Lecture 17:

The Real-Time 3D Graphics Pipeline

Visual Computing Systems
CMU 15-769, Fall 2016
Where we stand

Part 1: High-Efficiency Image Processing

The Digital Camera Image Processing Pipeline: Part I
From raw sensor measurements to an RGB image: demosaicing, correcting aberrations, color space conversions

The Digital Camera Image Processing Pipeline: Part II
JPG image compression, auto-focus/auto-exposure, high-dynamic range processing

Efficiently Scheduling Image Processing Algorithms on Multi-Core Hardware
Balancing parallelism/local/extra work, programming using Halide

Image Processing Algorithm Grab Bag
Fast bilateral filter and median filters, bilateral grid, optical flow

Specializing Hardware for Image Processing
Contrasting efficiency of GPUs, DSPs, Image Signal Processors, and FPGAs for image processing

H.264 Video Compression
Basics of H.264 video stream encoding

Part 2: Trends in Deep Neural Network Authoring and Acceleration

Basics of Deep Neural Network Evaluation
DNN topology, reduction to dense linear algebra, challenges of direct implementation

Parallel DNN Training
basics of back-prop, stochastic gradient descent (SGD), memory footprint issues, parallelizing SGD

Performance/Accuracy Optimization Case Study: End-to-End Training for Object Detection
R-CNN, Fast R-CNN, and then Faster R-CNN

Optimizing DNN Inference via Approximation
pruning and sparsification techniques, precision reduction, temporal rate reductions

Imposing Task-Specific Structure on DNN Topology
image compression networks, cross-stitch networks, spatial transformer networks, convolutional pose machines

Hardware Accelerators for Deep Neural Network Evaluation (discussion only)
A comparison of the various recent hardware accelerator papers
Oct 23: Exam 1 Released (take home exam)
Where we stand

Part 3: Systems Challenges of 3D Reconstruction

**Large-Scale 3D Reconstruction + Image Retrieval**
City-scale 3D reconstruction, content-based image retrieval

**Real-Time 3D Reconstruction from RGBD Input**
dense 3D reconstruction methods, implicit scene representations (TSDF), KinectFusion, BundleFusion

Reconstructing 3D scene geometric from images/videos
Where we stand

Part 4: Real-Time 3D Rendering Systems

Architecture of the GPU-Accelerated Real-Time 3D Graphics Pipeline
Graphics pipeline abstractions, scheduling challenges

Rasterization and Occlusion
Hardware acceleration, depth and color compression algorithms

Texture Mapping
Texture sampling and prefiltering, texture compression, data layout optimizations

Parallel Scheduling of the Graphics Pipeline
Molnar taxonomy, scheduling under data amplification, tiled rendering

Deferred Shading and Image-Space Rendering Techniques
Deferred shading as a scheduling decision, image-space anti-aliasing

Hardware-Accelerated Ray Tracing
Ray-tracing as an alternative to rasterization, what does modern ray tracing HW do?

Shading Language Design
Contrasting different shading languages, is CUDA a DSL?

Case Study: The Spire Shading Language
Discussion of relationship to other recent DSLs

Part 5: Miscellaneous Topics

DSLs for Physical Simulation: Lizst and Ebb
Open research questions on high-performance DSL design
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 C 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 its suitability for efficient implementation
x86 architecture?

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

- **Operations:**
  - x86 instructions (privileged and non-privileged)
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
The 3D rendering problem

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

Output: image

Problem statement: How does each geometric element contribute 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
(A review of the 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 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

1. Vertices in 3D space
2. Vertices in positioned on screen
3. Triangles positioned on screen
4. Fragments (one per pixel covered by triangle *)

* 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**
3D graphics system stack

Application
(e.g., a computer game, a CAD application, a web browser)

Scene graph
(application's database representing the scene: geometry, materials, lights, etc.)

Graphics pipeline
(OpenGL/Direct3D)

Graphics pipeline implementation
(software driver + GPU)

clients to the system (use the abstraction)

the abstraction we are discussing now

implements the abstraction
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

Memory

Output image buffer
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
V_0  V_1  ...  V_{N-1}
```

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

**Indexed access version ("gather")**

```
my_vtx_buffer
V_0  V_1  ...  V_{N-1}
```

```
my_vtx_indices  1  3  2  1  5  6
```

```c
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 pipelines set a limit of 32 float4 attributes per vertex (512 bytes)
Why? (to be answered in a later lecture)
What the vertex processing kernel does
Transform triangle vertices into camera space
What the vertex processing kernel does

Apply perspective projection transform to transform triangle vertices into normalized coordinate space

Camera-space positions: 3D

Normalized space positions
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

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

Per-vertex data: surface normal, surface color

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

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

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

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
- **Primitives**: 3 in / 1 out (for tris)
- **Fragments**: Rasterization (Fragment Generation)
- **Pixels**: Frame-Buffer Ops

**Memory**
- Uniform data

**Output image buffer**
Programmable primitive processing *

Input vertices for 1 prim → output vertices for N prims **

Independent processing of each INPUT primitive

* “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

Triangles before clipping

Triangles after clipping
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
  - Vertex Generation
  - Vertex Processing

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

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

- Pixels
  - Frame-Buffer Ops
  - Output image buffer

Memory

Uniform data

- Uniform data

Output image buffer
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

Vertex Generation

Vertex Processing

Primitive Generation

Primitive Processing

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)
}
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

Output image buffer

Object/world/camera space

screen space
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**
  - **Frame-Buffer Ops**

**Memory**
- Uniform data
- Uniform data
- Output image buffer

**Vertices**
- Vertices
  - 1 in / 1 out

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

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

**Pixels**
- Pixels
  - Frame-Buffer Ops
Fragment processing

```c
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
The graphics pipeline

Vertices
- 1 in / 1 out

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

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

Pixels
- Frame-Buffer Ops

** can be 0 out

Memory
- Uniform data
- Texture buffers

Output image buffer
Frame-buffer operations

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

Pixel Operations

Frame Buffer

Memory
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:

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 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

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

Depth buffer contents
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
Occlusion using the depth buffer

```cpp
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
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
  - 3 in / 1 out (for tris)

- Primitives
  - 1 in / small N out

- Fragments
  - 1 in / N out

- Vertices Processing
- Primitive Generation
- Primitive Processing
- Rasterization (Fragment Generation)
- Fragment Processing
- Frame-Buffer Ops
- Memory
  - Uniform data
  - Texture buffers

- 1 in / 0 or 1 out

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><strong>Bind contents of output image as texture 1</strong></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 \[\Rightarrow\] output image contents change

### Diagram

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

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

- **Memory**
  - Uniform data
  - Texture buffers
  - Output vertex buffer
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: 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
Orthogonality of abstractions

- All vertices treated the same regardless of primitive type
  - Result: vertex programs 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, modeling transforms
  - Only vertices, primitives, fragments, pixels, and STATE (state = buffers, shaders, and configuration parameters)
  - Applications use these basic abstractions to implement lights, materials, etc.

- The graphics pipeline has no concept of a scene

- No I/O or OS window management
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 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

Homework exercise: describe one example of a graphics pipeline design decision that enables high-performance implementations.
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 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 ~2-4 TFLOPs of performance for executing vertex and fragment shader programs.

T-OP's of fixed-function compute capability over here.

Scheduler / Work Distributor

Modern GPUs offer ~2-4 TFLOPs of performance for executing vertex and fragment shader programs.

T-OP's of fixed-function compute capability over here.

Scheduler / Work Distributor