Intro to OpenGL & Rasterization

Computer Graphics
CMU 15-462/662
Graphics APIs

- Graphics APIs perform many common rendering tasks for you
  - Important: you don’t need a graphics API to do graphics!
  - Generally Graphics APIs are used for real-time rendering

- Why use a Graphics API instead of writing it yourself?
  - Graphics hardware is designed around them (huge speedup!)
    - Graphics drivers translate standard graphics API calls to specialized GPU hardware instructions
  - Standardization of Graphics APIs allows for the creation of better debugging and tooling
    - This is why DirectX is so popular!
  - It takes way less time
Realtime Rendering as Smoke & Mirrors

“Phantasmagoria” in Theatre

“Potemkin Villages”
Realtime Rendering as Smoke & Mirrors

Billboarding in Mario 64

Rim Lighting
(http://blog.wolfire.com/2009/09/character-rim-lighting/)

“The Realtime Volumetric Cloudscapes of Horizon Zero Dawn,” Guerrilla Games @ SIGGRAPH 2015
**Rasterization vs Pathtracing**

**Rasterization**
Transform scene geometry via matrix operations to screen space, then use triangle fill algorithm.

*Optimized for performance*

DrawSVG (A1)

---

**Pathtracing**
Bounce simulated rays of light throughout your scene randomly for each pixel, and illuminate if it eventually intersects a light.

*Optimized for realism*

Pathtracer (A3)
The GPU

- The the GPU (Graphics Processing Unit) in your computer is designed to accelerate rasterization
  - CPU: ~4-16 “smart” cores
    - Handle branching well, but lower throughput for math
  - GPU: Many (thousands) of “dumb” compute units
    - Bad at branching, very fast at math, great parallelism

- Upshot: GPUs are very good at matrix / vector operations, and bad at handling branching (if statements, nontrivial loops, etc)

- We will generally not use the GPU in this class (see 15-418!)
Multi-core examples

Intel “Skylake” Core i7 quad-core CPU (2015)
Each core is sophisticated, out-of-order processor

NVIDIA GTX 980 GPU
16 replicated processing cores (“SM”) (2014)
Each core processors vectors of data

Check out 15-418!

CMU 15-418/618, Spring 2019
Things GPUs Are Good At: Triangle Fill

Fact: GPUs can fill in triangles on a 2D image plane quickly
Consider the matrix: \[ A = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \]

Question: What happens when you multiply \( A \) by a vector?

\[
\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} 5 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ -5 \end{bmatrix} \]
\[
\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 2 \\ -1 \end{bmatrix}
\]

Answer: \( A \) represents a 90 degree clockwise rotation about the origin!

- It turns out you can encode arbitrary rotations in a matrix (even in 3D!)
Things GPUs Are Good At: Transformation Matrices

- We can encode scaling (using 3D vectors here):

\[
\begin{pmatrix}
3 & 0 & 0 \\
0 & 5 & 0 \\
0 & 0 & 1/2
\end{pmatrix}
\begin{pmatrix}
1 \\
2 \\
3
\end{pmatrix}
= 
\begin{pmatrix}
3 \\
10 \\
3/2
\end{pmatrix}
\]

- If we cheat a little and add a 4th coordinate set to 1, we can also encode translation:

\[
\begin{pmatrix}
1 & 0 & 0 & 5 \\
0 & 1 & 0 & -2 \\
0 & 0 & 1 & 4 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
1 \\
2 \\
3 \\
1
\end{pmatrix}
= 
\begin{pmatrix}
6 \\
0 \\
7 \\
1
\end{pmatrix}
\]

XYZ Coordinates

W Coordinate used for translation
Things GPUs Are Good At: Transformation Matrices

The *projection matrix* accounts for objects vanishing in the distance due to perspective.

- After applying the projection matrix, you can throw away the z coordinate.
Things GPUs Are Good At: Transformation Matrices

- We can even combine these matrices into a single matrix: Let $T, R, S$ denote translation / rotation / scale matrices and let $\vec{v}$ be a point in 3D space.
  
  - Then $T \times (R \times (S \times (\vec{v})))$ means “scale $\vec{v}$ about origin, then rotate result about origin, then translate result”
  
  - But by associativity of matrix multiplication this is equivalent to $(T \times R \times S) \times \vec{v}$!

- This is very useful: we can encode arbitrary geometric transformations into a single matrix.

More on this in a later lecture!
Navigating Coordinate Frames

When we want to draw something on screen, it is loaded in via a 3D model file (.obj, .dae etc) with its own coordinate frame. We call this coordinate frame model space.

Figure source: [http://www.codinglabs.net/article_world_view_projection_matrix.aspx](http://www.codinglabs.net/article_world_view_projection_matrix.aspx)
Navigating Coordinate Frames

Suppose we wanted to place three of these teapots in a scene, each with their own position, rotation and scale.

- We can use the translation / rotation / scale (TRS) matrix discussed earlier to move them into world space.
- The model matrix is applied to each vertex to bring it from model space to world space.

Figure source: http://www.codinglabs.net/article_world_view_projection_matrix.aspx
Navigating Coordinate Frames

Of course, we also must have a camera that views the scene. We define view space to be the frame with the camera position as the origin and the camera direction as the Z-direction.

- Question: How do we build the view matrix, which converts from world space to view space?

Figure source: [http://www.codinglabs.net/article_world_view_projection_matrix.aspx](http://www.codinglabs.net/article_world_view_projection_matrix.aspx)
Navigating Coordinate Frames

Finally, the projection matrix brings us from view space to screen space (coordinate frame of the window)

- Depends on Field of View of the camera

Now, we just need to run triangle fill in screen space!

Figure source: http://www.codinglabs.net/article_world_view_projection_matrix.aspx
Putting it All Together: Rasterization is a Data Processing Pipeline

Input Vertex Coordinates (from .obj, .dae, etc)

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

Modelspace → Screenspace happens here!

Input Primitives (Triangle, Quad, Line, etc)

- **Pixel Operations** (Stencil, Blending, Dithering)
- **Fragment Processing**
- “Fragment” generation (Triangle Fill)

Output Image

All we need to do: tell OpenGL what our input points and transformation matrices are, and what color to fill triangles with!
Code!

https://github.com/Flafla2/GLTutorial

(See /checkpoints folder to review later)
Common Realtime Graphics APIs

- **OpenGL**: Runs on all platforms, but old, slow, and falling out of fashion. Mac support ending soon.
  - OpenGL ES: Subset of OpenGL for mobile GPUs
  - GL 1.1 != GL 2.1 != GL 3.3 != GL 4.6
    ‣ Massive breaking API changes between GL versions — hard to find tutorials!
    ‣ … and that’s not even counting extensions!
- **Vulkan**: Modern Graphics API, runs on every platform (macOS needs a Metal wrapper)
- **DirectX**: Windows / Xbox only, very popular in Game development due to engine support and tooling
  - DirectX 9: Used on Xbox 360 / WinXP, similar to GL 2.x
  - DirectX 10: Used on Xbox 360 / Vista, similar to GL 3.x
  - DirectX 11: Used on Xbox 360 / Xbox One / Win7, similar to GL 4.x
  - DirectX 12: Used on Xbox One, similar to Vulkan
- **Metal**: Apple’s low level graphics API for iOS / Mac
Choosing a Graphics API

■ The graphics API you should use depends on:
  - Platform(s) you are publishing on (including OS!)
  - **Example**: On Windows DirectX performs better than OpenGL due to driver support

■ Specific API Features
  - **Example**: Vulkan offers lower level control of GPU memory than OpenGL, but may be harder to use
  - Whatever new hotness comes out tends to take a while to arrive to all graphics APIs *(cough raytracing)*

■ Your own preference / familiarity
  - Much more important when considering shader languages
DrawSVG

- 5% of the assignment: regurgitate today’s demo in `hardware_renderer.cpp`
  - Everything we talked about today, up to blending

- 95% of the assignment: reimplement OpenGL calls on the CPU (using good old C++)!
Not discussed:
Shaders are highly parallel GPU programs

- **Input Vertex Coordinates**
  - (from .obj, .dae, etc)

- **Vertex Generation**
  - "Vertex Shader"

- **Vertex Processing**

- **Primitive Generation**
  - Input Primitives (Triangle, Quad, Line, etc)

- **Fragment Processing**
  - "Fragment Shader"

- **Pixel Operations**
  - (Stencil, Blending, Dithering)

- **Output Image**

- **"Fragment" generation**
  - (Triangle Fill)
Modern Graphics APIs are More Complex
(but more powerful too!)

Pictured: DirectX 12 Graphics Pipeline

Modern Graphics APIs are More Complex
(but more powerful too!)

Pictured: Vulkan Graphics Pipeline

Wrap-up

- OpenGL and similar APIs are the bread and butter of practical computer graphics, so start learning them!
  - Recommend: 15-466 Computer Game Programming which uses OpenGL 3.3
  - Great tutorial for modern OpenGL: http://learnopengl.com

- Check out: Shaders are highly parallel code compiled for the GPU. Modern graphics libraries use shaders to implement many common GL 2.1 “fixed function” effects.
  - http://shadertoy.com (simple example here)
  - There is a shading language for all modern graphics apis