Programmable Graphics Pipelines

Topics

The fixed-function graphics pipeline
Programmable stages
  ■ Vertex shaders
  ■ Fragment shaders
GL shading language (GLSL)
Mapping other applications to GPUs
The Graphics Pipeline

A Trip Down the Graphics Pipeline

Command
  ↓
Vertex
  ↓
Assembly
  ↓
Rasterization
  ↓
Fragment
  ↓
FB ops
  ↓
Display
  ↓
Texture
Application

Simulation
Input event handlers
Modify data structures
Database traversal
Primitive generation
Graphics library utility functions (glu*)

Command

Command queue
Command interpretation
Unpack and perform format conversion
Maintain graphics state

```c
glLoadIdentity();
glMultMatrix(T);
glBegin(GL_TRIANGLE_STRIP);
glColor3f(0.0, 0.5, 0.0);
glVertex3f(0.0, 0.0, 0.0);
glColor3f(0.5, 0.0, 0.0);
glVertex3f(1.0, 0.0, 0.0);
glColor3f(0.0, 0.5, 0.0);
glVertex3f(0.0, 1.0, 0.0);
glBegin();
glEnd();
```
**Vertex (per-vertex)**

- Vertex transformation
- Normal transformation
- Texture coordinate generation
- Texture coordinate transformation
- Lighting (light sources and surface reflection)

---

**Primitive Assembly**

- Combine transformed/lit vertices into primitives
  - 1 vert -> point
  - 2 verts -> line
  - 3 verts -> triangle

- Clipping
- Perspective projection
- Transform to window coordinates (viewport)
- Determine orientation (CW/CCW)
- Back-face cull
Rasterization

Setup (per-triangle)
Sampling (triangle = {fragments})
Interpolation (interpolate colors and coordinates)

Texture

Textures are arrays indexed by floats (Sampler)
Texture address calculation
Texture interpolation and filtering
Fragment

Combine texture sampler outputs
Per-fragment shading

Framebuffer Operations

Owner, scissor, depth, alpha and stencil tests
Blending or compositing
Display

Gamma correction
Analog to digital conversion

Framebuffer Pixels

Light

Programming Stages
Programmable Graphics Pipeline

- Command → Vertex → Transform → Inputs
  - Assembly → Lighting → Lighting
  - Rasterization → Texture → Outputs
  - Fragment → Shader Program
- FB ops → Display

Shader Program Architecture

- Inputs → Registers
  - Registers → Shader Program
  - Shader Program → Outputs
  - Shader Program → Texture
  - Texture → Constants
  - Constants
What’s in a GPU?

GLSL
Simple Vertex and Fragment Shaders

// simple.vert
void main()
{
    gl_Position =
        gl_ModelViewMatrix * 
            gl_ProjectionMatrix * gl_Vertex;
    gl_Normal = gl_NormalMatrix * gl_Normal;
    gl_FrontColor = gl_Color;
    gl_BackColor = gl_Color;
}

// simple.frag
void main()
{
    gl_FragColor = gl_Color
}

Uniform Variables

Uniforms are variables set by the program that can be changed at runtime, but are constant across each execution of the shader;
Changed at most once per primitive

    // Predefined OpenGL state
    uniform mat4 gl_ModelViewMatrix;
    uniform mat4 gl_ProjectionMatrix;
    uniform mat4 gl_NormalMatrix;

    // User-defined
    uniform float time;
Attribute Variables

Attributes variables are properties of a vertex. They are the inputs of the vertex shader.

```c
attribute vec4 gl_Color;
varying vec4 gl_FrontColor;
varying vec4 gl_BackColor;

void main() {
    gl_FrontColor = gl_Color;
}

N. B. All glVertex*() calls result in a vec4
```

Varying Variables

Varying variables are the outputs of the vertex shader.

```c
attribute vec4 gl_Color;
varying vec4 gl_FrontColor;
varying vec4 gl_BackColor;

void main() {
    gl_FrontColor = gl_Color;
}
```
Varying Variables

The varying variables are interpolated across the triangle

gl_Color is set to gl_FrontColor or gl_BackColor depending on whether the triangle is front facing or back facing

```cpp
varying vec4 gl_Color;
vec4 gl_FragColor;

void main() {
  gl_FragColor = gl_Color;
}
```

Vectors

Constructors

```cpp
vec3 V3 = vec3(1.0, 2.0, 3.0);
vec4 V4 = vec4(V3, 4.0);
```

Swizzling

```cpp
vec2 V2 = V4.xy;
vec4 V4Reverse = V4.wzyx;
vec4 Result = V4.xyzw + V4.xxxx;
```

Basic Vector Operators

```cpp
float Result = dot(V4, V4Reverse);
vec3 Result = cross(V3, vec3(1.0, 0.0, 0.0));
```

N. B. Points, vectors, normals and colors are all vec’s
Textures

```
uniform sampler2D SomeTexture;

void main()
{
    vec4 SomeTextureColor =
        texture2D(SomeTexture, vec2(0.5, 0.5));
}
```

N. B. Textures coordinates are from (0, 0) to (1, 1)

Communicating with GLSL

Graphics state is available as uniform variables
```
uniform mat4 gl_ModelViewMatrix;
```
Can extend state
```
uniform float x;
addr = GetUniformLocation( program, "x");
glUniform1f( addr, value );
```

Primitive attributes are available as attribute variables
Can extend attributes (inside glBegin/glEnd)
```
uniform float y;
addr = GetAttributeLocation( program, "y");
glVertexAttrib1f( addr, value );
```
The OpenGL Pipeline in GLSL - Vertex

Built-in attributes

<table>
<thead>
<tr>
<th>Type</th>
<th>Attribute</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec4</td>
<td>gl_Vertex</td>
<td>glVertex*()</td>
</tr>
<tr>
<td>vec4</td>
<td>gl_Color</td>
<td>glColor*()</td>
</tr>
<tr>
<td>vec4</td>
<td>gl_SecondaryColor</td>
<td>glSecondaryColor*()</td>
</tr>
<tr>
<td>vec4</td>
<td>gl_Normal</td>
<td>glNormal()</td>
</tr>
<tr>
<td>vec4</td>
<td>gl_MultiTexCoord0</td>
<td>glMultiTexCoord(0, ...)</td>
</tr>
</tbody>
</table>

The OpenGL Pipeline in GLSL - Fragment

Built-in varying

<table>
<thead>
<tr>
<th>Type</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec4</td>
<td>gl_Position, gl_FrontColor, gl_BackColor</td>
</tr>
<tr>
<td>vec4</td>
<td>gl_FrontSecondaryColor, gl_BackSecondaryColor</td>
</tr>
<tr>
<td>vec4</td>
<td>gl_TexCoord[n]</td>
</tr>
<tr>
<td>vec4</td>
<td>gl_FragCoord</td>
</tr>
</tbody>
</table>

Outputs

<table>
<thead>
<tr>
<th>Type</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec4</td>
<td>gl_FragColor</td>
</tr>
<tr>
<td>vec4</td>
<td>gl_FragDepth</td>
</tr>
</tbody>
</table>
Simple Pixel Shader

```glsl
varying vec2 TexCoord0;
varying vec2 TexCoord1;
uniform sampler2D SomeTexture0;
uniform sampler2D SomeTexture1;
void main()
{
    gl_FragColor =
        texture2D(SomeTexture0, TexCoord0) * 0.5 +
        texture2D(SomeTexture1, TexCoord1) * 0.5;
}
```

This makes it easy to build image processing filters

Limitations

Memory
- No access to neighboring fragments
- Limited stack space, instruction count
- Cannot read and write framebuffer

Performance
- Branching support is limited and slow
- Graphics card will timeout if code takes too long
- Variable support across different graphics cards
GPU Computing

Ideal GPU Problems

GPU’s are great if the problem:

- Executes the same code many times on different input
- Needs lots of math
- Does not share data between executing components
- Has lots of work to do without CPU intervention
Computation on GPU’s

Beyond basic graphics pipeline
- Collision detection
- Fluid and cloth simulation
- Physics
- Ray-tracing

Beyond graphics
- Protein folding (Folding@Home)
- Speech recognition
- Partial differential equation solvers
- Fourier transforms