GLSL Review

Monday, Nov. 28 2011
Aggregate per-vertex data such as location, normal, color, tex coords, etc.
OpenGL pipeline

- Command Stream
- Vertex Processing
- Geometry processing
- Rasterization
- Fragment processing
- Fragment Ops/Blending
- Display!

Perform model-view, perspective, normal translations to vertices and normals
OpenGL pipeline

Break down quads, lines, points, etc. into triangles for easy interpolation in the rasterizer. Custom shaders can actually add new primitives to the pipeline here.
OpenGL pipeline

- Command Stream
- Vertex Processing
- Geometry processing
- Rasterization
- Fragment processing
- Fragment Ops/Blending
- Display!

Divide triangle surface into per-pixel fragments, interpolating attributes from triangle vertices to each fragment.
Use normal, color, texture information to assign color to fragments
OpenGL pipeline

Command Stream → Vertex Processing → Geometry processing → Rasterization → Fragment processing → Fragment Ops/Blending → Display!

Blend overlapping fragments from different shapes according to transparency and OVER operator.
OpenGL pipeline

Send finalized framebuffer contents back to the CPU!
What can shaders do?

• More complex material shading models
• More realistic surface simulation using fractals, etc.
• Arbitrary modification of geometry

• Why do it on the GPU instead of the CPU? ...
What can shaders do?

Why do it on the GPU instead of the CPU?
– Parallelism
– Processor / bus highly optimized for graphics tasks
Shaders

• Access through STShaderProgram
• If we replace OpenGL’s default shaders, we *must* replicate all necessary functionality from the GL pipeline (i.e. transforming vertices to clip coordinates, assigning colors to fragments.)
Vertex Shader

• Input: untransformed vertex (as entered by GL programmer), vertex data (such as color, normals, etc.)

• Output: model-view and projection-transformed vertex, inverse transpose-transformed normal

• This is our chance to pass any model-related position information to the fragment shader!
Rasterize stage:
(not programmable, but important)

Takes **triangles** (sets of 3 vertices) and breaks them into **fragments** (pixel-sized tiles covering the surface of the triangle.)

- Vertex data is interpolated to produce fragment data
- Each fragment sent to the fragment shader
Fragment shader

• Can use any information passed from the vertex shader to calculate `gl_FragColor`
• Can access textures
• Blending (for semi-transparent, overlapping fragments) occurs later.
ModelView matrix

• Transforms objects into “eye” space
• This space is the physical model of the environment we’ve programmed: the locations, vector magnitudes, and relative angles of all objects we are drawing are defined in this space.
• The camera, at (0, 0, 0) is the “eye”
Projection Matrix

• Transforms vertices from eye coordinates into clip coordinates
  – Clip coordinates of objects within the image are in the \([-1, 1] \times [-1, 1] \times [0, 1]\) box.

• If model-view matrix places objects and camera into eye space, Projection matrix chooses the camera lens, sets camera fov and takes picture.
Projection Matrix

• OpenGL has 2 kinds of projection: Orthographic and Perspective
• We can create custom projection matrices to simulate different types of lenses (e.g. fisheye projection)
Projection Matrix

• Not guaranteed to preserve relative vector lengths or angles between vectors from eye space (In fact, probably won’t!)

• Is it possible to compute physical properties of the model (for shading purposes) without this metric information?