Graphics Processing

Georgi Chunev

IU-B 2009
1. Review of the OpenGL pipeline

Input: Vertices, Attributes, Textures

Output: a 2D pixel array
2. History of Graphics Hardware

- Pixel Pipelines
- Hardware Accelerated T&L (e.g. GF 256)
- Configurable vs. Programmable GPUs (e.g. GF3)
- Unified Shader Architecture GPUs (e.g. GF8)
- The Fermi Architecture
Pixel Pipelines

- 3dfx Voodoo (1996) – A single pipeline 3D accelerator (required a separate 2D VGA)

- nVidia Riva (1997) – One of the first integrated 2D and 3D videocards.
Hardware Accelerated T&L

• The technical definition of a GPU is:
  “A single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second.“

• GeForce 256 and Savage 2000 – the first GPUs
Transform & Lighting

- MV matrix
  
  ```
  vec4 transformed = gl_ModelViewMatrix * gl_Vertex;
  ```

- Projection Matrix
  
  ```
  vec4 projected = gl_ProjectionMatrix * transformed;
  ```

- Normal Matrix
  
  ```
  vec3 transformed_normal = gl_NormalMatrix * gl_Normal;
  ```

- Texture Matrix
  
  ```
  gl_TexCoord[0] = gl_TextureMatrix[0] * gl_MultiTexCoord0;
  ```

- Gouraud Shading
  
  per-Vertex Lighting
The Normal Matrix

• Issue:

\[ \text{dot}(T, N) = 0 \quad \text{dot}(T', N') \neq 0 \]

• Solution: \(T' = MV \cdot T\)

\[ N' = (MV^{-1})^T \cdot N = NM \cdot N \]

Note: \(NM\) can be a 3x3 matrix, since \(N_w = N'_w = 0\).
Texture Matrix

• Transforming texture coordinates is like transforming geometry in texture space!
Gouraud Shading

void main(void)
{
    // vertex MVP transform
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    // eye-space normal
    N = normalize(gl_NormalMatrix * gl_Normal);
    // vertex MV transform
    vec4 V = gl_ModelViewMatrix * gl_Vertex;
    // vector to the light source
    vec3 L = normalize(gl_LightSource[0].position.xyz - V.xyz);
    // average of the L and Eye vectors
    vec3 H = normalize(L + vec3(0.0, 0.0, 1.0)); // assuming the eye(camera) is in the +z direction

    float d_intensity = max(0.0, dot(N, L));
    float s_intensity = pow(max(0.0, dot(N, H)), gl_FrontMaterial.shininess);
    gl_FrontColor.rgb = (gl_FrontMaterial.diffuse.rgb * gl_FrontLightProduct[0].diffuse.rgb * d_intensity) +
    (gl_FrontMaterial.specular.rgb * gl_FrontLightProduct[0].specular.rgb * s_intensity) +
    (gl_FrontMaterial.ambient.rgb * gl_FrontLightProduct[0].ambient.rgb);
    gl_FrontColor.a = gl_Color.a; // or gl_FrontColor.a = 1.0;
}
Definition: a fragment describes the information associated with a pixel prior to the generation of the pixel.
The GeForce 6 Architecture
The need for unified shader designs, based on scalar stream processors

Why unify?
- Vertex Shader
- Pixel Shader

Unified pipeline
- Geometry
- Physics
- Floating Point Processor
- ROP
- Memory

Heavy Geometry
Workload Perf = 4

Heavy Pixel
Workload Perf = 8

Unified Design
Unified Shader Architecture GPUs

Specifications:
1.35 GHz cores
up to 512 threads/block
32 threads/warp
4 cycles/warp for most arithmetic instructions
up to 768 Threads/SM
up to 24 warps/SM
up to 8 blocks/SM
AG80 Scalar Stream Multiprocessor

• Increased hardware utilization due to:
  • Shader unification
  • Use of scalar processors
  • Decoupled texture and math operations
  • Some shared memory
  • Broadcast capabilities for shared memory lookups

• Has issues with:
  • Lack of fast shared memory between all multiprocessors
  • Lack of control over the thread scheduling process
  • No access to the lowest level code 😞
The Fermi Architecture

• Full Cache hierarchy:
3. GLSL

• Overview
  • Vertex Shaders
  • Geometry Shaders
  • Fragment Shaders
  • Compiling and Using a GLSL program from OpenGL

Note: GLSL is an extension to OpenGL. To access it, use GLEW for instance:

```c
// In main call:
glewInit();
// Before compiling your shaders check:
glewIsSupported("GL_VERSION_1_4 GL_ARB_vertex_shader\GL_ARB_fragment_shader\GL_EXT_geometry_shader4\GL_ARB_shader_objects\GL_ARB_shading_language_100");
```
Shader Data Characteristics

• Attribute variables - per-vertex input to a Vertex shader from the application (READ-ONLY)

• Uniform variables - input to Vertex and Fragment shaders from the application (READ-ONLY). Also, textures are accessed through uniform samplers of 1, 2, or 3 dimensions (e.g. `uniform sampler2D tex;`)

• Varying variables - output from a Vertex shader (READ/WRITE), which is interpolated, and then input to a Fragment shader (READ-ONLY)
A Simple Vertex Shader

At the very least, a vertex shader needs to write to the predefined output variable gl_Position, which is needed by the subsequent stages of the pipeline.

//Used Inputs:
// uniform mat4 gl_ModelViewProjectionMatrix;
// attribute vec4 gl_Vertex;
// attribute vec4 gl_Color;

//Produced Outputs:
// vec4 gl_Position;
// varying vec4 gl_FrontColor;

//The Shader:
void main(void)
{
    // vertex MVP transform
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    // pass the source color further down the pipeline
    gl_FrontColor = gl_Color;
}
A Simple Geometry Shader

Geometry shaders come with two special functions EmitVertex() and EndPrimitive(), which are used for the generation of geometry in the scene. Calling EndPrimitive(), when possible, creates a primitive from the outstanding emitted vertices.

```cpp
// Used Inputs:
// int gl_VerticesIn;  // number of input vertices
// varying in vec4 gl_PositionIn[gl_VerticesIn];
// varying in vec4 gl_FrontColorIn[gl_VerticesIn];

// Produced Outputs:
// vec4 gl_Position;  // the shader must write to this variable
// varying out vec4 gl_FrontColor;

// The Shader:
#version 120
#extension GL_EXT_geometry_shader4 : enable
void main(void)
{
    for (int i=0; i <= gl_VerticesIn-1; i++) {
        gl_Position = gl_PositionIn[i];
        gl_FrontColor = gl_FrontColorIn[i];
        EmitVertex();
    }
    EndPrimitive();
}
```
## Geometry Shader I/O Types

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Allowed GL primitives</th>
<th># of Input Vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points</td>
<td>POINTS</td>
<td>1</td>
</tr>
<tr>
<td>Lines</td>
<td>LINES, LINE_STRIP, LINE_LOOP</td>
<td>2</td>
</tr>
<tr>
<td>Lines with Adjacency</td>
<td>LINES_ADJACENCY, LINE_STRIP_ADJACENCY</td>
<td>4</td>
</tr>
<tr>
<td>Triangles</td>
<td>TRIANGLES, TRIANGLE_STRIP, TRIANGLE_FAN</td>
<td>3</td>
</tr>
<tr>
<td>Triangles with Adjacency</td>
<td>TRIANGLES_ADJACENCY, TRIANGLE_STRIP_ADJACENCY</td>
<td>6</td>
</tr>
</tbody>
</table>

*Table 1: Shows the allowed GL primitives and the number of accepted vertices for a geometry shader defined for a certain input type.*

- Note: GL_QUADS are processed as GL_TRIANGLES
A Simple Fragment Shader

At the very least, a fragment shader needs to write to the predefined output variable `gl_FragColor`.

```
// Used Inputs:
// varying vec4 gl_Color;

// Produced Outputs:
// vec4 gl_FragColor;

// The Shader:
void main(void)
{
    // pass the source color further down the pipeline
    gl_FragColor = gl_Color;
}
```
Compiling GLSL code

GLhandleARB vShader, gShader, fShader;
GLcharARB *vsStringPtr[1], *gsStringPtr[1], *fsStringPtr[1];
GLcharARB vsString[] = "Vertex Shader source goes here"
GLcharARB gsString[] = "Geometry Shader source goes here"
GLcharARB fsString[] = "Fragment Shader source goes here"

vShader = glCreateShaderObjectARB(GL_VERTEX_SHADER_ARB);
vsStringPtr[0] = vsString; // Do this to avoid warnings
glShaderSourceARB(vShader, 1, vsStringPtr, NULL);
glCompileShaderARB(vShader);

gShader = glCreateShaderObjectARB(GL_GEOMETRY_SHADER_EXT);
gsStringPtr[0] = gsString; // Do this to avoid warnings
glShaderSourceARB(gShader, 1, gsStringPtr, NULL);
glCompileShaderARB(gShader);

fShader = glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);
fsStringPtr[0] = fsString; // Do this to avoid warnings
glShaderSourceARB(fShader, 1, fsStringPtr, NULL);
glCompileShaderARB(fShader);
Linking GLSL Code

// Create a program object, attach shaders, then link
GLhandleARB progObj = glCreateProgramObjectARB();
glAttachObjectARB(progObj, vShader);
glAttachObjectARB(progObj, gShader);
glAttachObjectARB(progObj, fShader);

// Geometry Shader Setup
glProgramParameteriEXT(progObj, GL_GEOMETRY_INPUT_TYPE_EXT, GL_TRIANGLES);
glProgramParameteriEXT(progObj, GL_GEOMETRY_OUTPUT_TYPE_EXT, GL_TRIANGLES);

// 64 is just an example for the maximum number of output vertices
GLint numOutVertices = 64;
glProgramParameteriEXT(progObj, GL_GEOMETRY_REGIONS_OUT_EXT, numOutVertices);

glLinkProgramARB(progObj);
Using GLSL Shaders

//Attach the shading program:
glUseProgramObjectARB(progObj); // Pass 0 to switch back to fixed functionality

//To initialize the uniform variable varName with 0, do:
GLint value = 0;
GLint address = glGetUniformLocationARB(progObj, "varName");
if (address != -1)
    glUniform1iARB(address, value);
else fprintf(stderr, "Failed to load uniform\n");

Caution:
1) Unused variables get optimized out from the shaders during compilation
2) The types of value, varName, and glUniform[1,2,3,4][i,f]{v} MUST match
3) For texture samplers (e.g. uniform sampler2D tex;) use glUnifrom1iARB, and pass the texture unit index as a parameter. Inside the shader use texture[1,2,3]D (e.g. texture2D(tex, gl_TexCoord[0].st))
GLSL Reference Guide

• Link
Shading Examples

- Gouraud Shading (Vetrex Shader)

- **Phong Shading** (Fragment Shader)
  - Note: Interpolated normals are not unit length, so you need to renormalize them in the fragment shader!
4. Ray-Tracing

• Overview
  • Basic Ray-Tracing
  • Reflection Model
  • Smooth Shadows
Defining the Image Plane (frame)
Defining the Image Plane (frame)

For a camera looking down the –w axis:

\[ u_s = l + (r-l) \times (i + 0.5)/n_x; \]
\[ v_s = b + (t-b) \times (j + 0.5)/n_y; \]
\[ w_s = -n; //near clipping plane \]

We can parametrize a ray \( s \) as:

\[ s(t) = t \times (u_s, v_s, w_s), \]
and look for objects that intersect \( s \) at \( t>1 \), and \( t<=(far/n) \), where \( far \) is the \( w \) coordinate of the far clipping plane.

In general: \( s(t) = \text{eye} + t \times ((u_s, v_s, w_s) – \text{eye}); \)
Intersecting a sphere

An implicit equation for a sphere:

\[(p - c) \cdot (p - c) - R^2 = 0\]

Take: \[p = \text{eye} + t \star ((u_s, v_s, w_s) - \text{eye}) = \text{eye} + td\]
Solve for \(t\) in:

\[(d \cdot d)t^2 + 2d \cdot (\text{eye} - c)t + (\text{eye} - c) \cdot (\text{eye} - c) - R^2 = 0\]
Intersecting a triangle

For a triangle with points \( p_0, p_1, p_2 \), and normal \( n \), solve for \( t \) in: \((s(t) - p_0) \cdot n = 0\)

\[
t = \frac{((p_0 - \text{eye}) \cdot n)}{(d \cdot n)}
\]

To detect an intersection:

1) Make sure that \((d \cdot n) \neq 0\);
2) Compute \( t \) by the above formula
3) Compute \( s(t) \) – the intersection point with the triangle’s plane
4) Make sure that the following inequalities hold:
   \[
   ((p_1 - p_0) \times (s(t) - p_0)) \cdot n > 0
   
   ((p_2 - p_1) \times (s(t) - p_1)) \cdot n > 0
   
   ((p_0 - p_2) \times (s(t) - p_2)) \cdot n > 0
   \]
Basic Ray-Tracer Implementation

Using a software renderer

Algorithm for a single ray and a single light source:
1) Trace the ray from the eye through the frame and to an objects inside the clipping volume
2) Take the intersection point with the nearest intersected object
3) Trace a ray from the light source to the intersection point
4) If the light ray make it to the intersection point, then compute color for a lit pixel at the given frame location.

Caution:
- Avoid self Intersections!
- Consider back-faces!
Reflection

\[ r = d - 2(d \cdot n)n \]
Soft Shadows

Problem:
Point lights produce hard shadows.

Solution:
Use an array of point lights instead.

Problem:
Tracing multiple lights is slow and yet does not remove all sharp transitions in the penumbra(p).

Solution:
Use an area light and sample it randomly.