Graphics Hardware

MIT EECS 6.837
Jonathan Ragan-Kelley

with slides from
Frédo Durand, Hanrahan & Akeley, Gary McTaggart, NVIDIA, ATI
Input:

- Geometric model:
  Description of all object, surface, and light source geometry and transformations
- Lighting model:
  Computational description of object and light properties, interaction (reflection)
- Synthetic Viewpoint (or Camera):
  Eye position and viewing frustum
- Raster Viewport:
  Pixel grid onto which image plane is mapped

Output:

- Colors/Intensities suitable for framebuffer display
  (For example, 24-bit RGB value at each pixel)
Modern Graphics Hardware ("GPU")

Hardware implementation of the rendering pipeline

Programmability ("shaders") within specific stages of the pipeline

(Recent, last 9 years)

At the vertex and pixel level

// bowling pin, based on RenderMan example
(CIRCLE over (BRUNS over BASE)) * MARKS * Cd + Cs
First Generation - Wireframe

**Vertex:** transform, clip, and project

**Rasterization:** color interpolation (points, lines)

**Fragment:** overwrite

**Dates:** prior to 1987
Second Generation - Shaded Solids

**Vertex:** lighting calculation

**Rasterization:** depth interpolation (triangles)

**Fragment:** depth buffer, color blending

**Dates:** 1987-1992
Third Generation - Texture Mapping

Vertex: texture coordinate transformation
Rasterization: texture coordinate interpolation
Fragment: texture evaluation, antialiasing
Fourth Generation - Programmable Shading

Vertex: programmable transformation
Rasterization: user attribute interpolation
Fragment: programmable color computation
Dates: 2000-2009
Fifth Generation - Programmable Pipeline

Vertex, Rasterization, Fragment, ???:
  arbitrary parallel programs

Dates: 2009+
Questions?
Questions?
The Graphics Pipeline

- Geometry
- Rasterization
- Texture/shading
- Fragment
- Display
Programmable Graphics Hardware

Geometry and Pixel (fragment) stage become programmable

- Elaborate appearance
- More and more general-purpose computation (GPGPU)
The Direct3D 9 Graphics Pipeline

- **Vertex Buffer**
- **Index Buffer**
- **Texture**
- **Texture**
- **Frame Buffer (color)**
- **Z Buffer (depth)**

**Graphics Memory**

- **Input Assembler**
- **Vertex Shader**
- **Primitive Assembler**
- **Setup/Rasterizer**
- **Pixel Shader**
- **Output Merger**

- **5%**
- **10%**
- **5%**
- **10%**
- **50%**
- **20%**
Questions?
**Vertex Shaders**

\[ f(\text{position, attributes}) \rightarrow \text{new position, attributes} \]

*purely functional (no side-effects)*

Vertex shaders can be used to move/animate vertices.

Figure from NVIDIA
Vertex Shader: Blendshapes

50 face geometries
- angry, happy, sad, move eyebrow, ...

Each target stored as difference vector
- For each vertex:
  average position + 50 differences

Result is a weighted sum of all targets
- Only transmit new weights each frame, the targets remain in GPU memory
- Big multiply-add
  • Per active blend target
  • Per attribute
Vertex Shader: Skinning (SSD)

1. Transform each vertex \( p_i \) with each bone as if it was rigidly tied to it
2. Blend the results using bone weights

```plaintext
float4 skin(float4 restPos,
            uniform float4x4 xform[N_BONES],
            uniform float weight[N_BONES])
{
    float4 outPos = float4(0,0,0,0)
    for (int b = 0; b < N_BONES; b++) {
        outPos += weight[b]*mul(xform[b], restPos)
    }
    return outPos
}
```
Other jobs for vertex shaders

Prepare data for pixel shaders
(e.g. texture coordinates)
- Stored/computed at vertices
- Linearly interpolated per pixel

Modern graphics hardware provides tons of interpolants (32x4 floats)

Project vertices to the screen:

```cpp
float4 transform(float4 worldPos,
                 uniform float4x4 modelViewProjection) {
    return mul(modelViewProjection, worldPos)
}
```
Pixel Shaders

\[ f(\text{interpolants}) \rightarrow \text{color, [depth]} \]

 purely functional (no side-effects)

*Each pixel is calculated individually*

*Pixel shaders have little or no knowledge of neighboring pixels*

Figure from NVIDIA
Pixel Shader: Blinn-Torrance Phong

Uses the “halfway vector” $h$ between $l$ and $v$.

$$L_o = k_s \cos(\beta) q \frac{L_i}{r^2}$$

$$= k_s (n \cdot h)^q \frac{L_i}{r^2}$$

$$h = \frac{l + v}{||l + v||}$$
Pixel Shader: Blinn-Torrance Phong

```cpp
struct interpolants { float4 p, n, v }
struct light { float4 pos, float Li }

float4 phong(interpolants in, uniform light lgt,
              uniform float q, uniform float Ks)
{
    float4 l = lgt.pos - in.p
    float r2 = length(l)*length(l)
    float4 h = normalize(l+in.v)  // useful built-ins
    return Ks * pow(dot(in.n, h), q) * (lgt.Li / r2)
}
```

TODO: simple texture mapping?
Brushed Metal

Procedural texture
Anisotropic lighting
Melting Ice

- Procedural, animating texture
- Bumped environment map
Toon & Fur

Toon shading

Volumetric fur
Vegetation & Thin Film

Translucence
Backlighting

Custom lighting
Simulates iridescence
Allows for amazing quality
Earth, $4.5 \times 10^9$ B.C.E.  

Crysis, 2007
Rich scene appearance

• Vertex shader
  – Geometry (skinning, displacement)
  – Setup interpolants for pixel shaders

• Pixel shader
  – Visual appearance
  – Also used for image processing, GPGPU abuses

• Multi-pass
  – Render the scene or part of the geometry multiple times
  – Different viewpoints (e.g. shadow map, shadow volume)
  – More complex shading (e.g. pass per-light)
  – Image processing (e.g. HDR tone mapping, bloom)
Questions?
How to program shaders?

Assembly code, or
Higher-level language and compiler (e.g. HLSL, GLSL, Cg)

Write separate vertex and fragment programs

Send to the GPU via graphics API (Direct3D, OpenGL) (much like a texture or other piece of state)

(Heavily) optimized at runtime by the driver’s JIT compiler
General-purpose computation on GPUs

Dense data-parallel processor (TeraFLOPS)
Increasingly programmable (Code executed for each vertex or each pixel)
C & C-like languages

Use “shaders” for general-purpose computation

Peak performance difficult to reach
Parallelism = hard

Navier-Stokes on GPU [Bolz et al.]
Questions?
Graphics Hardware

High performance through:

- **Parallelism**
- **Specialization**
- Limited synchronization (minimal data dependency)
- Efficient latency hiding (pre-fetching, threading)

![Diagram showing parallel and data parallelism]
Modern Graphics Hardware

100s of parallel shader pipelines (TFLOPS)
Deep pipeline (~1k stages)
Hierarchical tiling of screen (~4x4)
  – Early Z rejection if entire tile is occluded
Pixels processed as quads (2x2 pixel neighborhoods)
  – Allows derivatives in pixel shader (for filtering)
Massive multithreading to hide texture latency
  – And smart memory layout
  – Throughput (bandwidth)-optimized memory controllers
    (at the expense of latency)
Why is graphics hardware so fast?

<table>
<thead>
<tr>
<th>GPU: throughput</th>
<th>CPU: latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Massive, explicit <strong>parallelism</strong></td>
<td>Predominantly <strong>scalar</strong></td>
</tr>
<tr>
<td>More transistors do <strong>compute</strong>, less cache, bookkeeping</td>
<td>Most transistors are <strong>bookkeeping, cache</strong></td>
</tr>
<tr>
<td><strong>Specialization</strong> (rasterizer, texture filtering)</td>
<td>Highly <strong>general-purpose</strong> (x86 ISA)</td>
</tr>
<tr>
<td><strong>High</strong> arithmetic intensity (math ops » memory ops)</td>
<td><strong>Low</strong> arithmetic intensity (math ops ≈ memory ops)</td>
</tr>
<tr>
<td>Deep pipeline, latency hiding, prefetching</td>
<td>Short pipeline, latency intolerance, out-of-order exec</td>
</tr>
<tr>
<td>Little data dependence</td>
<td>Arbitrary data dependence</td>
</tr>
<tr>
<td>Controlled memory access patterns</td>
<td>Arbitrary, dynamic memory access patterns</td>
</tr>
</tbody>
</table>
Questions?
Traditional hardware architecture

vertex units

one big parallel rasterizer

mipmap filtering

fragment units

raster operation units

z-buffer, framebuffer

screen-locked
Unified shading architecture (NVIDIA G80)

128 SPs
3 ALUs/SP
1.5 ghz
= 576 GFLOPs

85 GB/sec

~4k+ threads
Questions?
Bottlenecks?

- The bottleneck determines overall throughput.
- In general, the bottleneck varies over the course of an application and even over a frame.
- Getting good performance is all about finding and eliminating bottlenecks in the pipeline.

Slide from NVIDIA
Potential Bottlenecks

- PCIe transfer limited
- CPU limited
- Video Memory
  - Geometry
  - Commands
  - Textures
  - Frame Buffer
- On-Chip Cache Memory
  - Pre-TnL cache
  - Post-TnL cache
  - Texture cache
- System Memory
- CPU limited
- Frame buffer b/w limited
- vertex transform limited
- setup limited
- raster limited
- fragment shader limited

- Triangle Setup
- Rasterization
- Fragment Shading and Raster Operations

- CPU limited
- PCle transfer limited

- Textures
  - b/w limited
The term “transform/vertex/geometry bound” often means the bottleneck is “anywhere before the rasterizer”.

The term “fill/raster bound” often means the bottleneck is “anywhere after setup for rasterization” (computation of edge equations).

Can be both transform and fill bound over the course of a single frame!
Questions?
Shader zoo
Layering

// bowling pin, based on RenderMan example
(CIRCLE over (BRUNS over BASE)) * MARKS * Cd + Cs
Radiosity
Normal Mapped Radiosity
Albedo
Albedo * Normal Mapped Radiosity
Radiosity Normal Mapping Shade Tree

- Normal
- Lightmaps
- Albedo

Diagram shows the flow of information between Normal, Lightmaps, Albedo, and the final shade tree.
Cube Map Specular
Normal Mapped Specular
Specular Factor
Normal Mapped Specular
Normal Mapped Specular * Specular Factor
Final Result
Normal Map
Albedo * Normal Mapped Radiosity
Normal Mapped Specular
Specular Factor
Normal Mapped Specular * Specular Factor
Albedo * Normal Mapped Radiosity
Final Result
Refraction mapping (multipass)

Slide by Gary McTaggart (Valve)
Image processing

- Start with ordinary model
  - Render to backbuffer
- Render parts that are the sources of glow
  - Render to offscreen texture
- Blur the texture
- Add blur to the scene
More glow

From “Tron”
To learn more…

- [http://www.graphics.stanford.edu/courses/cs448a-01-fall/](http://www.graphics.stanford.edu/courses/cs448a-01-fall/)
- [https://graphics.stanford.edu/wikis/cs448-07-spring](https://graphics.stanford.edu/wikis/cs448-07-spring)

- [http://s09.idav.ucdavis.edu](http://s09.idav.ucdavis.edu)
- [http://s08.idav.ucdavis.edu](http://s08.idav.ucdavis.edu)

- [http://www.beyond3d.com](http://www.beyond3d.com)


  - [http://download.nvidia.com/developer/SDK/Individual_Samples/effects.html](http://download.nvidia.com/developer/SDK/Individual_Samples/effects.html)