In the Beginning: RenderMan Shaders

- Created ~1983 by Rob Cook and Loren Carpenter from LucasFilm (later spun off as Pixar)
- Big novelty – shaders! Refers to a piece of user-written code that gets embedded in the rendering process somewhere.
- You have seen these many times already. E.g., a checkerboard pattern on a surface, treads on a bicycle tire, light shining through windowpanes projected on the floor, procedural marble textures. The Beauty and the Beast ballroom scene has hundreds of shaders in it.
- The name comes from a take-off on “Walkman”
- Won a Technical Academy Award in 2001
The Goal: Place Olympic Rings on a Sphere

How could you do this in stock OpenGL?

```c
surface olympicsurf( float radius = 0.160, /* outer radius */
                    float thick = 0.050 /* dist to inner radius */ )
{
    float r; /* radius to center of ring */
    point Nf; /* norward-facing surface normal */
    point V; /* vector toward eye */
    color C; /* new color */
    color O; /* new opacity */

    /* transparency possibilities: */
    color clear = color ( 0., 0., 0. );
    color opaque = color ( 1., 1., 1. );

    /* colors of the circles: */
    color blue   = color "rgb" ( 0,0,1 );
    color yellow = color "rgb" ( 1,1,0 );
    color black  = color "rgb" ( 0,0,0 );
    color green  = color "rgb" ( 0,1,0 );
    color red    = color "rgb" ( 1,0,0 );
    color white  = color "rgb" ( 1,1,1 );

    /* centers of the circles in (s,t) space: */
    point pblue = point ( 0.250, 0.600, 0.0 );
    point pyellow = point ( 0.375, 0.400, 0.0 );
    point pblack = point ( 0.500, 0.600, 0.0 );
    point pgreen = point ( 0.625, 0.400, 0.0 );
    point pred = point ( 0.750, 0.600, 0.0 );
}
```
/* where we are right now in (s,t) space: */
point here = point ( s, t, 0.0 );

/* default: color & opacity are whatever was given by the program: */
C = Cs;
O = Os;

/* are we within the blue ring? */
r = distance( here, pblue );
if( (radius-thick) <= r  &&  r <= radius )
{
    C = blue;
    O = opaque;
}

/* are we within the yellow ring? */
r = distance( here, pyellow );
if( (radius-thick) <= r  &&  r <= radius )
{
    C = yellow;
    O = opaque;
}

* * *
How you expose OpenGL-isms: The Built-In Variables

**Vertex Uniform Variables:**
- `gl_ModelViewMatrix` mat4
- `gl_ModelViewProjectionMatrix` mat4
- `gl_ProjectionMatrix` mat4
- `gl_NormalMatrix` mat4
- `gl_TextureMatrix` mat4

**Per-Vertex Attribute Variables:**
- `gl_Vertex` vec4
- `gl_Normal` vec3
- `gl_Color` vec4
- `gl_MultiTexCoord0` vec4
- `gl_MultiTexCoord1` vec4

**Per-Vertex Output Variables:**
- `gl_Position` vec4
- `gl_PointSize` float
- `gl_ClipVertex` vec4

**Fragment Varying Input Variables:**
- `gl_Color` vec4
- `gl_SecondaryColor` vec4
- `gl_TexCoord` vec4
- `gl_FogFragCoord` float

**Fragment Output Variables:**
- `gl_FragColor` vec4
- `gl_FragDepth` float

---

How you expose OpenGL-isms: The Built-In Functions

- `sin( radians );`
- `cos( radians );`
- `atan( y, x );`
- `pow( x, toTheY );`
- `log2( x );`
- `sqrt( x );`
- `inversesqrt( x );`
- `abs( x );`
- `sign( x );`
- `fract( x );`
- `mod( x, y );`
- `min( x, y );`
- `max( x, y );`
- `clamp( x, min, max );`
- `mix( x, y, t );`
- `step( edge, x );`
- `smoothstep( edge0, edge1, x );`
- `length( x );`
- `distance( p0, p1 );`
- `dot( x, y );`
- `cross( x, y );`
- `normalize( x );`
- `ftransform( );`
- `faceforward( N, l, Nref );`
- `reflect( l, N );`
- `noise(x);`
Stripes Shader Example

```
varying vec4 Color;
varying float X;

void 
main( void )
{
  Color = gl_Color;
  X = gl_Vertex.x;
  LightIntensity = ... ;
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

```
varying float X;
varying vec4 Color;
uniform float A;
uniform float P;

void 
main( void )
{
  vec4 WHITE = vec4( 1., 1., 1., 1.);
  float f = fract( A*X );
  gl_FragColor = Color;
  if( f < P )
    glFragColor = WHITE;
  gl_FragColor.rgb *= LightIntensity;
}
```

Varying Variables are Interpolated by the Rasterizer
Stripes Shader Example

Smoothly Transitioning the Stripes

```
float t = smoothstep( 0.5-P-Tol, 0.5-P+Tol, f )  - smoothstep( 0.5+P-Tol, 0.5+P+Tol, f );
gl_FragColor = mix( WHITE, Color, t );
gl_FragColor.rgb *= LightIntensity;
```
What Does All of This Have to do with Visualization?

Dome Shader Example:

Cartesian:  

Dome:
Don't Send Colors to the GPU, Send the Raw Data and let the GPU Assign the Colors

Use the GPU turn the data into graphics on-the-fly

Bump-mapping to Create Apparent Surface Detail

Use the GPU to create the appearance of height without geometrically creating height
Terrain Height Bump-Mapping

Visualization by Nick Gebbie

GPU-Based Dynamic Image Decompression plus Pan and Zoom

Use the GPU to work with aerial and satellite images that were originally larger than what will fit in graphics card memory

Visualization by Dan Moffitt
GPU-Based Dynamic Image Decompression plus Pan and Zoom

Sharpening
Edge Detection

Non-photorealistic Rendering

Use the GPU to enhance scientific and engineering illustration
Image Manipulation Example – Where is it Likely to Snow?

Visible Infrared Water vapor

Visualization: Point Clouds

Can change:
• Color
• Alpha
• Pointsize

Use the GPU to interactively change the appearance of 3D data
Visualization: Extruding Shapes Along Flow Lines

Use the GPU to show flow information.

Add moving "humps" to create a peristaltic effect.

Visualization: 2D Line Integral Convolution

At each fragment:
1. Find the flow field velocity vector there
2. Follow that vector in both directions
3. Blend in the colors at the other fragments along that vector
Visualization: 3D Line Integral Convolution

Finite Automata Computations in the GPU

Use the GPU as a general-purpose computer to compute the time steps of a simulation.

We implemented John Conway's Game of Life - and achieved 300M computed pixels per second.
Conclusions

- GPU programming is one of the most exciting things that has ever happened to CG
- It enables application developers to have very tight control over graphics effects without sacrificing display performance
- It was really made for game development, but it has significant applications in visualization, imaging, and scientific computing
- OSU is one of the few universities that has an organized course in GPU Programming - CS 519 - next offered in Spring Quarter 2008. The prerequisite is having taken any of the other CS graphics classes. (We jump right into the graphics pipeline.)