Lecture 1:

Course Intro +

The Real-Time Graphics Pipeline

Visual Computing Systems
CMU 15-869, Fall 2013
Many applications driving the need for high efficiency computing involve visual computing tasks.
Many applications driving the need for high efficiency computing involve visual computing tasks.

Record/play HD Video

2D and 3D rendering: games, browsers, maps

Oculus Rift VR display (presents new graphics system requirements)
Computational photography:

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

- Automatic panorama:
- Lighting/color/tone adjustment:
- High dynamic range (HDR) imaging:
  - Traditional photograph: part of image is saturated due to overexposure
  - HDR image: image detail in both light and dark areas is preserved

Remove camera shake:
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)
Image interpretation and understanding:
(extracting value from images recorded by ubiquitous image sensors)

Auto-tagging, face (and smile) detection

Google Goggles: search by image

Kinect: character pose estimation

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

A systems architect must meet challenging application goals within specific design constraints.

Example goals:
- Real-time rendering of a 1M polygon scene on high resolution display
- Interactive user feedback when acquiring a panorama
- HD video recording for 1 hour per phone charge

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

Example: NVIDIA Tegra 4 system-on-a-chip

- Four high-performance ARM CPU cores
- One low performance (low power) ARM CPU core
- 72 GPU shader processors (run shader programs)
- Chimera ISP (image/video processing for camera)
- Fixed-function HW blocks for 3D graphics and image compression

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

Other modern examples:
Apple A6X
Qualcomm Snapdragon
Hardware specialization increases efficiency

**Area-normalized FFT Performance (40nm)**

- 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.

**FFT Energy Efficiency (40nm)**

- ASIC delivers same performance as one CPU core with only ~ 1/100th the power.

[Chung et al. MICRO 2010]
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)

- **Electrical limit**: max power that can be supplied to chip
- **Die temp**: (junction temp -- Tj): chip becomes unreliable above this temp (chip can run at high power for short period of time before chip heats to Tj)
- **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

---

*Credit: slide adopted by original slide from M. Shebanow*

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

- Run faster for a fixed period of time
  - Run at higher clock, use more cores (reduce latency of critical task)
  - Do more at once

- Run at a fixed level of performance for longer
  - e.g., video playback
  - Achieve “always-on” functionality that was previously impossible

### iPhone 5:
Siri activated by button press or holding phone up to ear

### Moto X:
Always listening for “ok, google now”

Device contains special ASIC for detecting this audio pattern.
Efficiency matters in desktop/server contexts as well

- For a hardware architect
  - Power efficiency
    - Maximize performance given power budget
    - Reduce cost (simpler heat dissipation mechanism)
  - Chip area efficiency (smaller chip = lower cost)

- For a software developer: enable new applications!
  - Achieve real-time rates for new classes of problems
  - Scale applications to much bigger datasets
  - Deploy applications in new settings (mobile, always on)
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**
(3D graphics, image processing, etc.)

**MACHINE ORGANIZATION**
Parallelism, heterogeneity throughput processing
The role of fixed-function HW

**DESIGN OF ABSTRACTIONS**
(e.g., the real-time graphics pipeline)
choice of primitives
level of abstraction

**m**apping/scheduling
Parallelism
Exploiting locality
Communication
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 abstractions in detail

Many excellent references...
Major course themes/topics

- Three major application areas
  1. Real-time 3D rendering: the real-time graphics pipeline and trends in interactive rendering techniques
  2. Image processing: the digital camera pipeline and basic computational photography workloads
  3. Image retrieval and visual data mining: systems for managing billions of images

- Reoccurring course themes
  - Understanding key computational characteristics of workloads
  - Understanding constraints of modern parallel machine architectures
  - End-to-end thinking: workloads influencing hardware design, and parallel hardware constraints influencing the design of algorithms
  - Defining good abstractions: identifying fundamental system primitives and operations
  - Tensions between maximizing efficiency and retaining programmability
Course Logistics
Logistics

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

- **Announcements will go out via Piazza**
  - https://piazza.com/cmu/fall2013/15869/home

- **Office hours: drop in or by appointment (EDSH 225)**

- **I hope to have a number of Friday (noon-1:20pm) sessions**
Grades / expectations

- 30% readings and summaries (approximately one required paper per class)
  - Everyone is expected to come to class and participate in discussions

- 25% mini-assignments (2-3 programming assignments + 1 written)
  - Will also release optional assignments that undergrads may perform as part of their project component

- 45% self-selected final project
  - Start talking to me now
What is an architecture?
Aspects of an architecture (system abstraction)

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

- **Operations (that manipulate things)**
  - Add registers, copy buffer, multiply vectors, blur image, draw triangle

- **Mechanisms for instantiating entities and expressing operations**
  - Execute machine instruction, make C++ API call, express logic in programming language

Notice different levels of granularity/abstraction in 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.
3D rendering problem

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

Output: image

How does each mesh triangle contribute to each pixel in the image, given model’s description of surface properties and lighting conditions.
The real-time graphics pipeline architecture
(A review of the OpenGL 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
   - Vertex Generation: Vertex stream
   - Vertex Processing: Vertex stream

2. Primitives
   - Primitive Generation: Primitive stream
   - Primitive Processing: Primitive stream

3. Fragments
   - Fragment Generation (Rasterization): Fragment stream
   - Fragment Processing: Fragment stream
   - Triangles positioned on screen
   - Fragments (one per pixel covered by triangle *)
   - Shaded fragments

4. Output image (pixels)

* 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)**
- 

- Vertex data buffers
- Buffers, textures
- Vertex transform matrices
- Buffers, textures
- Buffers, textures

**Output image buffer**

CMU 15-869, Fall 2013
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

Implements the abstraction
Issues to keep in mind

- Level of abstraction

- Orthogonality of abstractions

- How is it designed for performance/scalability?

- What a system does and DOES NOT do
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
“Assembling vertices”

Contiguous Version

my_vtx_buffer

V₀ V₁ V_N-1

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

Indexed Version (gather)

my_vtx_buffer

V₀ V₁ V_N-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”

Current pipelines set limit of 16 float4 (128 bit) attributes per vertex.
Uniform data: constant read-only data provided as input to every instance of the vertex shader e.g., vertex transform matrix
Vertex stage inputs

```
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 multiply
  return out;
}
```

Vertex Shader Program *

(* Note: for clarity, this is not valid GLSL syntax)
Vertex processing example: lighting

Per vertex data: surface normal, surface color
Uniform data: light direction, light color
Vertex processing example: 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
  - Vertex Generation
  - Vertex Processing

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

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

- **Pixels**: Frame-Buffer Ops

Memory

- Uniform data

Output image buffer
Primitive processing

- **Primitive Generation**
  - **Vertex Generation**
  - **Vertex Processing**
  - **Primitive Generation**
  - **Primitive Processing**

* Pipeline caps output at 1024 floats of output

**Memory**

- Uniform data

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

**Independent processing of each INPUT primitive**

* CMU 15-869, Fall 2013
The graphics pipeline

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

Memory:
- Uniform data

Output image buffer
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)
Rasterization

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

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

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

- **Pixels**
  - Frame-Buffer Ops

**Flowchart:***

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

**Memory:**
- Uniform data
- Output image buffer
Fragment processing

```
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;
}
```
Many uses for textures

Provide surface color/reflectance

Tom Porter’s Bowling Pin

Source: RenderMan Companion, Pls. 12 & 13
Bump mapping:
Displace surface in direction of normal (for lighting calculations)
Normal mapping

Modulate interpolated surface normal

\[(nx, ny, nz) = (r, g, b)\]
Many uses for textures

Store precomputed lighting

Blinn and Newell, 1976

Percentage of hemisphere visible

From Production ready global illumination, Hayden Landis, ILM

Slide credit: Pat Hanrahan
The graphics pipeline

** can be 0 out
Frame-buffer operations

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

Diagram:
- Frame Buffer
- Memory
- Pixel Operations

CMU 15-869, Fall 2013
Frame-buffer operations

```
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)
```
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);
}
```
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
  - Rasterization (Fragment Generation)

- **Frames**
  - 1 in / 1 out

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

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

**Memory**

- Uniform data
- Texture buffers

**Buffers**

- 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
Using the pipeline to create feedback loops

- 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>

Key idea for:
- shadows
- environment mapping
- post-processing effects

Modern games: 1000-1500 draw calls per frame
(source: Johan Andersson, DICE -- circa 1998)
Feedback loop: store intermediate geometry

- Issue draw commands ➔ save intermediate geometry

**Vertex Generation**
- Vertices: 1 in / 1 out
- Primitives: 3 in / 1 out (for tris)

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

**Primitive Generation**
- Primitives: 1 in / small N out

**Primitive Processing**
- Primitives: 1 in / small N out

Memory:
- Uniform data
- Texture buffers

Output vertex buffer
OpenGL state diagram (OGL 1.1)
Graphics pipeline with tessellation
(OpenGL 4, Direct3D 11)
Graphics pipeline characteristics

- **Level of abstraction**

  - Imperative abstraction, not declarative
    (Application says “draw these triangles, using this fragment shader, with depth testing on” rather than “draw a cow made of marble on a sunny day”)

  - **Programmable** stages give large amount of application 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 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
  - 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 oblivious to 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
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
What the pipeline DOES NOT do (non-goals)

- Pipeline has no concept of lights, materials, modeling transforms
  - Only vertices, primitives, fragments, pixels, and STATE
    - such as buffers, shaders, and config parameters
  - Applications use these basic abstractions to implement lights, materials, etc.
- Pipeline has no concept of a scene
- No I/O or OS window management
Perspective from Kurt Akeley

- Does the system meet original design goals, and then do much more than was originally imagined?
  - Simple, orthogonal concepts produce amplifier effect

- Often you’ve done a good job if neither system implementers nor system users are perfectly happy ;-) (of course, you still have to meet design goals)
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*