Using Vertex Shaders for Hyperbolic Geometry
Zooming and Panning Around a Complex 2D Display

- Standard (Euclidean) geometry zooming forces much of the information off the screen
- This eliminates the context from the zoomed-in display
- This problem can be solved with *hyperbolic methods* if we are willing to give up Euclidean geometry
- At one time, this would have also meant severely giving up graphics performance, but not now
Zooming in Euclidean Space

123,101 line strips
446,585 points
Zooming in Polar Hyperbolic Space
Polar Hyperbolic Equations

\[ R' = \frac{R}{R+K} \]
\[ \Theta' = \Theta \]
\[ X' = R' \cos \Theta' \]
\[ Y' = R' \sin \Theta' \]

Overall theme: something divided by something a little bigger

\[ \lim_{K \to 0} R' = 1 \]
\[ \lim_{K \to \infty} R' = 0 \]
Polar Hyperbolic Equations

\[ R = \sqrt{X^2 + Y^2} \]

\[ \Theta = \tan^{-1}\left(\frac{Y}{X}\right) \]

\[ R' = \frac{R}{R + K} \]

Coordinates moved to outer edge when \( K = 0 \)

Coordinates moved to center when \( K = \infty \)

\[ X' = R' \cos \Theta = \frac{R}{R + K} \times \frac{X}{R} = \frac{X}{R + K} \]

\[ Y' = R' \cos \Theta = \frac{R}{R + K} \times \frac{Y}{R} = \frac{Y}{R + K} \]
Cartesian Hyperbolic Equations

\[
\begin{align*}
X' &= \frac{X}{R + K} \\
Y' &= \frac{Y}{R + K}
\end{align*}
\]

Coordinates moved to outer edge when \( K = 0 \)
Coordinates moved to center when \( K = \infty \)

\[
\begin{align*}
X' &= \frac{X}{\sqrt{X^2 + K^2}} \\
Y' &= \frac{Y}{\sqrt{Y^2 + K^2}}
\end{align*}
\]

Coordinates moved to outer edge when \( K = 0 \)
Coordinates moved to center when \( K = \infty \)
Zooming in Cartesian Hyperbolic Space
The vertex shader code provided is designed for version 330 compatibility and can be used to create hyperbolic models. Here's a detailed breakdown of the code:

```glsl
#version 330 compatibility

uniform bool uPolar;
uniform float uK;
uniform float uTransX;
uniform float uTransY;
out vec3 vColor;

void main( void )
{
    vColor = aColor.rgb;

    vec2 pos = ( uModelViewMatrix * aVertex ).xy;
    pos += vec2(uTransX, uTransY);
    float r = length(pos.xyz);

    vec4 pos2 = vec4(0., 0., -5., 1.);

    if( uPolar )
        pos2.xy = pos / ( r + uK );
    else
        pos2.xy = pos / ( pos*pos + uK*uK );

    gl_Position = uProjectionMatrix * pos2;
}
```

The shader begins by defining several uniforms, including `uPolar`, `uK`, `uTransX`, `uTransY`, and an output `vec3` variable `vColor`. The `main` function then uses these uniforms and variables to calculate the final position for the vertex.

The `vec2 pos` variable is computed from the model-view matrix multiplied by the vertex position, then shifted by `uTransX` and `uTransY`. The length `r` of `pos` is also calculated.

The shader then defines a `vec4` variable `pos2` with a set of predefined values, and uses an `if` statement to choose between dividing `pos` by `(r + uK)` or `(pos*pos + uK*uK)` based on the value of `uPolar`. The final position is computed by multiplying `uProjectionMatrix` with `pos2` and storing it in `gl_Position`.

This shader is likely used to create hyperbolic models by modifying the position of vertices in a way that maintains the curvature of the space.
# version 330 compatibility
in vec3 vColor;
out vec4 fFragColor;

void main( )
{
    fFragColor = vec4( vColor, 1. );
}
Corvallis Streets, Buildings, Parks

Kelley Engineering Center