Lecture 15:
The Real-Time 3D Graphics Pipeline Architecture

Visual Computing Systems
Stanford CS348V, Winter 2018
What is an “architecture”?  
(not distinguishing between software or hardware architecture)
A system architecture is an abstraction

- **Entities (state)**
  - Registers, buffers, vectors, triangles, lights, pixels, images

- **Operations (that manipulate state)**
  - Add two registers, copy buffers, multiply vectors, blur images, draw triangles

- **Mechanisms for creating/destroying entities, expressing operations**
  - Execute machine instruction, make API call, express logic in a programming language

Notice the different levels of granularity/abstraction in my examples

Key course theme: choosing the right level of abstraction for system’s needs
Decision impacts system’s expressiveness/scope and potential for efficient implementation
Example: x86 architecture?

- **State:**
  - Maintained by execution context (registers, PC, VM mappings, etc.)
  - Contents of memory

- **Operations:**
  - x86 instructions (privileged and non-privileged)
Example: GPU compute architecture (as defined by CUDA)?

- **State:**
  - Execution context for all executing CUDA threads
  - Contents of global memory

- **Operations:**
  - Bulk launch N CUDA threads running of kernel K: $\text{Launch}(N, k)$
  - Individual instructions executed by CUDA thread
CUDA constructs (the kernel)

```c
// CUDA kernel definition
__global__ void scale(float amount, float* a, float* b)
{
    int i = threadIdx.x; // CUDA builtin: get thread id
    b[i] = amount * a[i];
}

// note: omitting array initialization via cudaMalloc()
float scale_amount;
float* input_array;
float* output_array;

// launch N CUDA threads, each thread executes kernel 'scale'
scale<<1,N>>(scale_amount, input_array, output_array);
```

Question: What should N be?

Question: Do you normally think of “threads” this way?
The 3D rendering task

**Input: description of a scene**
- 3D surface geometry (e.g., triangle meshes)
- surface materials
- lights
- camera

**Output: image**

Problem statement: Determine how each geometric element contributes to the appearance of each output pixel in the image, given a description of a scene’s surface properties and lighting conditions?
Goal: render very high complexity 3D scenes

- 100’s of thousands to millions of triangles in a scene
- Complex material, lighting, and animation computations
- High-resolution screen outputs (2-4 Mpixel + supersampling)
- 30-60 fps
Goal: render very high complexity 3D scenes

Ryse: Son of Rome (image credit: http://www.gamespot.com/ryse-son-of-rome/images/)
The real-time graphics pipeline architecture

(GPU-accelerated OpenGL/D3D graphics pipeline, from a systems perspective)

The graphics pipeline is an architecture for driving modern GPU execution

(Note to CUDA programmers: graphics pipeline was the original interface to GPU hardware. Compute mode execution came later...)
Real-time graphics pipeline entities

Vertices

Primitives
(triangles, points, lines)

Fragments

Pixels
Real-time graphics pipeline operations

Vertices
- Vertex Generation
  - Vertex stream
  - Vertex Processing
    - Vertex stream

Primitives
- Primitive Generation
  - Primitive stream
  - Primitive Processing
    - Primitive stream

Fragments
- Fragment Generation (Rasterization)
  - Fragment stream
  - Fragment Processing
    - Fragment stream

Pixels
- Pixel Operations

* Imprecise definition: will give precise definition in later lecture
Real-time graphics pipeline state

Vertices
- Vertex Generation
  - Vertex stream
- Vertex Processing
  - Vertex stream

Primitives
- Primitive Generation
  - Primitive stream
- Primitive Processing
  - Primitive stream

Fragments
- Fragment Generation (Rasterization)
  - Fragment stream
- Fragment Processing
  - Fragment stream

Pixels
- Pixel Operations
  - 

Memory Buffers (system state)

1. Vertex data buffers
2. Buffers, textures
3. Buffers, textures
4. Buffers, textures

Output image buffer
Issues to keep in mind during this overview*

- Level of abstraction
- Orthogonality of abstractions
- How is the pipeline designed for performance/scalability?
- What the pipeline does and **DOES NOT** do

* These are great questions to ask yourself about *any system* you study
The graphics pipeline

- Vertices
  - Vertex Generation
  - Vertex Processing

- Primitives
  - Primitive Generation
  - Primitive Processing

- Fragments
  - Rasterization (Fragment Generation)
  - Fragment Processing

- Pixels
  - Frame-Buffer Ops

Output image buffer

Memory
Command: draw these triangles!

Inputs:

```
list_of_positions = { v0x, v0y, v0z,  
v1x, v1y, v1x,  
v2x, v2y, v2z,  
v3x, v3y, v3x,  
v4x, v4y, v4z,  
v5x, v5y, v5x  
};

list_of_texcoords = { v0u, v0v,  
v1u, v1v,  
v2u, v2v,  
v3u, v3v,  
v4u, v4v,  
v5u, v5v  
};
```

Object-to-camera-space transform: \( T \)

Perspective projection transform \( P \)

Size of output image \( (W, H) \)

Use depth test /update depth buffer: YES!
"Assembling" vertices

Contiguous version data version

```
my_vtx_buffer
V0  V1  VN-1

glBindBuffer(GL_ARRAY_BUFFER, my_vtx_buffer);
glDrawArrays(GL_TRIANGLES, 0, N);
```

Indexed access version ("gather")

```
my_vtx_buffer
V0  V1  VN-1

my_vtx_indices  1  3  2  1  5  6

glBindBuffer(GL_ARRAY_BUFFER, my_vtx_buffer);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT,
               my_vtx_indices);
```
“Assembling” vertices

Output of vertex generation is a collection of vertex records.

Current architectures set a limit of 128 32-bit attributes per vertex.
What the vertex processing kernel does

Transform triangle vertices from world-space coordinates into camera space coordinates
What the vertex processing kernel does

Apply perspective projection transform to transform triangle vertices into normalized coordinate space
Vertex processing: inputs

Uniform data: constant read-only data provided as input to every instance of the vertex shader e.g., object-to-clip-space vertex transform matrix

Vertex processing operates on a stream of vertex records + read-only “uniform” inputs.
**Vertex processing: inputs and outputs**

```c
struct input_vertex {
    float3 pos;  // object space
};

struct output_vertex {
    float3 pos; // NDC space
};

uniform mat4 my_transform;   // P * T

output_vertex my_vertex_program(input_vertex in) {
    output_vertex out;
    out.pos = my_transform * in.pos; // matrix-vector mult
    return out;
}
```

(* Note: this is pseudocode, not valid GLSL syntax)
Example per-vertex computation: lighting

Input per-vertex data: surface normal, surface color

Input uniform data: light direction, light color
Example per-vertex computation: skeletal animation via “skinning”

\[ V_{\text{skinned}} = \sum_{b \in \text{bones}} w_b M_b V_{\text{base}} \]

Input per-vertex data: base vertex position \((V_{\text{base}})\) + blend coefficients \((w_b)\)

Input: uniform data: “bone” matrices \((M_b)\) for current animation frame

Image credit: http://www.okino.com/conv/skinning.htm
Primitive generation: group vertices into primitives

- Vertices
  - 1 in / 1 out
  - Vertex Generation
  - Vertex Processing
  - Primitives
    - 3 in / 1 out
      - Primitive Generation
      - Primitive Processing
      - Fragments
        - Rasterization (Fragment Generation)
        - Fragment Processing
        - Pixels
        - Frame-Buffer Ops
        - Output image buffer

Memory
- Uniform data
Programmable primitive processing *

**Geometry shader** in OpenGL/Direct3D terminology

**Pipeline caps output at 1024 floats of output**
Primitive processing: clipping

- Discard triangles that lie complete outside the unit cube (culling)
  - They are off screen, don’t bother processing them further
- Clip triangles that extend beyond the unit cube to the cube
  - Note: clipping may create more triangles
Transform to screen coordinates

Transform vertex xy positions from normalized coordinates into screen coordinates (based on screen w,h)
The graphics pipeline

- Vertices: 1 in / 1 out
- Primitives: 3 in / 1 out (for tris)
- Fragments: 1 in / small N out
- Pixels
  - Vertex Generation
  - Vertex Processing
  - Primitive Generation
  - Primitive Processing (Fragment Generation)
  - Rasterization
  - Fragment Processing
  - Frame-Buffer Ops
  - Output image buffer

Memory:
- Uniform data
- Uniform data

Stanford CS348V, Winter 2018
Rasterization (fragment generation)

1 input prim $\rightarrow$ N output fragments

N is unbounded
(size of triangles varies greatly)

struct fragment  // note similarity to output_vertex from before
{
    float x, y;    // screen pixel coordinates (sample point location)
    float z;      // depth of triangle at sample point
    float3 normal; // interpolated application-defined attribs
    float2 texcoord; // (e.g., texture coordinates, surface normal)
};
Rasterization (fragment generation)

Compute covered pixels
Sample vertex attributes once per covered pixel

struct fragment
// note similarity to output_vertex from before
{
  float x, y;  // screen pixel coordinates (sample point location)
  float z;    // depth of triangle at sample point
  float3 normal; // interpolated application-defined attribs
  float2 texcoord; // (e.g., texture coordinates, surface normal)
}

Stanford CS348V, Winter 2018
Implementation of rasterization
Computing triangle coverage

What pixels does the triangle overlap?

Input:
projected position of triangle vertices: \( P_0, P_1, P_2 \)

Output:
set of pixels “covered” by the triangle
What does it mean for a pixel to be covered by a triangle?
Sampling 2D triangle coverage signal

\[
\text{coverage}(x,y) = \begin{cases} 
1 & \text{if the triangle contains point } (x,y) \\
0 & \text{otherwise}
\end{cases}
\]
Edge cases (literally)

Is this sample point covered by triangle 1? or triangle 2? or both?
Edge rules

- Direct3D rules: when edge falls directly on sample, sample classified as within triangle if the edge is a “top edge” or “left edge”
  - Top edge: horizontal edge that is above all other edges
  - Left edge: an edge that is not exactly horizontal and is on the left side of the triangle. (triangle can have one or two left edges)

Source: Direct3D Programming Guide, Microsoft
Results of sampling triangle coverage

Note: I’m not drawing the boundaries of image pixels anymore.
I have a sampled signal, now I want to display it on a screen

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

So if we send the display this...
We might see this when we look at the screen
(assuming a screen pixel emits a square of perfectly uniform intensity of light)
Recall: the real coverage signal was this

Note: I’m not drawing the boundaries of image pixels anymore.
Problem: aliasing

- Undersampling high frequency signal results in aliasing
  - “Jaggies” in a single image
  - “Roping” or “shimmering” in an animation

- High frequencies exist in coverage signal because of triangle edges
Initial coverage sampling rate (1 sample per pixel)

Note: I’m not drawing the boundaries of image pixels anymore.
Increase density of sampling coverage signal
Supersampling

Example: stratified sampling using four samples per pixel
Resample to display’s pixel resolution

(Because a screen displays one sample value per screen pixel...)
Resample to display’s pixel rate (box filter)
Resample to display's pixel rate (box filter)
Displayed result (note anti-aliased edges)
End:
Implementation of rasterization
Fragment generation: sampling coverage

Evaluate attributes (depth, u, v) at all covered samples
The graphics pipeline

- **Vertices**
  - Vertex Generation
  - Vertex Processing

- **Primitives**
  - Primitive Generation
  - Primitive Processing

- **Fragments**
  - Rasterization (Fragment Generation)
  - Fragment Processing

- **Pixels**
  - Frame-Buffer Ops

**Spaces**
- **Object/world/camera space**
- **Screen space**

**Output image buffer**

Stanford CS348V, Winter 2018
The graphics pipeline

- **Vertices**
  - 1 in / 1 out (for vertices)

- **Primitives**
  - 3 in / 1 out (for tris)
  - 1 in / small N out

- **Fragments**
  - 1 in / N out ( Fragment Generation )

- **Pixels**
  - Frame-Buffer Ops

**Memory**

- Uniform data

**Output image buffer**
struct input_fragment  
{  
    float  x,y;  
    float  z;  
    float3 normal;  
    float2 texcoord;  
};

struct output_fragment  
{  
    int    x,y; // pixel  
    float  z;  
    float4 color;  
};

texture my_texture;

output_fragment my_fragment_program(input_fragment in)  
{  
    output_fragment out;  
    float4 material_color = sample(my_texture, in.texcoord);

    for (each light L in scene)  
    {  
        out.color += shade(L) // compute reflectance towards camera due to L  
    }  

    return out;  
}
Example per-fragment operation: computing fragment color

e.g., sample texture map
Many different materials in the world

Images from Matusik et al. SIGGRAPH 2003
Materials

[Image credit: Jakob et al. 2014]
More complex materials

Fresnel reflection: reflectance is a function of viewing angle (notice higher reflectance near grazing angles)

Anisotropic reflection: reflectance depends on azimuthal angle (e.g., oriented microfacets in brushed steel)
Subsurface scattering materials

[Wann Jensen et al. 2001]

- Account for scattering inside surface
- Light exits surface from different location it enters
  - Very important to appearance of translucent materials (e.g., skin, foliage, marble)
The graphics pipeline

Vertices
- **1 in / 1 out**

Primitives
- **3 in / 1 out** (for tris)

Fragments
- **1 in / small N out**

** 1 in / N out**

** 1 in / 1 out**

** can be 0 out

Frames
- **1 in / 1 out**

** 1 in / out**

** can be 0 out

Memory
- Uniform data
- Texture buffers
Frame-buffer operations

- **Key responsibilities:**
  - Accumulate/blend fragment color into frame buffer based on “depth test”
Implementation of depth testing
Occlusion using the depth-buffer (Z-buffer)

For each coverage sample point, depth-buffer stores depth of closest triangle at this sample point that has been processed by the renderer so far.

Closest triangle at sample point \((x,y)\) is triangle with minimum depth at \((x,y)\)

Initial state of depth buffer before rendering any triangles (all samples store farthest distance)

Grayscale value of sample point used to indicate distance

Black = small distance
White = large distance
Depth buffer example
Example: rendering three opaque triangles
Occlusion using the depth-buffer (Z-buffer)

Processing yellow triangle:
depth = 0.5

Grayscale value of sample point used to indicate distance
White = large distance
Black = small distance
Red = sample passed depth test

Color buffer contents

Depth buffer contents
# Occlusion using the depth-buffer (Z-buffer)

**After processing yellow triangle:**

<table>
<thead>
<tr>
<th>Color buffer contents</th>
<th>Depth buffer contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Color buffer contents" /></td>
<td><img src="image2.png" alt="Depth buffer contents" /></td>
</tr>
</tbody>
</table>

Grayscale value of sample point used to indicate distance:
- White = large distance
- Black = small distance
- Red = sample passed depth test
Occlusion using the depth-buffer (Z-buffer)

Processing blue triangle:
depth = 0.75

Grayscale value of sample point used to indicate distance
White = large distance
Black = small distance
Red = sample passed depth test

Color buffer contents

Depth buffer contents
Occlusion using the depth-buffer (Z-buffer)

After processing blue triangle:

Color buffer contents

Depth buffer contents

Grayscale value of sample point used to indicate distance
White = large distance
Black = small distance
Red = sample passed depth test
Occlusion using the depth-buffer (Z-buffer)

Processing red triangle:
depth = 0.25

Grayscale value of sample point used to indicate distance
White = large distance
Black = small distance
Red = sample passed depth test

Color buffer contents

Depth buffer contents
Occlusion using the depth-buffer (Z-buffer)

After processing red triangle:

Color buffer contents

Depth buffer contents

Grayscale value of sample point used to indicate distance
White = large distance
Black = small distance
Red = sample passed depth test

White = large distance
Black = small distance
Red = sample passed depth test
Occlusion using the depth buffer

```c
bool pass_depth_test(d1, d2) {
    return d1 < d2;
}

depth_test(tri_d, tri_color, x, y) {
    if (pass_depth_test(tri_d, zbuffer[x][y])) {
        // triangle is closest object seen so far at this
        // sample point. Update depth and color buffers.
        zbuffer[x][y] = tri_d;       // update zbuffer
        color[x][y] = tri_color;    // update color buffer
    }
}
```
Does depth-buffer algorithm handle interpenetrating surfaces?

Of course!

Occlusion test is based on depth of triangles at a given sample point. The relative depth of triangles may be different at different sample points.
Does depth-buffer algorithm handle interpenetrating surfaces?

Of course!

Occlusion test is based on depth of triangles at a given sample point. The relative depth of triangles may be different at different sample points.
Summary: occlusion using a depth buffer

- Store one depth value per coverage sample (not per pixel!)

- Constant space per sample
  - Implication: constant space for depth buffer

- Constant time occlusion test per covered sample
  - Read-modify write of depth buffer if “pass” depth test
  - Just a read if “fail”

- Not specific to triangles: only requires that surface depth can be evaluated at a screen sample point
Early occlusion-culling ("early Z")

Idea: GPU discards fragments that are known to not contribute to image as early as possible in the pipeline.
Early occlusion-culling ("early Z")

A GPU implementation detail: not reflected in the graphics pipeline abstraction

Key assumption: occlusion results do not depend on fragment shading
- Example operations that prevent use of this early Z optimization: enabling alpha test, fragment shader modifies fragment’s Z value

Note: early Z only provides benefit if closer triangle is rendered by application first!
(application developers are encouraged to submit geometry in as close to front-to-back order as possible)
End:
Implementation of depth testing
Frame-buffer operations (full view)

```
struct output_fragment {
    int   x, y;
    float z;
    float4 color;
};

if (fragment.z < zbuffer[fragment.x][fragment.y]) {
    zbuffer[fragment.x][fragment.y] = fragment.z;
    color_buffer[fragment.x][fragment.y] = blend(color_buffer[fragment.x][fragment.y], fragment.color);
}
```

Depth test (hidden surface removal)
The graphics pipeline

- **Vertices**
  - 1 in / 1 out
  - **Vertex Generation**
  - **Vertex Processing**

- **Primitives**
  - 3 in / 1 out (for tris)
  - **Primitive Generation**
  - **Primitive Processing**

- **Fragments**
  - 1 in / N out
  - **Rasterization (Fragment Generation)**
  - **Fragment Processing**

- **Pixels**
  - 1 in / 0 or 1 out
  - **Frame-Buffer Ops**

**Memory**

- Uniform data
- Texture buffers

- Output image buffer
## Programming the graphics pipeline

- Issue draw commands → output image contents change

<table>
<thead>
<tr>
<th>Command Type</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>State change</td>
<td>Bind shaders, textures, uniforms</td>
</tr>
<tr>
<td>Draw</td>
<td>Draw using vertex buffer for object 1</td>
</tr>
<tr>
<td>State change</td>
<td>Bind new uniforms</td>
</tr>
<tr>
<td>Draw</td>
<td>Draw using vertex buffer for object 2</td>
</tr>
<tr>
<td>State change</td>
<td>Bind new shader</td>
</tr>
<tr>
<td>Draw</td>
<td>Draw using vertex buffer for object 3</td>
</tr>
<tr>
<td>State change</td>
<td>Change depth test function</td>
</tr>
<tr>
<td>State change</td>
<td>Bind new shader</td>
</tr>
<tr>
<td>Draw</td>
<td>Draw using vertex buffer for object 4</td>
</tr>
</tbody>
</table>

Note: efficiently managing stage changes is a major challenge in implementations.
A series of graphics pipeline commands

State change (set “red” shader)
Draw
State change (set “blue” shader)
Draw
Draw
Draw
State change (change blend mode)
State change (set “yellow” shader
Draw
Feedback loop 1: use output image as input texture in later draw command

- Issue draw commands → output image contents change

<table>
<thead>
<tr>
<th>Command Type</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Draw</td>
<td>Draw using vertex buffer for object 5</td>
</tr>
<tr>
<td>Draw</td>
<td>Draw using vertex buffer for object 6</td>
</tr>
<tr>
<td>State change</td>
<td>Bind contents of output image as texture 1</td>
</tr>
<tr>
<td>Draw</td>
<td>Draw using vertex buffer for object 5</td>
</tr>
<tr>
<td>Draw</td>
<td>Draw using vertex buffer for object 6</td>
</tr>
</tbody>
</table>

Rendering to textures for later use is key technique when implementing:
- Shadows
- Environment mapping
- Post-processing effects
Feedback loop 2: output intermediate geometry for use in later draw command

- Issue draw commands  →  output image contents change

- Vertices
  - 1 in / 1 out
  - Vertex Generation
  - Vertex Processing
  - Memory
  - Uniform data
  - Texture buffers

- Primitives
  - 3 in / 1 out (for tris)
  - Primitive Generation
  - Primitive Processing
  - Uniform data
  - Texture buffers

- Output vertex buffer

Stanford CS348V, Winter 2018
Analyzing the design of the graphics pipeline

- Level of abstraction
- Orthogonality of abstractions
- How is pipeline designed for performance/scalability?
- What the pipeline does and DOES NOT do

* These are great questions to ask yourself about any system we discuss in this course
Level of abstraction

- Imperative abstraction, not declarative
  - Application code specifies: “draw these triangles, using this fragment shader, with depth testing on”.
  - It does not specify: “draw a cow made of marble on a sunny day”

- Programmable stages provide application large amount of flexibility (e.g., to implement wide variety of materials and lighting techniques)

- Configurable (but not programmable) pipeline structure: application can turn stages on and off, create feedback loops

- Abstraction is low enough to allow application to implement many techniques, but high enough to abstract over radically different GPU implementations (NVIDIA, AMD, Intel GPUs, mobile GPUs, etc.)
Orthogonality of abstractions

- All vertices treated the same regardless of primitive type
  - Result: vertex programs are oblivious to primitive types
  - The same vertex program works for triangles and lines

- All primitives are converted into fragments for per-pixel shading and frame-buffer operations
  - Fragment programs are oblivious to source primitive type and the behavior of the vertex program *
  - Z-buffer is a common representation used to perform occlusion for any primitive that can be converted into fragments

* Almost oblivious. Vertex shader must make sure it passes along all inputs required by the fragment shader
What the pipeline DOES NOT do (non-goals)

- Modern graphics pipeline has no concept of lights, materials, geometric modeling transforms
  - Only streams of records processed by application defined kernels: vertices, primitives, fragments, pixels
  - And pipeline state (input/output buffers, “shaders”, and fixed-function configuration parameters)
  - Applications implement lights, materials, etc. using these basic abstractions

- The graphics pipeline has no concept of a scene

- It is just a virtual machine that executes pipeline state change and primitive drawing commands
Pipeline design facilitates performance/scalability

- [Reasonably] low level: low abstraction distance to implementation
- Constraints on pipeline structure:
  - Constrained data flow between stages
  - Fixed-function stages for common and difficult to parallelize tasks
  - Shaders: independent processing of each data element (enables data parallelism)
- Provide frequencies of computation (per vertex, per primitive, per fragment)
  - Application can choose to perform work at the rate required
- Keep it simple:
  - Only a few common intermediate representations
    - Triangles, points, lines
    - Fragments, pixels
  - Z-buffer algorithm computes visibility for any primitive type
- “Immediate-mode system”: pipeline processes primitives as it receives them
  (as opposed to buffering the entire scene)
  - Leave global optimization of how to render scene to the application
Perspective from Kurt Akeley

- Does the system meet original design goals, and then do much more than was originally imagined?

- If so, the design is a good one!
  - Simple, orthogonal concepts often produce this amplifier effect
Graphics pipeline implementation: GPUs

Specialized processors for executing graphics pipeline computations

Discrete GPU card (NVIDIA GeForce Titan X)

Integrated GPU: part of modern Intel CPU die
Modern GPUs offer many TFLOPs of performance for executing vertex and fragment shader programs.