Real-Time Graphics Architecture

Lecture 2
The Graphics Pipeline

Kurt Akeley
Pat Hanrahan
http://graphics.stanford.edu/cs448-07-spring/

This lecture

Overview today
Details in subsequent lectures
Today’s topics
- Defining graphics architectures
- The graphics pipeline
- Computation and bandwidth requirements
Declarative vs. imperative

Declarative (what, not how)
- Descriptive: specify input and desired result
  - E.g., `OmitHiddenSurfaces();`
- Example systems:
  - RenderMan scene description
  - Inventor and Performer scene graphs

Imperative (how, not what)
- Procedural: specify actions to create the result
  - E.g., `EnableDepthBuffer();`
- Example systems
  - PostScript and Xlib
  - OpenGL and Direct3D

Graphics system stack

Declarative

<table>
<thead>
<tr>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenegraph API</td>
</tr>
<tr>
<td>Graphics API</td>
</tr>
<tr>
<td>GPU</td>
</tr>
</tbody>
</table>

Imperative
A procedural API defines an architecture

We mean architecture in the Brooks/Blaauw sense:
- Separate from and above implementation

Requires a thorough specification
- Good examples: OpenGL, Direct3D 10
- Not so good examples: early Direct3D versions

(unsual) can separate API interface and architecture
- Direct3D 10 (interface)
- WGF 2.0 (architecture)

OpenGL drawing commands

```cpp
#include <GL/glew.h>

int main()
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    // Set up projection matrix
    // ... (code not shown)

    // Begin drawing
    glBegin(GL_POLYGON);
    glColor3f(1.0f, 0.0f, 0.0f); // Red
    glVertex3f(0.0f, 0.0f, 0.0f);
    glVertex3f(1.0f, 0.0f, 0.0f);
    glVertex3f(0.0f, 1.0f, 0.0f);
    glEnd();

    // Set up lighting
    // ... (code not shown)

    // Draw a yellow triangle
    glBegin(GL_POLYGON);
    glColor3f(0.0f, 1.0f, 0.0f); // Yellow
    glVertex3f(1.0f, 0.0f, 0.0f);
    glVertex3f(0.0f, 1.0f, 0.0f);
    glVertex3f(0.0f, 0.0f, 1.0f);
    glEnd();
}
```
OpenGL architecture

Unpack Pixels → Pixel Operations → Image Rasterization → Texture Memory → Fragment Operations → Frame Buffer

Unpack Vertexes → Vertex Operations → Point, Line, Polygon Rasterization → Frame Buffer

Image

Geometry

CS448 Lecture 2
Kurt Akeley, Pat Hanrahan, Spring 2007
ISA vs. API

CPU architecture
- Specified as ISA (instruction set architecture)
  - But could be API (e.g., Java machine)
- ISA is extremely low level
  - Constraining for implementors
  - Powerful for programmers

GPU architecture
- Specified as an API (but could be ISA ... shaders)
- API is somewhat higher level
  - Less constraining for implementors
  - Less powerful for programmers (but still imperative!)

ISA vs. API

Can think of graphics API as VLIW-like instructions:
- Specify operations
  - Multiple functional units
  - Orthogonal operations
- Specify data paths
  - Composition of operations

Popular for 3rd generation GPUs
Useful for understanding equivariance
OpenGL (ISA-based) equivariance example

Let

\( R = \) render a scene, and
\( T = \) translate by an integral number of pixels

Then

\[ R(T(\text{primitive})) = T(R(\text{primitive})) \]

OpenGL is not a pixel-exact specification

- Equivariance guarantees allow powerful usage
- With less implementation constraint

EQUIVariance is not the same as INVariance

- OpenGL specifications missed this distinction
A trip down the graphics pipeline

Application

Simulation
Input event handlers
Modify data structures
Database traversal
Primitive generation
Utility functions
Command

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

- `glLoadIdentity();`
- `glMultMatrix(T);`
- `glBegin(GL_TRIANGLE_STRIP);`
- `gColor3f(0.0, 0.5, 0.0);`
- `glVertex3f(0.0, 0.0, 0.0);`
- `gColor3f(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);`
- `gColor3f(0.5, 0.0, 0.0);`
- `glVertex3f(1.0, 1.0, 0.0);`
- `glEnd();`

Geometry

Evaluation of polynomials for curved surfaces
Transform and projection
Geometry

Evaluation of polynomials for curved surfaces
Transform and projection (object $\rightarrow$ image space)
Clipping, culling and primitive assembly
Lighting (light sources and surface reflection)
Texture coordinate generation

Object-space triangles
Screen-space lit triangles

Rasterization

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

Screen-space triangles
Fragments
Texture

- Texture transformation and projection
- Texture address calculation
- Texture filtering

Fragment

- Texture combinators
**Fragment**

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

---

**Display**

Gamma correction  
Analog to digital conversion
Canonical graphics pipeline

Application
Command
Geometry
Rasterization
Texture
Fragment
Display

Object

Image

Canonical

Application
Command
Geometry
Rasterization
Texture
Fragment
Display

OpenGL 1.2

Application
Unpack
Vertex
Rasterization
Texture
Fragment
Display

CS448 Lecture 2
Kurt Akeley, Pat Hanrahan, Spring 2007
CS448 Lecture 2

Kurt Akeley, Pat Hanrahan, Spring 2007

Canonical

Application
Command
Geometry
Rasterization
Texture
Fragment
Display

OpenGL 1.2

Application
Unpack
Vertex
Rasterization
Texture
Fragment
Display

CS448 Lecture 2

Kurt Akeley, Pat Hanrahan, Spring 2007

OpenGL architecture

CS448 Lecture 2

Kurt Akeley, Pat Hanrahan, Spring 2007
### REYES

**Canonical**
- Application
- Command
- Geometry
- Rasterization
- Texture
- Fragment
- Display

**REYES**
- Application
- Command
- Tessellation
- Shade/Texture
- Sample
- Fragment
- Display

---

### Direct3D 10

**Canonical**
- Application
- Command
- Geometry
- Rasterization
- Texture
- Fragment
- Display

**Direct3D 10**
- Application
- Input Assembler
- Vertex Shader
- Geometry Shader
- Rasterize Stage
- Pixel Shader
- Output Merger
- Display

---

CS448 Lecture 2
Kurt Akeley, Pat Hanrahan, Spring 2007
Sequence differences

Semantic
- New operations, e.g.,
  - Full-scene antialiasing
- Different pipelines, e.g.,
  - REYES shading and texture mapping in object space
  - Direct3D 10 tessellation after vertex shading

Implementation (no semantic difference)
- Early Z-cull (as in Direct3D 10)
- Tiled rendering
- Abstraction
  - abstraction distance
Graphics state (aka context)

*Required to minimize data transmission*

Resources (shared or global, persistent)
- Fonts
- Texture
- Display lists

Attributes
- Appearance: Lights, Materials, Colors, ...
- Transformation: camera, model, texture, ...
- Options: fb formats, constant per-frame

Graphics State

Ideally small and bounded
- e.g., maximum number of lights
- OpenGL: ~12kb

Distributed throughout the pipeline
- Difficult to manage (forces major design decisions)
- Must often be broadcast; must be consistent

Difficult to implement context switching
- OpenGL has a single context; X has multiple contexts
- One active context → windows are difficult

Expensive to query state
The Wheel of Reincarnation

General-purpose processor + display processor

Display processor evolution:

1. Refresh display from vector- or frame-buffer
2. Augment with drawing processor
3. Augment with structured display list processor
4. General drawing processor + display processor

Myers and Sutherland: Break cycle and KISS

J. Clark: Terminals should be workstations and workstations should support immediate-mode graphics efficiently

Computational Requirements
**Functionality vs. frequency**

Geometry processing = per-vertex
- Transformation and Lighting (T & L)
- Floating point; complex operations
- 100s of millions of vertices

Fragment processing = per-fragment
- Blending and texture combination
- Fixed point; limited operations
- 10s of billions of fragments

---

**Geometry computations**

Application
- Command
- Geometry
  - Rasterization
  - Texture
  - Fragment
- Display

Geometry (per-vertex)
- Assumptions:
  - 1 infinite light
  - Texture coordinates

<table>
<thead>
<tr>
<th></th>
<th>ADD</th>
<th>CMP</th>
<th>MUL</th>
<th>DIV</th>
<th>SPE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>40</td>
<td>8</td>
<td>53</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Rough estimate: 100 ops
Rasterization computations (vertex)

Rasterization: per-vertex
Assumptions:
- 7 interpolants (z,r,g,b,s,t,q)

<table>
<thead>
<tr>
<th></th>
<th>ADD</th>
<th>CMP</th>
<th>MUL</th>
<th>DIV</th>
<th>SPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>62</td>
<td>22</td>
<td>55</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Rough estimate: 150 ops

Rasterization computations (fragment)

Rasterization: per-fragment
Assumptions:
- 7 interpolants (z,r,g,b,s,t,q)

<table>
<thead>
<tr>
<th></th>
<th>ADD</th>
<th>CMP</th>
<th>MUL</th>
<th>DIV</th>
<th>SPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>16</td>
<td>3</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Rough estimate: 25 ops
Texture computations

Application: per-fragment
Assumptions:
- Projective texture mapping
- Level of detail calculation
- Trilinear interpolation

ADD | CMP | MUL | DIV | SPE
---|------|-----|-----|-----
42  | 5    | 48  | 1   | 3

Rough estimate: 100 ops

Fragment computations

Application: per-fragment
Assumptions:
- Texture blending
- Color blending
- Depth buffering

ADD | CMP | MUL | DIV | SPE
---|------|-----|-----|-----
8   | 1    | 16  | 0   | 0

Rough estimate: 25 ops
Total computations (rough estimate)

- **Application**
  - Per-vertex: ~250 ops
    - ADD: 102
    - CMP: 30
    - MUL: 108
    - DIV: 5
    - SPE: 1
  - Per-fragment: ~150 ops
    - ADD: 66
    - CMP: 9
    - MUL: 70
    - DIV: 1
    - SPE: 3

Bandwidths (rough estimate)

- **Application**
  - 0.880 GB/s
- **Command**
  - 20 Mvert/s
- **Geometry**
  - 1000 Mpix/s
- **Rasterization**
  - Texture memory
- **Texture**
  - 4 GB/s
- **Fragment**
  - 150 Gops
- **Display**
  - 120 Mpix/s
  - Frame buffer
  - 16 GB/s
  - 0.36 GB/s
Comparison to current GPU

<table>
<thead>
<tr>
<th></th>
<th>Gops/sec</th>
<th>Gbytes/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our 2001 estimation</td>
<td>155</td>
<td>21</td>
</tr>
<tr>
<td>GeForce 7900 GTX</td>
<td>2100</td>
<td>51</td>
</tr>
</tbody>
</table>

Summary

An imperative API defines an architecture

- Much as an ISA defines one

An imperative API defines an abstract machine

- Allows variations in implementation
- But large “abstraction distances” are problematic

The canonical OpenGL-like pipeline persists

- Many features added over the years
- OpenGL has slowed the Wheel of Reincarnation

We have some sense of GPU speeds and feeds
Real-Time Graphics Architecture

Lecture 2
The Graphics Pipeline

Kurt Akeley
Pat Hanrahan
http://graphics.stanford.edu/cs448-07-spring/