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

DETC2006-99477

Accelerating the Hyperbolic Display of Complex 2D Scenes Using the GPU

Mike Bailey
Oregon State University

Nick Gebbie
University of California San Diego

Abstract

Hyperbolic geometry is a useful visualization technique for displaying a large quantity of two dimensional data simultaneously. What distinguishes this technique is that one particular area can be displayed in full detail with all other areas displayed in lesser detail. In this way, the entire scene is still on the screen so that the entire set of relationships, as well as emergency indicators, can always be seen. But, because the hyperbolic transformation equation is non-linear, it cannot be placed in the standard graphics hardware 4x4 homogeneous matrix. Thus, using the hyperbolic transformation prevents full use of the graphics pipeline hardware, and drastically reduces interactive speed. This paper discusses a way to recoup this display performance by encoding the hyperbolic transform into an OpenGL vertex shader that resides on the graphics GPU hardware. In this way, the hyperbolic transform can still be used interactively, even for very complex 2D scenes.

1 Introduction

There are many applications in which a user wants to display a complex 2D scene. Such applications include mechanical drawings, maps, network diagrams, etc. But, such display applications face a tradeoff in display interaction. It is nice to be able to view the whole scene at once. This allows all spatial relationships to be shown in context. But, at that scale, no detail can be seen. Zooming in is also nice. It provides the detail, but loses the overall context. It would be nice to find a way to do both.

A good solution is to use hyperbolic display techniques [Munzner1995, Walter2003]. A hyperbola is bounded by asymptotes, that is, no matter how far out one goes, the hyperbola approaches the asymptotes but never reaches them. In hyperbolic display techniques, we look for a transformation that will always keep the scene bounded, regardless of how large it gets. But, the transformation that accomplishes this is nonlinear, and thus cannot by used with the standard fast computer graphics hardware display methods. This makes large, intricate hyperbolic scenes slow to display and sluggish to interact with.

2 Polar Hyperbolic Displays

The fundamental principle of using hyperbolic geometry in 2D information visualization displays is to transform all scene geometry, regardless of actual size, into the range [-1.,+1.]. There are several ways to do this, but we have had the best visualization experience with doing it in polar coordinates as described here. The original 2D coordinates are translated so that the region of interest is at the origin. They are then converted to polar coordinates:

$$R = sqrt(X^2+Y^2) \quad (1)$$
$$\Theta = atan2(Y, X) \quad (2)$$

This angle is left unchanged, but the radius is modified so that it never becomes larger than 1.

$$R' = R / (R+K) \tag{3}$$

K is a constant that determines how focused one is on the center of the scene, at the expense of the rest of the scene. Clearly a small K value forces most of the scene to the circumference where R=1. Similarly, a large value of K forces most of the scene to the center where R=0.

Having transformed R into R', the angle is re-incorporated:

$$X' = R' \cos \Theta \tag{4}$$

$$Y' = R' \sin \Theta \tag{5}$$

This can be further simplified by noting that since:

$$\cos \Theta = X / R \tag{6}$$

$$\sin \Theta = Y / R \tag{7}$$

then X' and Y' can be simplified to:

$$X' = X / (R+K) \tag{8}$$

$$Y' = Y / (R+K) \tag{9}$$

The following figures show how this appears visually. Figure 1 below shows a map of the major streets of San Diego, California. There are over 120,000 polylines in the scene, comprising over 320,000 individual line segments. This is a nice overall view, but it is difficult to see street detail in the dense portions of the map. In the following figures, we will zoom in on the downtown area of San Diego, centering on Balboa Park. Figure 2 shows a Euclidean geometry zoom. It reveals more detail, but at the expense of the overall context of the display. In the most-zoomed-in view (Figure 2c), the viewer has completely lost the sense of where they are in the overall map.

Figure 3 shows a polar hyperbolic zoom. It progressively zooms into the region of interest by decreasing the value of the constant K. Note that there is a continual sense of where we are in the overall map.

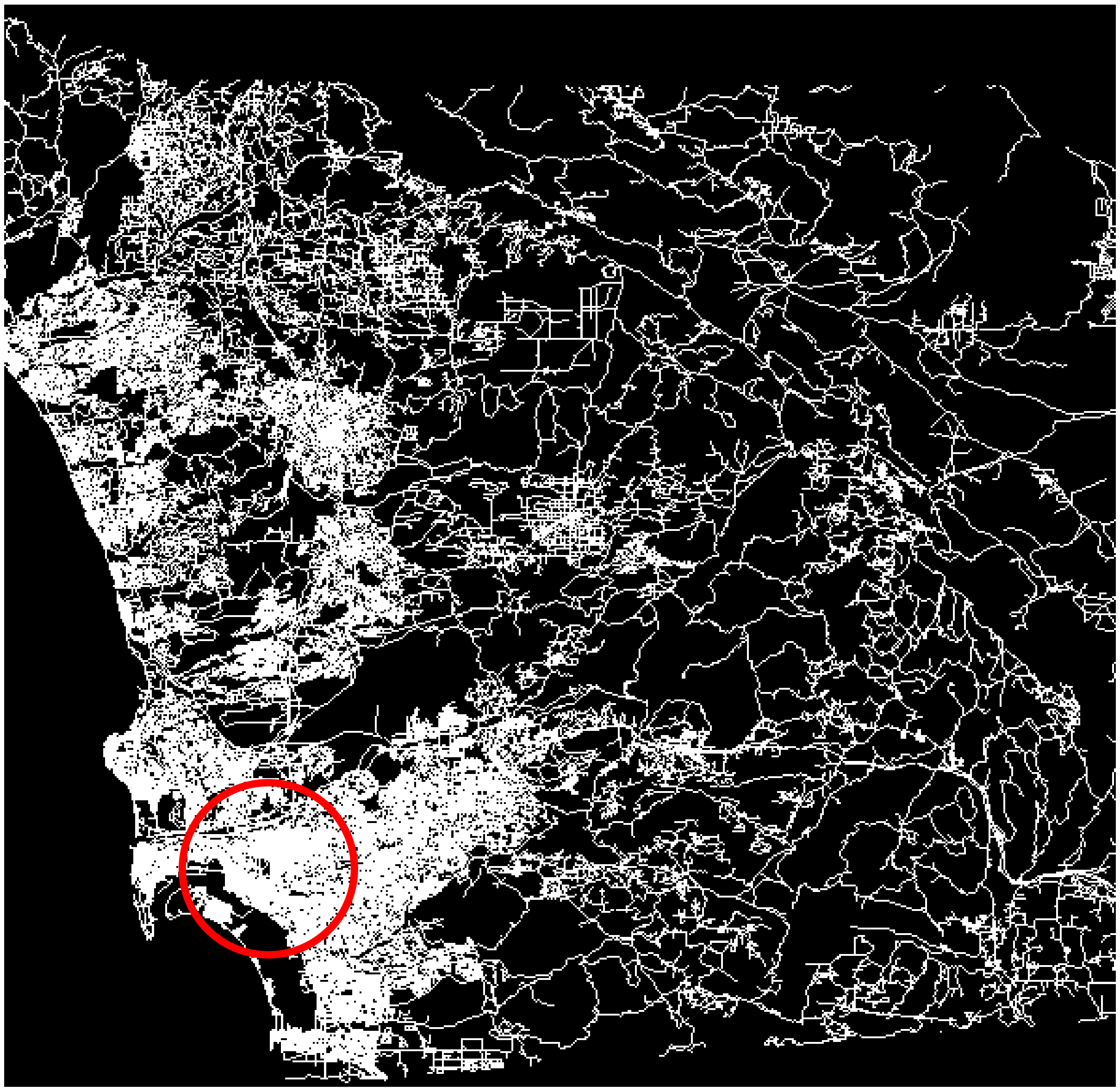


Figure 1: 320,000 Line Segment Scene of San Diego Streets

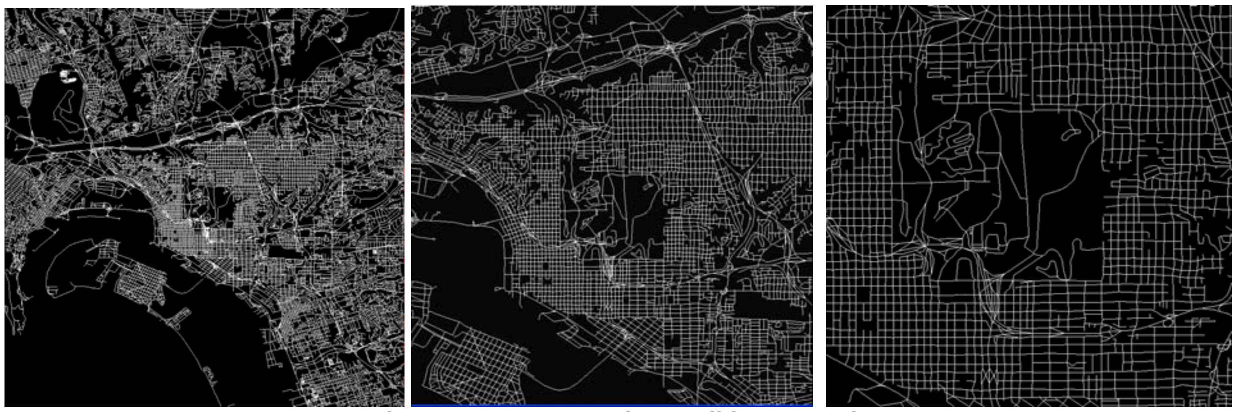


Figure 2a-c: Progressive Euclidean Zooming

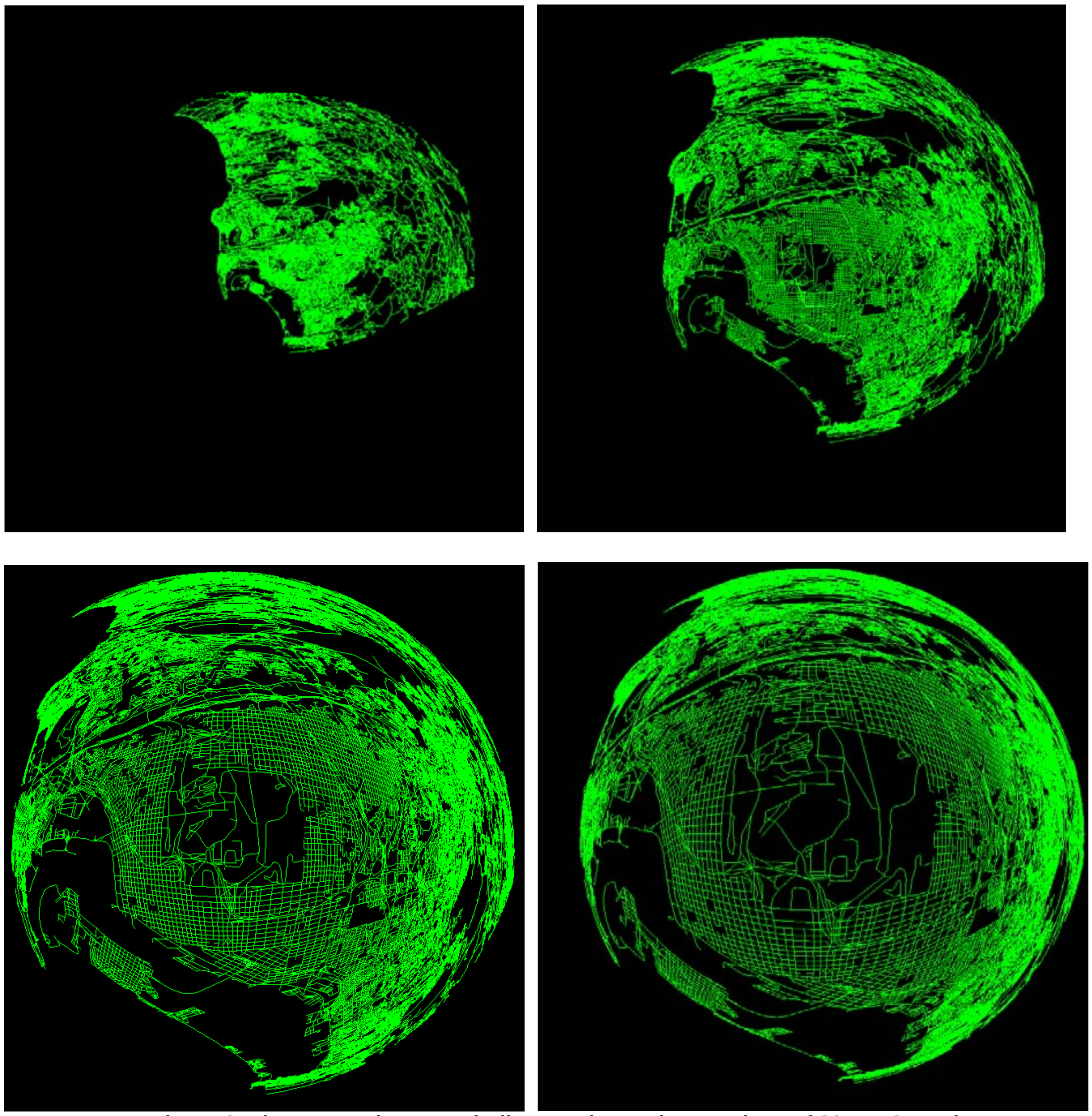
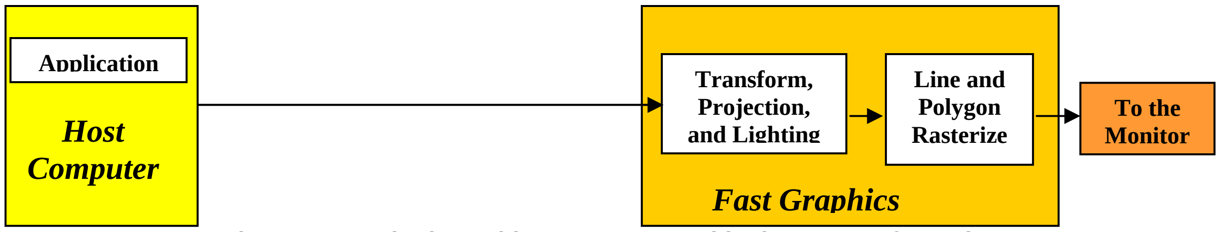


Figure 3a-d: Progressive Hyperbolic Zooming, using K values of 20., 5., 2., and 1.

3 Why It Matters Where the Hyperbolic Transform is Implemented


Application
Host Computer
Transform, Projection, and Lighting
Line and Polygon Rasterize
Fast Graphics
To the Monitor
Figure 4: Standard Graphics Care Usage with Linear Transformations

Today's under-$200 graphics cards are capable of producing dynamic displays that, 10 years ago, were the sole domain of million dollar flight simulators. This by itself creates many opportunities for engineering graphics because we can display large datasets in a very realistic and interactive way. These graphics cards, displaying to a standard monitor or projector, operate as shown in Figure 4. The application generates graphics commands and sends them to the card. The speed of the card comes from implementing the standard 3D graphics functions of transformation, projection, texturing, lighting, and rasterization in hardware.

For nonlinear transformations, such as the hyperbolic transform, life is different. Because this transform is non-linear, it cannot use the graphics pipeline's matrix hardware. Also, because it must come *after* the hardware transform, the hardware transform must be pushed back onto the CPU also, as shown in Figure 5:

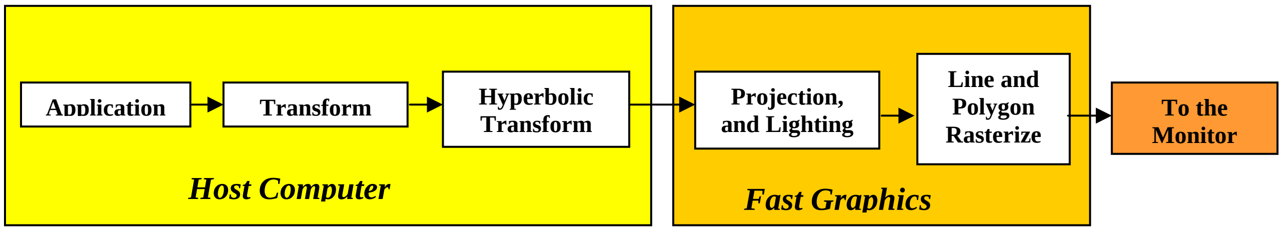


Figure 5: Standard Graphics Card Usage with Nonlinear Transformations

This cripples the display speed. To solve this, we turned to OpenGL vertex and fragment shaders. These shaders allow developers to place their own code in the graphics hardware in place of the vertex processing and fragment processing. We have programmed the dome projection as an OpenGL vertex shader. As shown here, this allows the entire process to return to fast graphics hardware as shown in Figure 6:

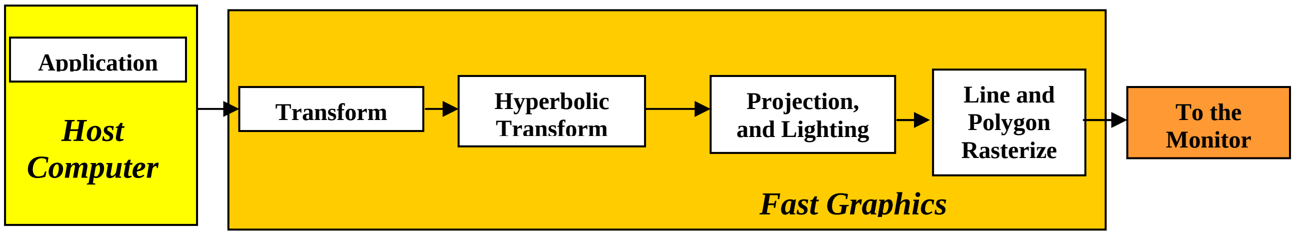


Figure 6: GPU programmable Graphics Card Usage with Nonlinear Transformations

This allows us to have the best of all worlds: (1) large, information-rich datasets; (2) running at interactive speeds; on (3) inexpensive, common, easy-to-develop commodity graphics platforms.

4 Coding it in GLSL

Here's how to code this in an OpenGL vertex shader using the OpenGL Shading Language (GLSL) [Rost2006]. The constant K is passed in as an OpenGL uniform variable. The first step is to apply the model-view matrix transformation to the original (x,y) point. This takes into account any translation, rotation, or scaling transformations that the viewer had wanted. After that, the polar hyperbolic transformation is performed. Finally, the projection matrix transformation is applied and the final point passed on to the rest of the pipeline. The code to do this is shown below:

```
void main( void )
{
  vec4 pos = gl_ModelViewMatrix * gl_Vertex;
  float r = length( pos.xyz );
  pos.xyz /= (r+K);
  gl_Position = gl_ProjectionMatrix * pos;
}
```

`gl_ModelViewMatrix` and `gl_ProjectionMatrix` are pre-defined GLSL variables. `gl_Vertex` is the original coordinate passed by the program, typically with a `glVertex*()` call.

`gl_Position` holds the transformed coordinates that are sent downstream to the rest of the hardware.

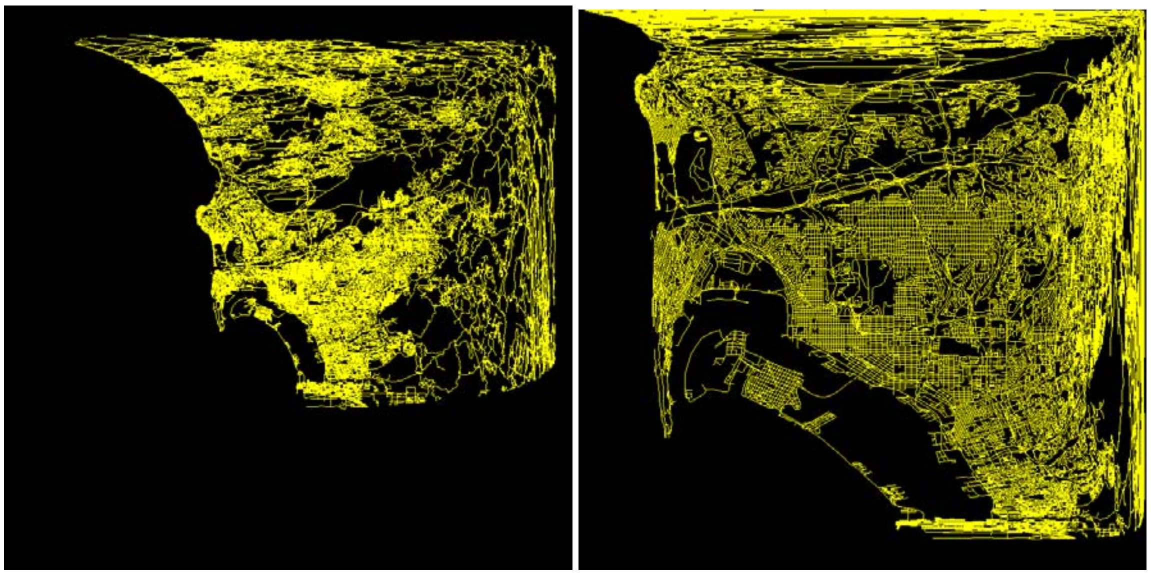
`length()` is a built-in GLSL function that computes the square root of the sum of the squares of the coordinate components of a point. Note the use of the ".xyz" construct. Because graphics engines are constructed as SIMD machines, the shader languages are at their fastest when the code takes advantage of that kind of parallelism.

5 Cartesian Hyperbolic Displays

The polar form of the hyperbolic transform is visually appealing. Because the polar angle, Θ, remains the same within the transformation, angular relationships are maintained. It is also possible, however, to create a hyperbolic display in Cartesian form, like this:

$$X' = X / sqrt(X^2+K^2) \qquad (10)$$
$$Y' = Y / sqrt(Y^2+K^2) \qquad (11)$$

The nature of these equations guarantee that X' and Y' will never leave the region [-1.,1.]. They give results shown here in Figure 7. Notice that, unlike the polar case, the X and Y dimensions can be translated independently, and angular relationships are not maintained.



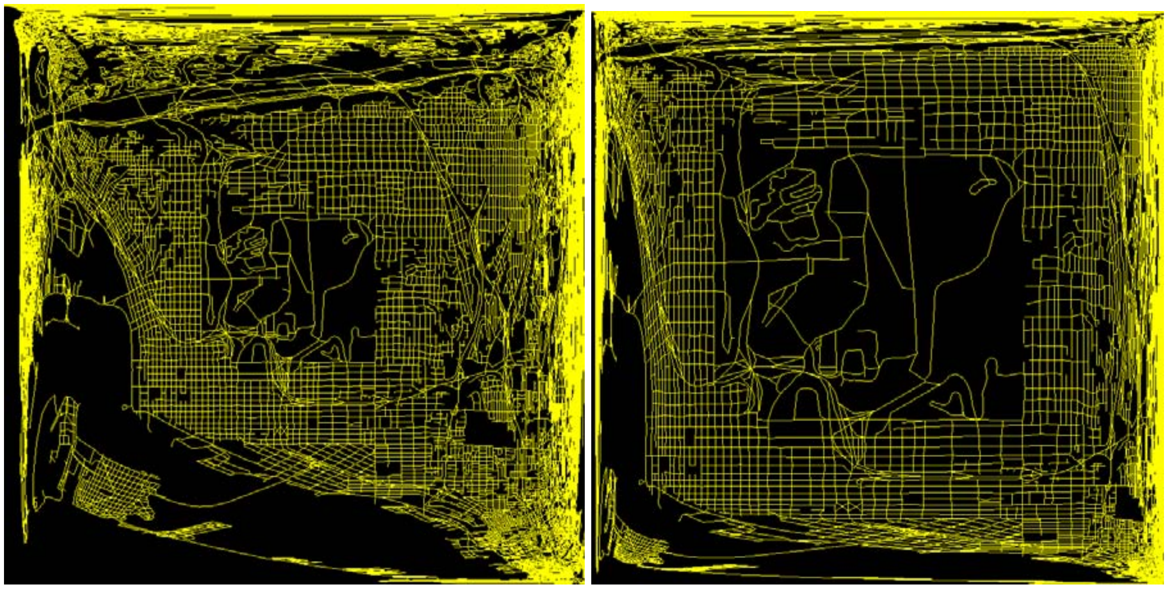
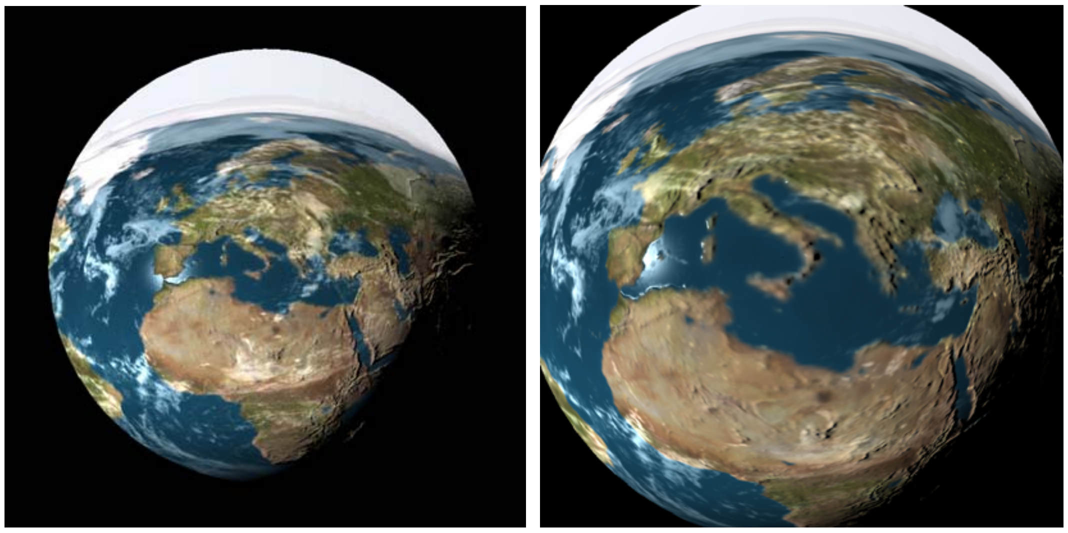


Figure 7a-d: Cartesian Hyperbolic Display with K = 20., 5., 2., and 1.

6 Other Applications

We have also used this technique in terrain-mapping applications. Figure 8 shows how the "boot" of Italy is centered in the window and then the value of K is reduced. The result is a zoom into the boot, but unlike a linear scaling operation, the rest of the scene does not leave the window, it is merely shoved to the outside:



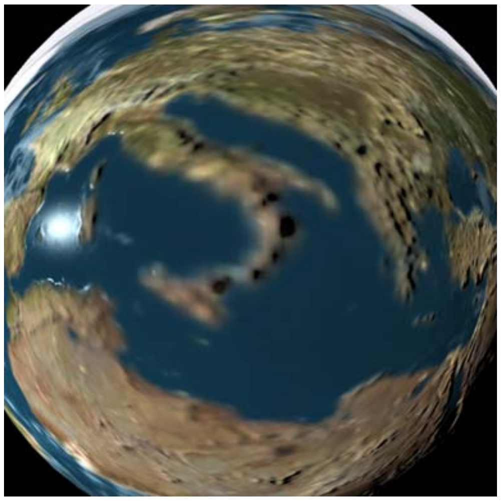


Figure 8a-c: Hyperbolic Zoom into Mediterranean Image

7 Performance

Even though this is a nonlinear transform, the interactive speed is remarkable. But, using the hyperbolic vertex shader does indeed incur a speed penalty over using the fixed-function pipeline to perform Euclidean panning and zooming. For applications that are vertex-bound, the bottleneck is in the vertex processor. In these cases (such as the San Diego map zoom), we have found the performance penalty to be around 10%. For applications that are pixel-bound, the bottleneck is elsewhere and the slight slowing of the vertex processing makes no overall difference in update speed.

8 Conclusions

Hyperbolic geometry is a very useful visualization technique for panning and zooming large complex 2D scenes, while retaining the overall context. Its nonlinear transformation, however, has always forced it to be implemented in the CPU. Worse yet, it has forced the scene transformations to be removed from the GPU and placed in the CPU as well. With GPU programming, we can return both the scene transformation and the nonlinear hyperbolic transformation back to the GPU, resulting in little or no speed penalty.

9 References

MUNZNER1995 Tamara Munzner and Paul Burchard, “Visualizing the Structure of the World Wide Web in 3D Hyperbolic Space”, *Proceedings of VRML '95* (San Diego, California, December 14-15, 1995), special issue of Computer Graphics, pp 33-38, ACM SIGGRAPH.

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

WALTER2003 Jörg Walter, Jörg Ontrup, Daniel Wessling, and Helge Ritter, “Interactive visualization and navigation in large data collections using the hyperbolic space”, *IEEE International Conference on Data Mining, ICDM'03*, pages 355-362, Melbourne, Florida, USA, November 2003.