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);
gBegin(GL_TRIANGLE_STRIP);
gColor3f(0.0, 0.5, 0.0);
gVertex3f(0.0, 0.0, 0.0);
gColor3f(0.5, 0.0, 0.0);
gVertex3f(1.0, 0.0, 0.0);
gColor3f(0.0, 0.5, 0.0);
gVertex3f(0.0, 1.0, 0.0);
gMultMatrix(T);
gLoadIdentity();
gColor3f(0.5, 0.0, 0.0);
gVertex3f(1.0, 1.0, 0.0);
```

...
**Vertex (per-vertex)**

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

![Diagram showing object-space triangles and screen-space lit triangles]

**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)

Screen-space triangles  \rightarrow  Fragments

Texture

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

Fragments  \rightarrow  Texture Fragments
**Fragment**

Combine texture sampler outputs  
Per-fragment shading  
Special effects

---

**Framebuffer Operations**

Owner, scissor, depth, alpha and stencil tests  
Blending or compositing  
Dithering and logical operations

---

---
Display

Gamma correction
Analog to digital conversion

Framebuffer Pixels  →  Light

Programming Stages
Programmable Graphics Pipeline

Shader Program Architecture
What’s in a GPU?

Shader Core  Shader Core  Tex  Primitive Assembly
Shader Core  Shader Core  Tex  Rasterizer
Shader Core  Shader Core  Tex  Framebuffer Ops
Shader Core  Shader Core  Tex  Work Distributor

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

\begin{verbatim}
attribute vec4 gl_Color;
varying vec4 gl_FrontColor;
varying vec4 gl_BackColor;

void main() {
    gl_FrontColor = gl_Color;
}
\end{verbatim}

N. B. All \texttt{glVertex*()} calls result in a \texttt{vec4}

Varying Variables

Varying variables are the outputs of the vertex shader

\begin{verbatim}
attribute vec4 gl_Color;
varying vec4 gl_FrontColor;
varying vec4 gl_BackColor;

void main() {
    gl_FrontColor = gl_Color;
}
\end{verbatim}
Varying Variables

The varying variables are interpolated across the triangle

\( \text{gl\_Color} \) is set to \( \text{gl\_FrontColor} \) or \( \text{gl\_BackColor} \) 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

\[ \text{vec3} \ V3 = \text{vec3}(1.0, 2.0, 3.0); \]
\[ \text{vec4} \ V4 = \text{vec4}(V3, 4.0); \]

Swizzling

\[ \text{vec2} \ V2 = V4.\text{xy}; \]
\[ \text{vec4} \ V4\text{Reverse} = V4.\text{wzyx}; \]
\[ \text{vec4} \ \text{Result} = V4.\text{xyzw} + V4.\text{xxxx}; \]

Basic Vector Operators

\[ \text{float} \ \text{Result} = \text{dot}(V4, V4\text{Reverse}); \]
\[ \text{vec3} \ \text{Result} = \text{cross}(V3, \text{vec3}(1.0, 0.0, 0.0)); \]

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**

- \texttt{vec4 gl\_Vertex} \hspace{1cm} \texttt{glVertex*()}
- \texttt{vec4 gl\_Color} \hspace{1cm} \texttt{glColor*()}
- \texttt{vec4 gl\_SecondaryColor} \hspace{1cm} \texttt{glSecondaryColor*()}
- \texttt{vec4 gl\_Normal} \hspace{1cm} \texttt{glNormal()}
- \texttt{vec4 gl\_MultiTexCoord0} \hspace{1cm} \texttt{glMultiTexCoord(0, \ldots)}

The OpenGL Pipeline in GLSL - Fragment

**Built-in varying**

- \texttt{vec4 gl\_Position}
- \texttt{vec4 gl\_FrontColor, gl\_BackColor}
- \texttt{vec4 gl\_FrontSecondaryColor, gl\_BackSecondaryColor}
- \texttt{vec4 gl\_TexCoord[n]}
- \texttt{vec4 gl\_FragCoord}

**Outputs**

- \texttt{vec4 gl\_FragColor}
- \texttt{vec4 gl\_FragDepth}
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 bind output framebuffer (render target) as an input texture

**Performance**
- Branching support is limited and slow
- Graphics card will timeout if code takes too long
- Variable support across different graphics cards
General-Purpose GPU (GPGPU)

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
- Folding@Home
- Speech recognition
- Partial differential equation solvers
- Fourier transform