Lecture 1:
Course Introduction +
Review of the Real-Time Graphics Pipeline

Visual Computing Systems
CMU 15-869, Fall 2014
Why does this course exist?
Many applications that drive the need for high efficiency computing involve visual computing tasks

First thing that comes to mind: 3D “AAA” games
Efficiency gets you: more advanced graphics at 30 fps
Result: multi-TFLOP GPUs

Simple mobile games
Efficiency gets you: don’t run down the battery
Result: different rendering algorithms
Many applications that drive the need for high efficiency computing involve visual computing tasks.

Record/play HD Video

2D rendering to “Retina” resolution displays*: maps, browsers, 60 fps touch UIs

* Maps apps and web content have 3D rendering capabilities as well.
High pixel count sensors and displays

Nokia Lumina smartphone camera: 41 megapixel (MP) sensor

Nexus 10 Tablet: 2560 x 1600 pixel display (~ 4MP)
(higher pixel count than 27” Apple display on my desk)

4K TV

Rendering for VR and light-field displays: need for much higher pixel counts
Computational photography:

Current focus: achieve high-quality pictures with a lower-quality smartphone lenses/sensors through the use of image analysis and processing.

**Automatic panorama:**

**High dynamic range (HDR) imaging:**

Traditional photograph: part of image is saturated due to overexposure

HDR image: combine multiple exposures so image detail in both light and dark areas is preserved

Remove camera shake:
Image interpretation and understanding:

Extracting information from images recorded by ubiquitous image sensors

Big area of interest at both mobile device and data-center scales.

Auto-tagging, face (and smile) detection

Kinect: character pose estimation

Google Goggles: object identification search by image

Collision anticipation, obstacle detection
Enabling current and future visual computing applications requires focus on system efficiency.

In this class we are going to think like architects. Which means we’re going to talk a lot about a system’s goals and about its constraints.

Example goals:

- Real-time rendering of a one-million polygon scene at 30 fps on a high-res display
- Provide interactive user feedback when acquiring a panorama
- 1080p video recording for one hour per phone charge

Example constraints:

- Chip die area (chip manufacturing cost)
- System design complexity
- Preserve easy application development effort
- Backward compatibility with existing software
- Power
Parallelism and specialization in HW design

Example: NVIDIA Tegra K1

Four high-performance ARM Cortex A15 CPU cores for applications
One low performance (low power) ARM CPU core
One Kepler SMX core (to run graphics shaders and CUDA programs)
Fixed-function HW blocks for 3D graphics and image/video compression and camera image processing (image signal processor = ISP)

Design philosophy:
Run important workloads on the most efficient hardware for the job.

Other modern examples:
Apple A6X
Qualcomm Snapdragon
Specialized hardware is efficient!

ASIC delivers same performance as one CPU core with ~ 1/100th the chip area.

GPU cores: ~ 5-7 times more area efficient than CPU cores.

ASIC delivers same performance as one CPU core with only ~ 1/100th the power.
**Limits on chip power consumption**

- **General rule:** the longer a task runs the less power it can use
  - Processor’s power consumption (think: performance) is limited by heat generated (efficiency is required for more than just maximizing battery life)

**Graph**

- **Electrical limit:** max power that can be supplied to chip
- **Die temp:** (junction temp -- $T_j$): chip becomes unreliable above this temp (chip can run at high power for short period of time before chip heats to $T_j$
- **Case temp:** mobile device gets too hot for user to comfortably hold (chip is at suitable operating temp, but heat is dissipating into case)
- **Battery life:** chip and case are cool, but want to reduce power consumption to sustain long battery life for given task

**Graph Data**

- iPhone 5 battery: 5.4 watt-hours
- 4th gen iPad battery: 42.5 watt-hours
- 15in Macbook Pro: 95 watt-hours

Credit: slide adopted by original slide from M. Shebanow
What this course is about

1. The characteristics/requirements of important visual computing workloads
2. Techniques used to achieve efficient system implementations

VISUAL COMPUTING WORKLOADS
Algorithms for 3D graphics, image processing, compression, etc.

MACHINE ORGANIZATION
High-throughput hardware designs: Parallel and heterogeneous

DESIGN OF GOOD ABSTRACTIONS FOR VISUAL COMPUTING
choice of programming primitives
level of abstraction

mapping/scheduling
Parallelism
Exploiting locality
Minimizing communication
In other words

It is about understanding the fundamental structure of problems in the visual computing domain . . .

To design better algorithms

To build the most efficient hardware to run these applications

To design the right programming systems to make developing new applications simpler and also highly performant.
What this course is **NOT** about

- **This is not an [OpenGL, CUDA, OpenCL] programming course**
  - But we will be analyzing and critiquing the design of these systems in detail
  - I expect you know these systems or pick them up as you go.

Many excellent references...
Course Logistics
Major course themes/topics

Rendering systems:
Primarily real-time 3D graphics as implemented by modern games

High-performance image processing
Camera image pipeline (for photography)
Image processing for computer vision at scale.

Miscellaneous topics (may change)
Logistics

- **Course web site:**
  - http://15869.courses.cs.cmu.edu

- **All announcements will go out via Piazza**
  - https://piazza.com/cmu/fall2014/15869/home

- **Kayvon’s office hours:** drop in or by appointment (EDSH 225)

- **Your knowledgable TA:** Yong He (GHC 7117)
Expectations of you

- **30% participation**
  - There will be ~1-2 assigned paper readings per class
  - Everyone is expected to come to class and participate in discussions based on readings
  - You are encouraged discuss papers and or my lectures on the course discussion board.
  - If you form a weekly course reading/study group, I will buy Pizza.

- **25% mini-assignments (2-3 short programming assignments)**
  - Implement a basic parallel triangle renderer
  - Implement a RAW image processing pipeline

- **45% self-selected final project**
  - I suggest you start talking to me now (can be teams of up to two)

- **We have toys to play around with throughout the semester:**
  - You are encouraged to experiment with them and report what you learn back to the class
  - Two Oculus DK2s
  - Two NVIDIA Shields, one Jetson K1
Somewhat philosophical question:
What is an “architecture”?
An architecture is an abstraction

It defines:

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

- **Operations (that manipulate state)**
  - Add 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 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
Choice impacts system’s expressiveness/scope and its suitability for efficient implementation.
The 3D rendering problem

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

Output: image

Main problem statement: How does each geometric element contribute to the appearance of each output pixel in the image, given a description of surface properties and lighting conditions.
The real-time graphics pipeline architecture
(A review of the GPU-accelerated OpenGL/D3D graphics pipeline, from a systems perspective)
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. 4. Buffers, textures

Output image buffer
3D graphics system stack

Application
(e.g., a computer game)

↓

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 review *

- 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
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
"Assembling" vertices

Vertex Generation

Vertex records

Vertex Processing

Contiguous version data version

```c
my_vtx_buffer

V0 V1 Vn-1

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

Indexed access version ("gather")

```c
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 pipelines set a limit of 32 float4 attributes per vertex. (512 bytes) Why? (to be answered in a later lecture)
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

Vertex Shader Program *

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

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

uniform mat4 my_transform;

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 lighting

Per-vertex data: surface normal, surface color

Uniform data: light direction, light color
Example: vertex 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
The graphics pipeline

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

Memory:
- Uniform data

Output image buffer
Primitive processing *

Vertex Generation ➔ Vertex Processing ➔ Primitive Generation ➔ Primitive Processing

Memory

Uniform data

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
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**

- **Memory**
  - Uniform data

- **Output image buffer**
Rasterization

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 attrs
  float2 texcoord; // (e.g., texture coordinates, surface normal)
};
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)
}

Compute covered pixels
Sample vertex attributes once per covered pixel
The graphics pipeline

Vertices
- Vertex Generation
  - Vertex Processing

Primitives
- Primitive Generation
  - Primitive Processing

Fragments
- Rasterization (Fragment Generation)
  - Fragment Processing

Pixels
- Frame-Buffer Ops

Object/world/camera space

screen space

Output image buffer
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
- **Pixels**: 1 in

**Memory**

- Uniform data

**Flow**

1. **Vertex Generation**
2. **Vertex Processing**
3. **Primitive Generation**
4. **Primitive Processing**
5. **Rasterization (Fragment Generation)**
6. **Fragment Processing**
7. **Frame-Buffer Ops**
8. **Output image buffer**

**Data Types**

- **Vertices**: 1 in / 1 out
- **Primitives**: 3 in / 1 out (for tris), 1 in / small N out
- **Fragments**: 1 in / N out
- **Pixels**: 1 in

**Uniform Data**

- 1 in / small N out
- 1 in / N out
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_vertex 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;
}
```
The graphics pipeline

- **Vertices**: 1 in / 1 out
- **Primitives**: 3 in / 1 out (for tris)
- **Fragments**: 1 in / small N out
- **Pixels**: 1 in / N out

**Stage Descriptions**

- **Vertex Generation**
- **Vertex Processing**
- **Primitive Generation**
- **Primitive Processing**
- **Rasterization** (Fragment Generation)
- **Fragment Processing**
- **Frame-Buffer Ops**

**Memory Blocks**

- Uniform data
- Texture buffers

**Outputs**

- **Output image buffer**

**Notes**

- **can be 0 out**
Frame-buffer operations

```c
struct output_fragment {
    int x, y;
    float z;
    float4 color;
};
```
Frame-buffer operations

Depth test (hidden surface removal)

```c
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);
}
```
Frame-buffer operations (full view)

```c
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

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

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

**Pixel**
- 1 in / 0 or 1 out

**Processes**
- Vertex Generation
- Vertex Processing
- Primitive Generation
- Primitive Processing
- Rasterization
- Fragment Processing
- 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

<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  ➔ save intermediate geometry

### Memory

- Uniform data
- Texture buffers

### Vertices

- 1 in / 1 out

### Primitives

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

### Vertices Processing

### Primitive Generation

### Primitive Processing
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
Readings

- **Required**

- **Suggested:**
  - Chapter 2 and 3 of Real-Time Rendering, Third Edition (see link on course site)
  - M. Segal and K. Akeley. *The Design of the OpenGL Graphics Interface*