design ideas. But, helmets are bulky and awkward. A better solution for immersive design is a partial dome. Originally the exclusive domain of flight simulators, dome projection is now being brought to the masses with less expensive dome displays and because its immersiveness makes it such a unique design and display experience. A fisheye lens is needed for the projector to display across the nearly 180° of the dome. This necessarily introduces a distortion of the graphics that is being displayed through it. The trick is to then "pre-distort" the graphics in the opposite direction before sending it on to the projector. This paper describes the use of the OpenGL Shading Language (GLSL) to perform this non-linear dome distortion transformation in the GPU. This makes the development of dome-ready interactive graphics code barely different from developing monitor-only graphics code, and with little runtime performance penalty. The shader code is given along with real examples from our work with San Diego's Reuben H. Fleet Science Center.

1 Previous Work

Projected immersive displays have periodically appeared in graphics research literature. Nelson Max [Max1979] showed how to incorporate dome distortion equations in the display of spheres for immersive molecular modeling. This was a ray-tracing application, so was meant for IMAX movie production, not interaction. Max then used this information to convince various research groups to produce IMAX-distorted computer graphics movie segments for display at SIGGRAPH '84 in Minneapolis-St. Paul. [Max1983] (Collectively, this set of sequences was entitled *The Magic Egg*.)

One of the first immersive projection interaction efforts was the 3, 4, 5, or 6-sided CAVE [Cruz-Neira1993]. The CAVE uses multiple flat-screens arranged in a room configuration. It had display anomalies at the seam between the screens, and synchronization issues among the graphics pipelines driving each screen, but the CAVE demonstrated the advantages of interactive immersion.

Since then, several commercial dome projection products have emerged such as the eLumens VisionDome [Elumens2006] and the SEOS V-Dome [Seos2006]. These systems have created interactive dome projection environments, but are limited in how much scene detail they can display because they implement the dome distortion in software.

To create a hardware solution, Bourke [Bourke2006] used an OpenGL cube map with distorted textures to generate a 3D dome distortion projection. Unfortunately, it required four separate renderings of the scene to produce the cube map, greatly reducing what scene detail could be displayed at interactive rates. We wanted our programs to be able to exploit a hardware solution using only one pass.

2 Dome-Distortion Vertex Shader Code

The dome projector is aimed towards the center of the dome. To be immersive, the 3D scene must be mapped to a hemisphere defined by 90º from this center in all directions. The dome projection algorithm is to convert an (x,y,z) coordinate into spherical angle coordinates (Θ,Φ).

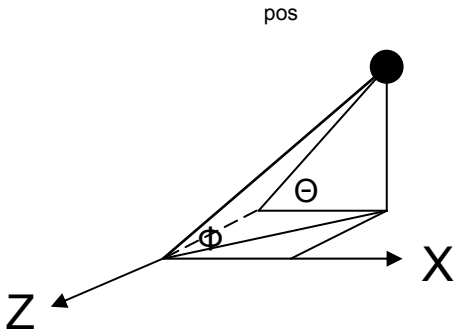


Figure 1a: "God's Eye" View

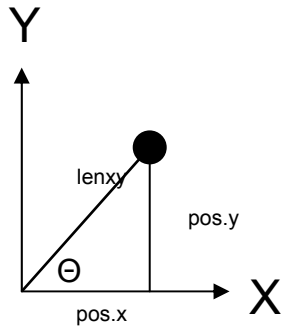


Figure 1b: As the eye sees it

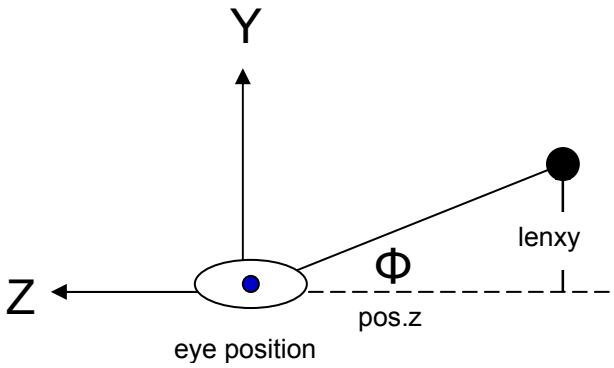


Figure 1c: View from the side

As shown in Figure 1, the angle Θ is the polar "twist" and varies from -180° to +180°. It is preserved through the dome transformation. The angle Φ is the angle away from the centerline of projection and varies from 0° to +90°, where 0° is the centerline and 90° is off to the side. A vertex shader was written to take the place of the standard computer graphics vertex processing. (See [Rost2006] for a complete description of vertex shaders.) This was necessary because the dome distortion is a nonlinear transformation. The vertex shader code is as follows:

```
 1      const float PI =  3.14159265;

 7      void
 8      main( void )
 9      {
10             float phi;
11             vec4 pos = gl_ModelViewMatrix * gl_Vertex;
12             float rxy = length( pos.xy );

13             if( rxy != 0.0 )
14             {
15                    float phi = atan( rxy, -pos.z );
16                    float lens_radius = phi / (PI/2.);
```

```
17                      pos.xy *= ( lens_radius / rxy );
18              }

19              gl_Position = gl_ProjectionMatrix * pos;
20      }
```

In Line 11, **`gl_Vertex`** is the current homogeneous vertex coordinate being plotted and **`gl_ModelViewMatrix`** is the current ModelView transformation matrix. These are multiplied to convert the vertex from world coordinates into eye coordinates.

In Line 15, the angle Φ is computed. Because `rxy` and `-pos.z` are both positive, Φ is in the range [0., π/2]. It is converted to the range [0.,1.] in Line 16, which is the lens radius on which a vertex needs to fall to achieve its proper dome projection. The lens radius is used to scale the eye coordinates in Line 17. Note that the quantities (`pos.x/rxy, pos.y/rxy`) are actually equal to (cos Θ , sin Θ). Also note the SIMD style of programming used by graphics hardware, where identical operations on multiple coordinate values, such as

```
pos.xy *= ( lens_radius / rxy )
```

are performed in one statement. This style matches the internal graphics pipeline hardware architecture, and thus is very efficient.

In Line 19, the eye coordinates are multiplied by the Projection matrix to convert the coordinates into Normalized Device Coordinates, which is what the rest of the pipeline expects. Because of how the shader produces the final eye coordinates, the OpenGL Projection matrix needs to be set as follows:

```
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
glOrtho( -1., 1.,     -1., 1.,     0.0, 1000. );
```

This code appears to be producing an orthographic projection, but it's not. The perspective projection actually takes place in the `atan()` function in Line 15 above. The call to `glOrtho()` here simply preserves it.[1]

3 Nonlinear Radial Distortion

The code in Line 16:

```
float lens_radius = phi / (PI/2.);
```

only works correctly if the fisheye lens has a linear mapping from a point's distance from the len's center to its Φ angle on the dome.. Fortunately for us, the Spitz lens that we used for testing was fairly linear, so that the Line 16 above worked very well. But, in [Max1979], a non-linear radial distortion function for his lens was derived from dome projection tests. A polynomial function was fit through the projection data from the point of view of a ray-tracing program. That is, the polynomial function answered the question "to fill a given pixel in the dome projection, where does my ray need to be aimed?" If that particular additional mapping is needed, additional code can be added to the dome shader. To do that, one would need the inverse of that original mapping, which would then answer the question "where do I need to draw on the lens to make a 3D point look like it's in the right place on the dome?" The code below computes that location. After deriving the constants C1-C5 from the inverse of the lens polynomial function, the required lens radius would be:

$$lens\_radius = c_1\Phi + c_2\Phi^2 + c_3\Phi^3 + c_4\Phi^4 + c_5\Phi^5$$

This would change the shader code to:

[1] The call to `glOrtho()` could actually be eliminated by performing the last of the clip-space Z scaling in the vertex shader. We chose not to do it this way to buffer the application-writer from the details of the hardware's clip-space coordinate system.

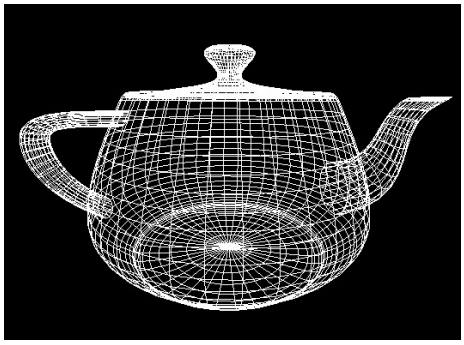
```
 2      const float C1 = ??;
 3      const float C2 = ??;
 4      const float C3 = ??;
 5      const float C4 = ??;
 6      const float C5 = ??;

16      float lens_radius = phi* (C1 + phi* (C2 + phi* (C3 + phi* (C4 + phi*C5) ) ) );
```

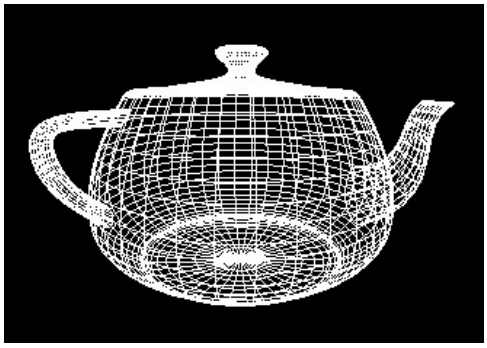
where the constants C1-C5 depend on the optical characteristics of the specific lens in use.

4 Results on a Monitor

The figure below shows the computer graphics wireframe teapot displayed in 3D with the monitor projection and with the dome projection. Both of these images are shown from a monitor.

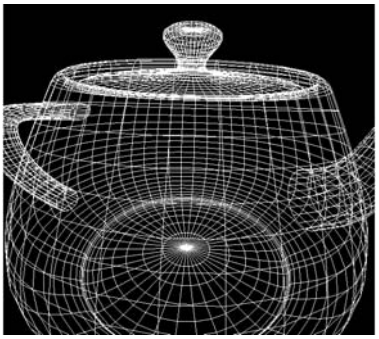


Far Teapot:
Monitor Projection on the Monitor

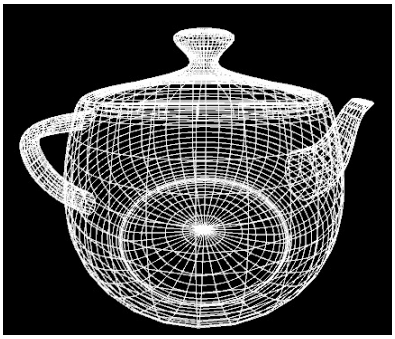


Far Teapot:
Dome Projection on the Monitor

The two figures are very much alike, but this is because they are both small and in the center of the screen. In the center of the dome, the projection surface is approximately flat, and is thus a good approximation to a flat screen. But, make the object bigger so that it encompasses more of the spherical dome, and significant differences emerge. The following two figures show what happens when the teapot is brought closer to the viewer:

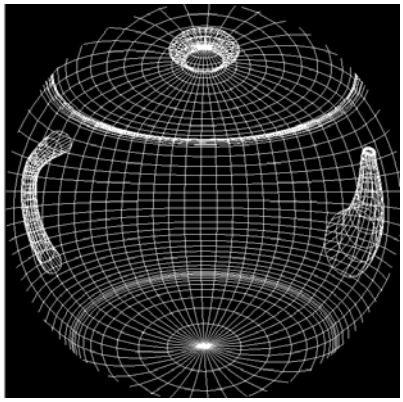


Closer Teapot:
Monitor Projection on the Monitor



Closer Teapot:
Dome Projection on the Monitor

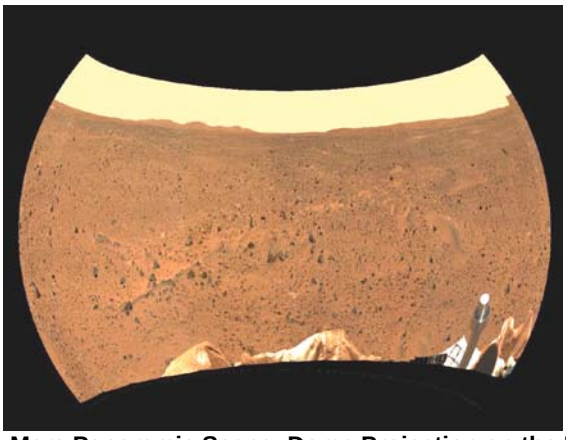
Notice that the front surface of the dome-projection teapot is starting to bow outward. Also notice that the handle and spout of the teapot are clipped in the monitor-projection view, but not in the dome-projection view. One of the interesting aspects of dome projection is that objects never leave the screen to the left, right, bottom, or top. They simply move asymptotically towards the rim of the dome so that they are even with the user. Objects do, however, "leave the screen" by passing over or under, and then behind the user. If we bring the teapot even closer to the viewer, we get the following:



Closest Teapot: Dome Projection on the Monitor

Note that the front face of the teapot has passed behind the viewer and has been clipped away. The center section of the teapot is now even with the viewer, and thus is displayed at the very edge of the dome. Thus, we see a grossly bowed-out teapot on the monitor. But, on the dome, this would look correct. If a user looked to the left, she would see the teapot handle. If she looked to the right, she would see the spout. Looking down and up would show the base and top knob, respectively.

Another application was displaying of panoramic scenes. For this application, we used NASA's Mars 360° panoramic photograph and texture-mapped it to a 3D cylinder composed of 50x50 quadrilaterals, with the eye position in the middle. This image shows the distorted Mars panoramic scene. Like the teapot, the extent of the image shown here is actually a 180º view, from the viewer's left to the viewer's right. Interaction is achieved by rotating eye position inside the cylinder and zooming it closer and farther from the cylinder surface.

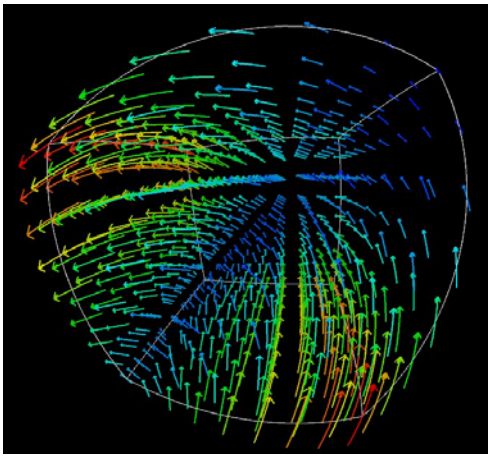


Mars Panoramic Scene: Dome Projection on the Monitor

Similarly, digitized movies can be played this way. To test this, we dynamically texture mapped video streams onto a large polygon in front of the viewer. The images below show this experiment with frames from the IMAX movie *Cosmic Voyage*:

Two Scenes from *Cosmic Voyage*: Dome Projection on the Monitor

Another application was vector field visualization. This is a vector cloud and bounding box, displayed on the monitor with dome distortion.



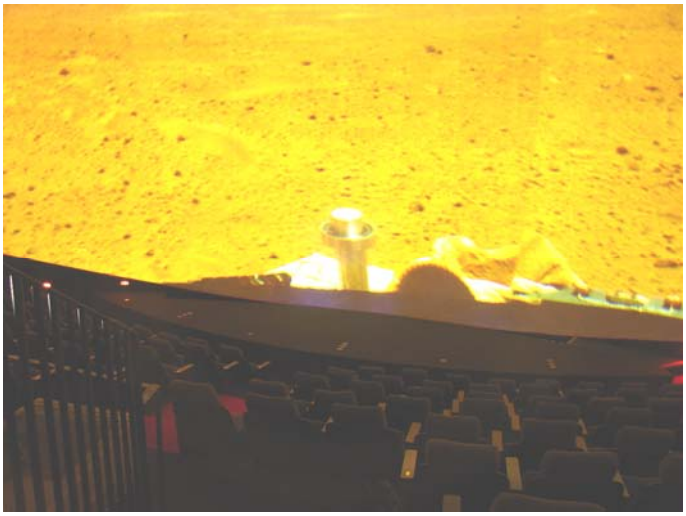
Vector Visualization: Dome Projection on the Monitor

5 Results in the Dome

5.1 Dome Imagery

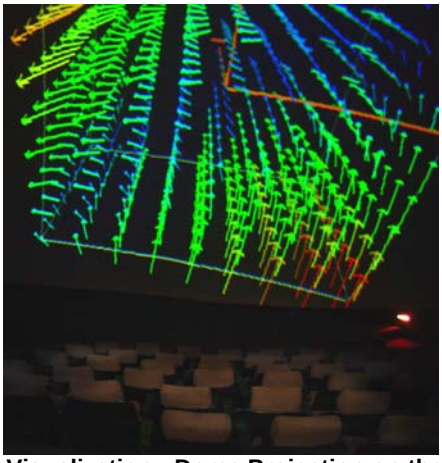
We tested the dome shader in the Reuben H. Fleet Science Center's dome theatre. The RHF theatre uses a tilted dome, aligned 30° below horizontal. We borrowed a JVC QX1G projector and a Spitz fisheye lens. The QX1G projects a 4:3 frame of 2048 x 1536 pixels. It outputs 7000 lumens over the 4:3 frame. However, it only sends the center 1536 x 1536 through the fisheye lens. The projector and lens are shown here.

The interactive Mars panoramic program was run. The image below shows the results. Note that the image does not appear distorted as it did on the monitor. Observing the relatively small theatre seats in the foreground gives a sense of scale.



Mars Panoram: Dome Projection on the Dome

The vector field visualization was also tested. Again, note the scale relative to the seats. This was a very interesting visualization experiment, in that "getting inside" the visualization imparted a strong insight into the patterns going on around us.



Vector Visualization: Dome Projection on the Dome

5.2 Performance

As always, extra functionality does not come for free. For graphics scenes which are vertex-bound, using the dome shader does indeed incur a speed penalty over using the fixed-function pipeline. In our experiments, we have found this to be around 10% across all vertex-bound applications. For example, the per-frame display time for the vector cloud and streamlines was 2.80 ms with the fixed function pipeline and 3.08 ms with the dome vertex shader. This let us interact at the refresh rate of the display system, which is very smooth. For applications that are pixel-bound instead of vertex-bound, this slowing of the vertex processing makes no difference in the overall scene display speed.

6 Lessons Learned

In course of this project, we learned a few things about writing these sorts of applications:

- Large lines must be broken up into shorter line segments, the number of which depends on the dome arc subtended by the two endpoints after they have been transformed by the ModelView matrix. A large line

defined by just the two end vertices will be drawn as a straight line on the monitor, but will map to a curved line on the dome, which is not what was intended. A large line drawn as a series of shorter line segments will look curved on the monitor and straight on the dome, which is correct. Similarly, large polygons need to be tessellated into smaller polygons.[2]

- The first time we tried this on the real dome, we connected the PC directly to the projector. This worked, but forced us to lie on our backs to login because of where the login box showed up on the dome. It also made interaction difficult. The second time, we also brought a 2048x1536 monitor and a 400 MHz VGA splitter. This let us interact while looking at the monitor, knowing that the audience was seeing the correct image on the dome. Similarly, because the projector only projects the center square viewport, we were able to hide non-distorted user interface windows on the sides where we could see them on the monitor, but the audience couldn't see them on the dome.

- We learned to be careful with the dynamics. The combination of large-screen immersion and fast interaction makes some people very dizzy. We learned that we needed to be sure everyone was sitting down before starting.

7 Conclusions

The ability to combine fast, inexpensive graphics with dome projection is a new development. There are several advantages in doing immersive projection this way:

- This creates a single display system interactive immersive projection environment. It is live, and no coordination needs to take place between multiple display systems. It is also a single-pass operation. No display time needs be expended rendering multiple passes.

- All transformations, linear and nonlinear, happen in the spot where hardware is dedicated to doing them. Nothing needs to be moved to CPU software, where it would be much slower. For vertex-bound applications, the speed penalty for using the dome shader was around 10%. There was no speed penalty for pixel-bound applications.

- Dome applications can be written and debugged on a monitor and then easily moved to the dome. The difference is just a handful of lines of code. One program will work for both uses. The software development process can actually be made even easier than that. In one of our experiments, we created a customized version of the GLUT dynamic library which setup the dome transformation transparently to the application. In this way, an executable can be converted from monitor to dome without *any* code changes or re-compilation.

- This is not limited to just spherical screen applications. The same sort of technique could be applied to controlling flat-screen keystoning, projecting on a cylindrical screen, or even correctly projecting on a 3D object.

This has dramatically changed the way we think about public presentation of interactive immersive visualization. This leverages all the years of experience everyone has with interactive OpenGL programming. It turns an immersive dome into "just another display device".

[2] The actual number of line segments in the line break-up could be determined in the CPU, but it would require duplicating the dome transformation mathematics on the CPU, thus eliminating the advantage of doing it in the GPU. The best solution would be to have the GPU vertex shader break up the line and send the required number of line segments into the rest of the pipeline. Unfortunately, GPU vertex shaders can not yet do this sort of vertex-generating operation. We have been told that this is a future enhancement.

8 Acknowledgements

We acknowledge the National Science Foundation under grant 9809224, and the California Institute for Telecommunications and Information Technology, CAL(IT)^2, for funding this project. We thank Bill Licea-Kane from ATI and Randi Rost from 3DLabs for giving us early access to hardware and software that support OpenGL shaders. We thank Ed Lantz and Mark Jarvis from Spitz Ltd., and Bill Bleha from JVC for the loan of the QX1G projector and fisheye lens to test these ideas. We also thank San Diego's Reuben H. Fleet Science Center and its director, Dr. Jeff Kirsch, for letting us conduct these experiments in their dome theatre.

9 References

BOURKE2006 Paul Bourke, *Interactive Fisheye Image Generation,* http://astronomy.swin.edu.au/~pbourke/projection/dome.

CRUZ-NEIRA1993 Carolina Cruz-Neira, Daniel J. Sandin, Thomas A. DeFanti., "Surround-screen Projection-based Virtual Reality: The Design and Implementation of the CAVE", *Proceedings of SIGGRAPH '93*, pp. 135-142, 1993.

ELUMENS2006 http://www.elumens.com

MAX1979 Nelson Max, "ATOMLLL: -- ATOMS with shading and highlights", *Computer Graphics (Proceedings of SIGGRAPH 79)*. Volume 13, Number 3, pp. 165-173, 1979.

MAX1983 Nelson Max, ``SIGGRAPH '84 Call for Omnimax Films',' *Computer Graphics*, Vol. 17, No. 1 (1983) pp. 73-76.

ROST2006 Randi Rost, *The OpenGL Shading Language*, Addison-Wesley, 2006.

SEOS2006 http://www.seos.com