The Graphics Pipeline
Ray Tracing: Why Slow?

Basic ray tracing: 1 ray/pixel
Ray Tracing: Why Slow?

Basic ray tracing: 1 ray/pixel

But you really want shadows, reflections, global illumination, antialiasing...

- 100-1000 rays/pixel
Rendering: Rasterization

Tessellate objects into primitives
Rendering: Rasterization

Tessellate objects into primitives
Draw each separately:
• determine position and color
• draw pixels to screen
Rendering: Rasterization

Tessellate objects into primitives
Draw each separately:
• determine position and color
• draw pixels to screen

Embarrassingly parallel
Fast
Rendering: Rasterization

How to deal with overlaps?
Rendering: Rasterization

How to deal with overlaps?

Keep track of **depth** of previously-drawn pixels

Depth image or depth buffer
Rendering: Rasterization

How to deal with overlaps?
• depth buffer

How to deal with shadows/reflections?
Rendering: Rasterization

How to deal with overlaps?
• depth buffer

How to deal with shadows/reflections?
• hmm...
Ray Tracing vs Rasterization

Ray Tracing

Loop over pixels
Light effects “easy”
shadows, reflections, caustics, …
Slow-ish
Used in movies

Rasterization

Loop over triangles
Light effects require hacks and tricks
Blazingly fast
Used in games
Ray Tracing vs Rasterization
Rasterization Algorithms

Actually rasterizing objects not so easy…
Rasterization Algorithms

Actually rasterizing objects not so easy…

…so use specialized hardware to do it
Vertex List
Triangle List
Primitive Assembly
Fragment Shader
Rasterization
Framebuffer
Screen

Vertex Shader
Tessellation/Geometry Shaders
Textures
GPU
CPU

Tessellation/Geometry Shaders
Fragment Shader
Rasterization
Framebuffer
Screen

CPU
GPU
Vertices and Triangles

Vertex List

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>0.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>1.0</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Triangle List

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
Sending Data to the GPU

One vertex/triangle at a time: very slow

Vertex Buffer Objects: big arrays of data
- vertex positions
- vertex colors
- texture info
- etc
Shaders

Small arbitrary programs that run on GPU
Massively parallel
Shaders

Small arbitrary programs that run on GPU
Massively parallel

Four kinds: vertex, tessellation, geometry, fragment
Shaders

Small arbitrary programs that run on GPU
Massively parallel

Four kinds: vertex, tessellation, geometry, fragment

These days: used for many non-rendering applications (GPGPU)
**Vertex Shader**

Runs in parallel on every vertex

- no access to triangles or other verts
Vertex Shader

Runs in parallel on every vertex
• no access to triangles or other verts

Main job: transform vertex positions
**Vertex Shader**

Runs in parallel on every vertex
- no access to triangles or other verts

Main job: transform vertex positions

Also used for shading
Processing Primitives

Assembly: group verts into polygons
Processing Primitives

Assembly: group verts into polygons

Tessellation shader: runs on each triangle
- can split triangles into subtriangles
- increase level of detail near camera, etc
Processing Primitives

Assembly: group verts into polygons

Tessellation shader: runs on each triangle
- can split triangles into subtriangles
  - increase level of detail near camera, etc

Geometry shader: runs on each triangle
- can access verts and neighbors
- more general than tessellation, slower
Fragment Shader

Runs in parallel on each fragment (pixel)
• rasterization: one tri -> many fragments

Writes color and depth for one pixel (only)

Final texturing/coloring of the pixels
Fragment Shader

Many fragments per triangle…
Fragment Shader

Many fragments per triangle…

GPU automatically applies barycentric interpolation

UV coords, normals, colors, …
Normalized Device Coordinates

Before rasterization, must decide what geometry to show and where
Normalized Device Coordinates

Before rasterization, must decide what geometry to show and where

GPU draws everything in unit cube
Normalized Device Coordinates

Before rasterization, must decide what geometry to show and where

GPU draws everything in unit cube

Everything clipped
Normalized Device Coordinates

X & Y axes map to screen width & height
Normalized Device Coordinates

X & Y axes map to screen width & height
Z used for depth
Normalized Device Coordinates

Notice: deeper points have **higher** $z$

(not right-handed)
Camera Coordinates

Notice: look down \textbf{negative} z direction
Camera Coordinates

Notice: look down negative z direction

Projection: transform from camera to NDC
(typically in vertex shader)
Coordinate Systems in Graphics

object

model matrix $M$

world

view matrix $V$

camera

perspective matrix $P$

normalized device
For Extra Confusion

Screen coordinates

$$(0, 0)$$

$$(W, H)$$
Framebuffer

Memory region containing pixel data

The old days: mapped to RAM with DMA
• CPU could write to it directly
Framebuffer

Memory region containing pixel data

The old days: mapped to RAM with DMA
- CPU could write to it directly

Now: GPU controls it
Framebuffer

Several layers:

- Color buffer: RGB of each pixel
Framebuffer

Several layers:

- Color buffer: RGB of each pixel
- Depth buffer
Framebuffer

Several layers:

- Color buffer: RGB of each pixel
- Depth buffer
- Stencil buffer, etc
Framebuffer

Several layers:
- Color buffer: RGB of each pixel
- Depth buffer
- Stencil buffer, etc

Can be saved to file, to texture, to screen
Displaying the Framebuffer

CRTs: beam sweeps across screen drawing pixels
• one pass: 1/60 secs
Displaying the Framebuffer

CRTs: beam sweeps across screen drawing pixels
• one pass: 1/60 secs

LCDs: grabs framebuffer every 1/60 secs
Flickering and Tearing

Framebuffer changes while monitor draws
Double-Buffering to Stop Tearing

Use two framebuffers

Render to **back buffer** while showing **front buffer**

Then swap
Double-Buffering to Stop Tearing

On CRTs: must wait for vertical retrace to swap
Double-Buffering to Stop Tearing

On CRTs: must wait for **vertical retrace** to swap
- “vsync”
- occurs 1/60 sec
Double-Buffering to Stop Tearing

On CRTs: must wait for **vertical retrace** to swap
- “v sync”
- occurs 1/60 sec

On LCDs: swap when not reading
Communicating with GPU

Very low level / awkward
Communicating with GPU

Very low level / awkward

Two types of data:

- vertex attributes in VBOs
- global variables ("uniforms")
Communicating with GPU

Very low level / awkward
Two types of data:
• vertex attributes in VBOs
• global variables ("uniforms")

GPU stores no variable names – just location numbers
Communicating with GPU

Very low level / awkward
Two types of data:
• vertex attributes in VBOs
• global variables ("uniforms")
GPU stores no variable names – just location numbers
GPU programming is lots of "plumbing"
• binding inputs and outputs correctly
Shaders

Inputs
position
normal

Uniforms
view
lightPos

GPU

Vertex Attributes
1
2
3

Global Memory
1
2
3

CPU

VBOs
vertPos[]
vertNormals[]

Uniforms
view
lightPos
Shaders

Inputs
- position
- normal

Uniforms
- view
- lightPos

GPU

Vertex Attributes
- 1
- 2
- 3

Global Memory
- 1
- 2
- 3

VBOs
- vertPos[]
- vertNormals[]

Uniforms
- view
- lightPos

when shader is compiled
Shaders

Inputs
- position
- normal

Uniforms
- view
- lightPos

GPU

Vertex Attributes
- vertPos[]
- vertNormals[]

Global Memory
- 1
- 2
- 3

VBOs

Uniforms
- view
- lightPos

CPU

glBindAttribLocation()
**Shaders**

- Inputs:
  - position
  - normal

- Uniforms:
  - view
  - lightPos

**GPU**

- Vertex Attributes:
  - vertPos[]
  - vertNormals[]

- Global Memory:
  - 1
  - 2
  - 3

**CPU**

- VBOs:
  - vertPos[]
  - vertNormals[]

- Uniforms:
  - view (2)
  - lightPos (1)

```c
glfwGetUniformLocation()
```
CPU

VBOs
vertPos[]
vertNormals[]

Uniforms
view (2)
lightPos (1)

at render time:
glVertexAttribPointer()
Shaders

Inputs
position
normal

Uniforms
view
lightPos

GPU

Vertex Attributes
1
2
3

Global Memory
1
2
3

VBOs
vertPos[]
vertNormals[]

Uniforms
view (2)
lightPos (1)

at render time:
glVertexAttribPointer()  
glUniform**()
Shaders

- Inputs
  - position
  - normal

Uniforms
- view
- lightPos

Global Memory

VAOs store the VBO state

GPU

- Vertex Attributes
  - 1
  - 2
  - 3

- vertPos[]
- vertNormals[]

CPU

- VBOs
- Uniforms
  - view (2)
  - lightPos (1)
Ray Tracing: Why Slow? Reprise

Basic ray tracing: 1 ray/pixel

But you really want shadows, reflections, global illumination, antialiasing…
• 100-1000 rays/pixel

Much less hardware support
• inhomogeneous / unpredictable work