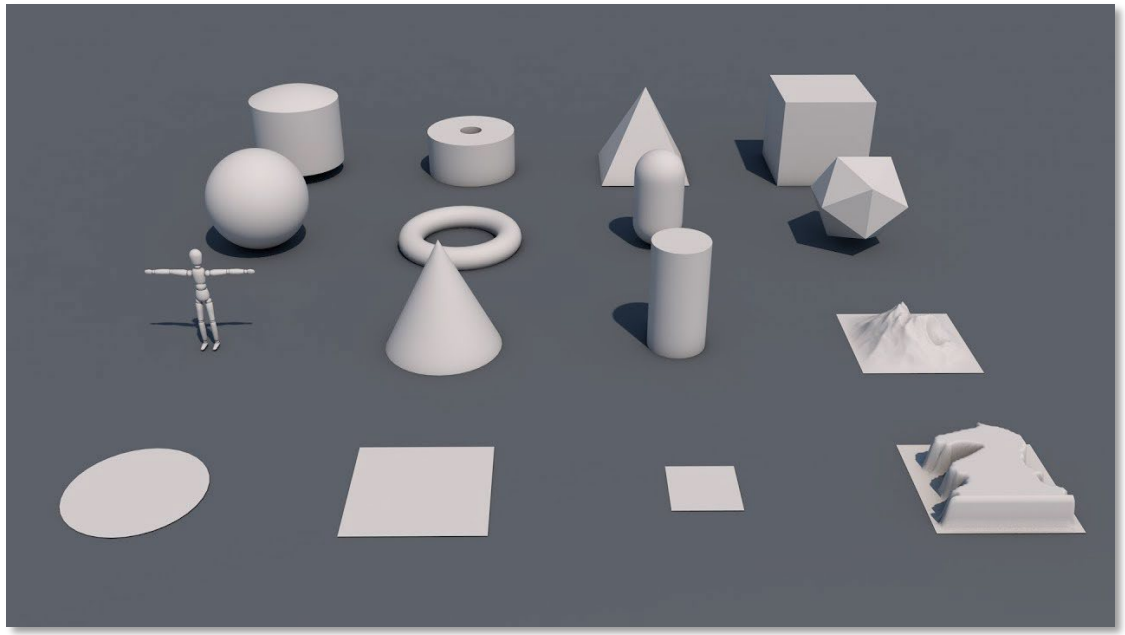
Real-Time Graphics

An overview

Computer graphics



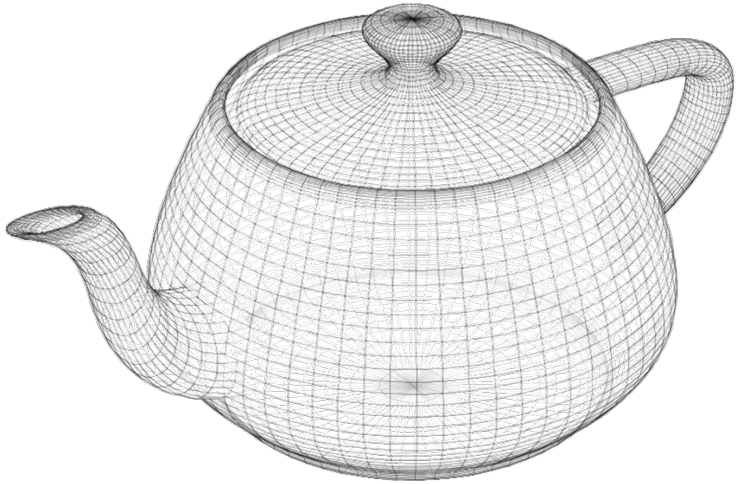
- Encompasses many topics
  - Games
  - Film
  - Photography
  - Image processing
  - Computer vision
  - Etc.

- Our focus: real-time graphics for games
  - Rendering a 3D world at 60+ frames per second
  - Using modern GPU hardware

3D rendering

- Generating an image from one or more 3D models

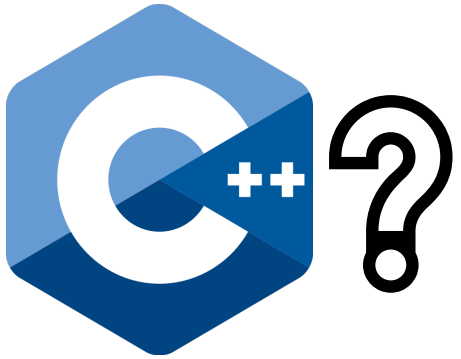
- 3D models comprised of triangles
  - Each triangle has 3 vertices
  - Each vertex contains surface information



- Special hardware not *strictly* necessary
  - Can be done completely in software
  - C++, C#, Javascript, any language

3D rendering for games

- We *can* perform rendering in software
  - Often called “offline rendering”
  - But games generally *don’t*
  - Too slow!

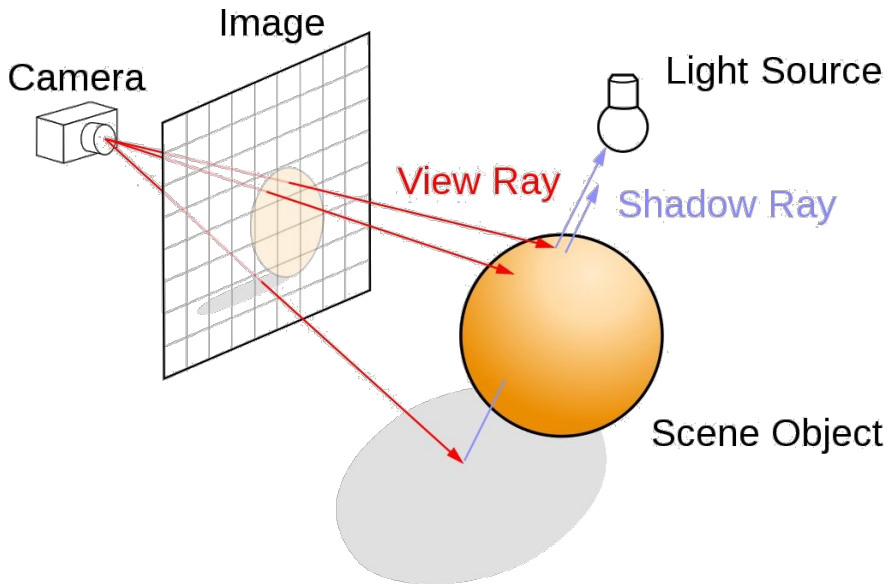


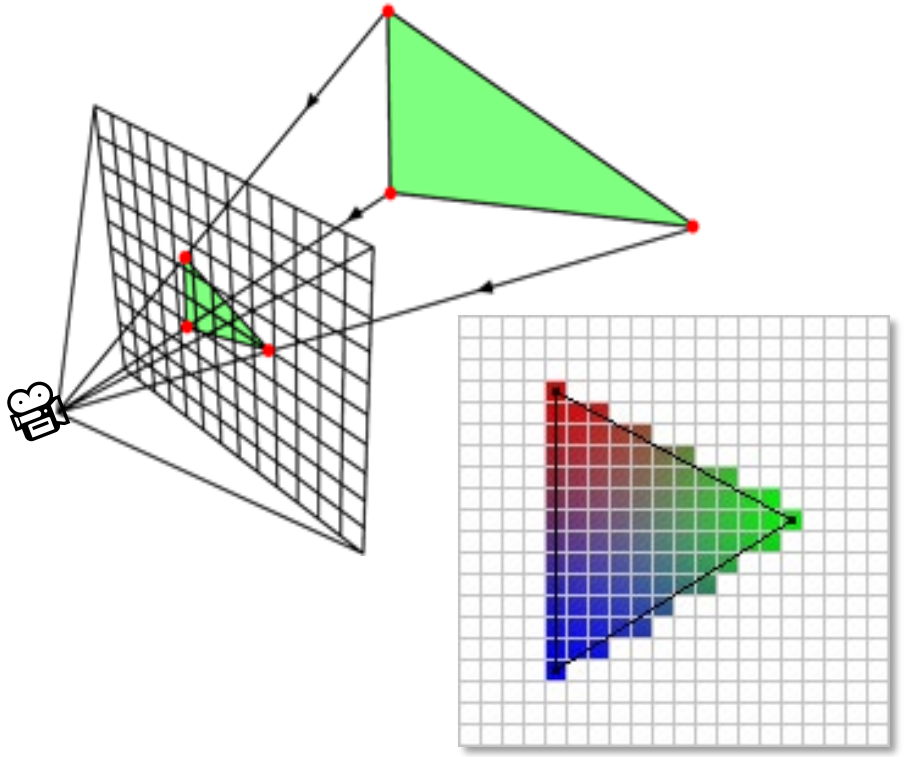
- Games utilize features of modern GPUs
  - Speeds up the rendering process
  - Allows for real-time rendering
  - “Interactive computer graphics”

Rendering techniques

- Raytracing
  - Calculate rays through screen pixels
  - Perform ray/shape intersections
  - Use surface info from closest hit
  - Can recursively trace new rays

- Rasterization
  - Project vertices onto screen pixels
  - Determine pixels within triangle's bounds
  - Interpolate vertex data for each pixel





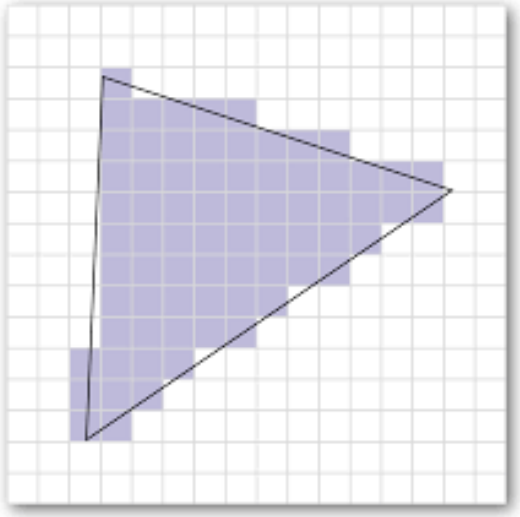
Raytracing vs. rasterization

- Both can produce high quality images

- Raytracing (slower)
  - Has access to entire scene
  - Inherently handles reflections

- Rasterization (faster)
  - One object at a time
  - Only has surface information
  - (Additional techniques can improve results)

For speed, rasterization wins



- Generally, rasterization is faster than ray tracing
- GPUs have long been built for rasterization
  - Contain specialized raster hardware
  - Can rasterize many, many triangles quickly
- All modern graphics APIs handle rasterization
- Until recently, every game out there was rasterized

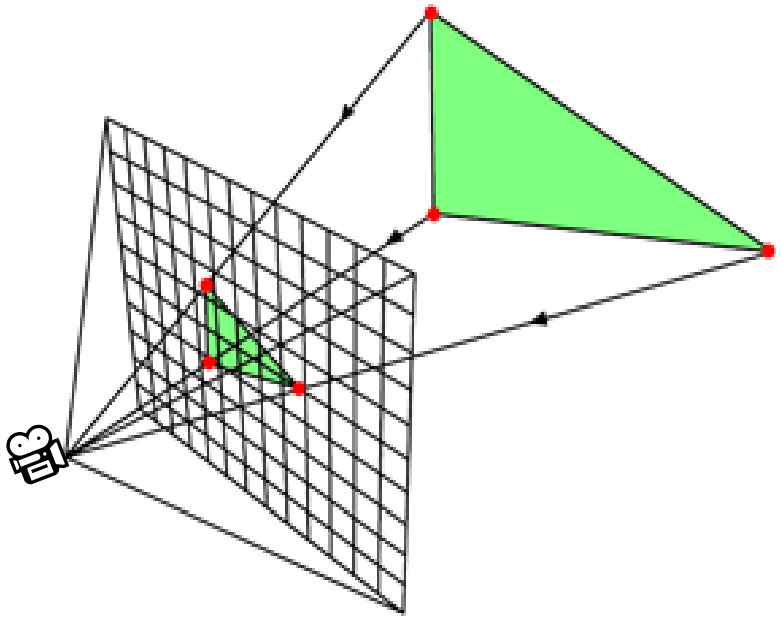
Sidenote: Real-time raytracing?

- Hardware accelerated raytracing
- Really impressive
  - But not a silver bullet
  - Still expensive in terms of performance
  - Often used in conjunction with rasterization
- Requires a pretty deep understanding of a low-level API
  - Such as DirectX 12 or Vulkan
  - And beefy hardware

So is rasterization bad? Not at all

Okay, rasterization it is. How is it done?

- Take one or more triangles
  - Vertices contain surface information
  - Color, surface normal, texture coords, etc.
- Project vertices onto a plane
  - The image we're creating
  - Also the near plane of our 3D camera
  - Need some matrix math here
- Determine which pixels are covered
- Interpolate triangle data and use it to color the pixels



General pseudocode of rasterization

- Foreach triangle
  - Project 3 vertices onto 2D plane
- Foreach projected triangle
  - Rasterize into pixels
- Foreach rasterized pixel
  - Determine color
  - Textures, lighting equations, etc.
- Lots of O(n) loops here 😬

- What if triangles share vertices?
- Maybe we can denote unique vertices?

- What if two triangles overlap?
- Like the front and back of a sphere

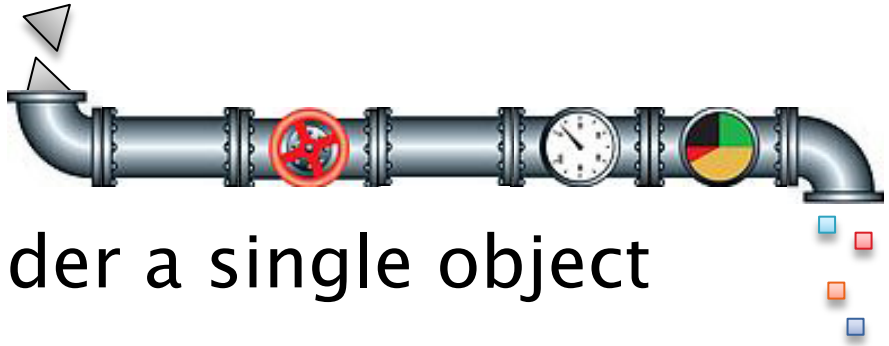
- Should this color replace other colors we've already rendered?
- What about transparency?
- What if we've already rendered something closer than this object?

Let's label these steps

- 1. **Input organization** → handle duplicate vertices, etc.
- 2. **Vertex transformation** → project vertices onto plane
- 3. **Rasterization** → determine which pixels are covered
- 4. **Pixel coloring** → what color is the surface at this pixel?
- 5. **Merge** → Combine colors based on occlusion, blending, etc.

- This is known as a "rendering pipeline"
  - Input is triangles
  - Output is pixel colors

Rendering pipeline



- The set of steps to go through to render a single object

- What is “a single object”?
  - One distinct mesh of triangles
  - With a single surface material

- What if I want to render multiple objects?
  - Do it again! (mesh → rasterize → pixels)
  - And again (mesh → rasterize → pixels)
  - And again (mesh → rasterize → pixels)

- Sidenote: Does your 3D model use two separate textures?
  - Those are *two different materials*
  - Must split the model and render each independently

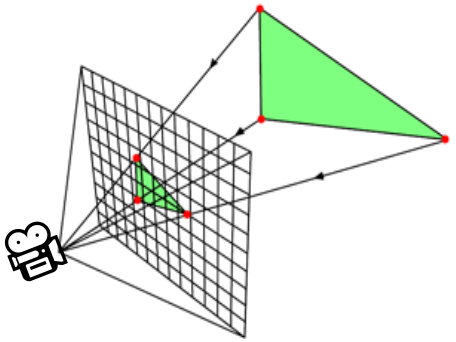
Speeding up a rendering pipeline

- Must go through all 5 steps *in order*
  - How can we speed them up?
  - Either do less work or get the work done faster
- It's a bunch of loops, right?
  - Foreach triangle…
  - Foreach pixel…
  - Etc.
- Can we do the work of each loop in parallel?
  - Process ALL triangles at once
  - Then process ALL pixels at once

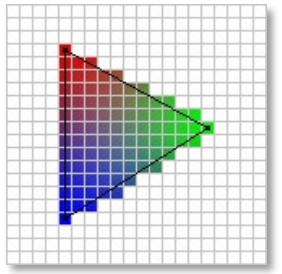
GPUs to the rescue

- Each pipeline step can be parallelized on the GPU
  - Built for highly parallel number crunching
  - And primitive rasterization (points, lines & triangles)

- For example:
  - Each vertex can be transformed simultaneously
  - Vertex A's transformation does not affect Vertex B's



  - Each pixel can be colored simultaneously
  - Color of Pixel A does not affect color of Pixel B



Sidenote: Doing less work?

- Using a GPU is obviously a big win
- But doing less work overall is also a great optimization
  - **Frustum culling** → don't draw what the camera can't see
  - **Batching** → draw multiple of the same thing at once
  - **Material IDs** → pre-process meshes and encode material IDs at each vertex, then dynamically index textures at runtime
- Advanced topics – for now, we'll just draw everything

How do we “use a GPU”?

- Directly accessing the hardware isn’t really feasible
- Using the drivers directly isn’t, either
  - Each driver is different
  - And has very specific requirements
- Need an Application Programming Interface (API) for graphics
  - A standardized way of using a GPU
  - That takes care of talking to the underlying hardware
  - And makes our lives (a little) easier

Graphics API

- System for utilizing graphics hardware for real-time rendering
- Several exist
  - DX11, DX12,
    OpenGL, Vulkan, Metal
  - Each is syntactically different

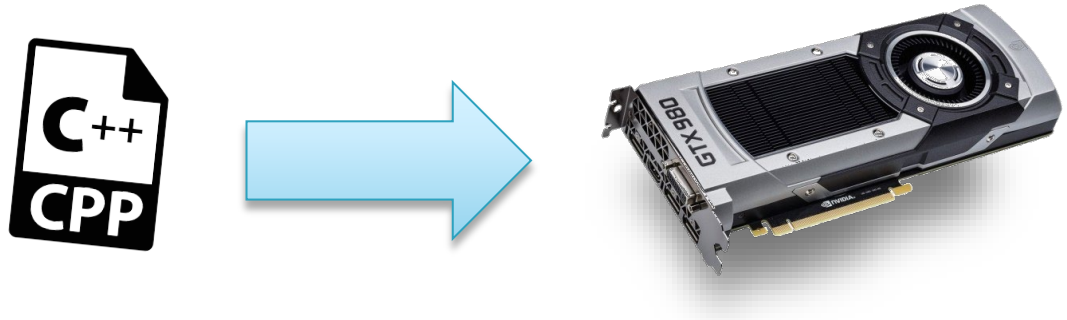
- Each has the same goal → Put something on the screen!
  - And mostly have the same features
  - Since they target the same hardware

What's a Graphics API do?

- They all have the same goal: Rendering
  - Putting something on the screen
  - Also called "drawing"

What's a Graphics API really?

- C++ code library
- Allows an application to utilize built-in features of our graphics hardware (GPUs) to greatly speed up rendering



- Technically…
  - Our C++ Code → Graphics API → Graphics Drivers → GPU Hardware

Graphics API organization

- All APIs have a similar high-level organization
- **GPU Resources** → Data in GPU memory used while rendering
- **Graphics State** → Options for rendering & active resources
- **Issuing Commands** → Changing state, start rendering, etc.
- **Rendering Pipeline** → Steps to actually render

Graphics APIs & GPUs

- Graphics APIs help us…
  - **Manage** the current rendering state (options)
  - **Store** data properly on the GPU
  - **Invoke** specific graphics functionality
  - **Synchronize** GPU with our application
  - And much more

- All of this goes into properly rendering
  - And rendering quickly
  - To maintain interactive framerates

Graphics API usage

- All APIs have similar overarching steps for usage
  - VASTLY differ in the details & syntax
- **Initialize API** → Turn it on, bind to a window
- **Create Resources** → Load models & textures, set up state objects
  - Done during start-up as these are slow!
  - Will have many resources in GPU memory at once
  - But only use a few at a time
- **Render one frame** → Part of game loop
  - Repeat!

Rendering a frame

- **Clear output** → reset to a solid color
- Foreach entity we want to render…
  - Set graphics state for this object
    - Change rendering options
    - Set active triangles
    - Set active textures
    - Set active shaders
  - Issue **draw( )** command → run rendering pipeline w/ active gfx state
- **Present** → show results of the frame to the user

Easy, right?

- Using a graphics API is easier than the alternative
  - Dealing with the drivers ourselves
- But they can be complex
- We're using DirectX 11 in this class
  - It's powerful
  - It's mature (many, many examples exist)
  - It's easier than lower-level APIs like DX12/Vulkan
  - We can learn the basics and do cool stuff with it quickly