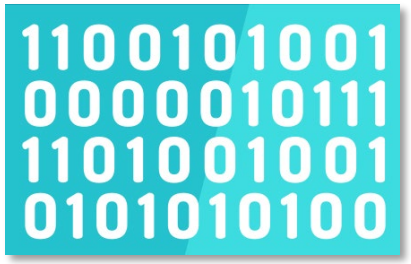
# Graphics APIs, Direct3D & The Rendering Pipeline

# Terminology




- **Buffer** – A place to store a "chunk" of data
  - Generally holds sets of numbers

- **CPU-Side**
  - The code running on your CPU *or* the data in system RAM
  - Your C++ code and what it directly interacts with

- **GPU-Side**
  - Video card hardware, video RAM or shader code
  - Anything that happens on the video card
  - Many Direct3D constructs (buffers, textures, etc) are GPU-side

# Terminology – Geometry

- **Vertex** – A single point
  - Often part of a larger geometric primitive
  - Can contain data in addition to a position in space
- **Geometric Primitive** – Simple geometric shapes
  - Points, lines & triangles
  - This is what the GPU draws
- **Mesh** – A unique set of geometry
  - Vertices and potentially indices
  - No transformation information
  - Can be shared among render-able entities

# Terminology – Output buffers

- **Texture** – An image on the GPU
  - A “buffer of pixels”
- **Render Target**
  - A texture on the GPU that can hold rendering output
  - Any number of these can exist
- **Back Buffer** – Specific render target used for screen output
- **Depth Buffer** – Texture holding depth of each pixel rendered
  - Another “buffer of pixels”
  - These pixels hold a single depth value (not a color)
  - Allows us to render 3D objects in any order

# Terminology – Rendering

- **Rendering Pipeline**
  - Steps to render triangles
  - The actual rendering process
  - Performed by Direct3D/drivers/GPU

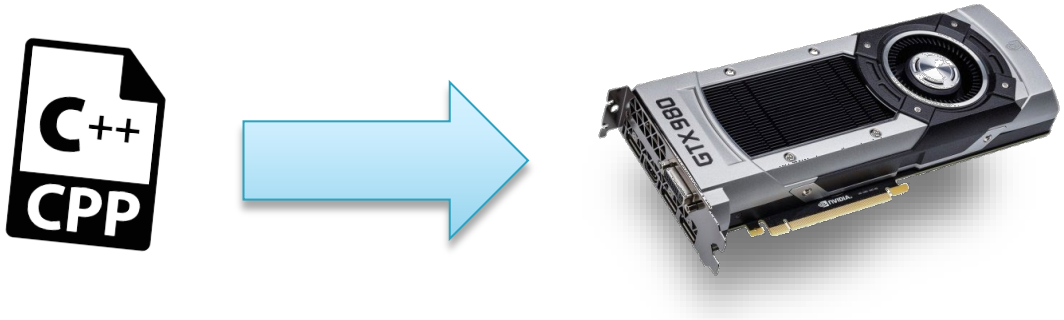
- **Fixed-Function Pipeline** – All stages are hardwired
  - You choose pre-defined options
  - Can't execute arbitrary code (no shaders)

- **Programmable Pipeline** –Supports executing arbitrary code
  - Certain steps require custom code (shaders)
  - Allows for far more variety and power

# Graphics APIs

# What's a Graphics API?

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




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

# Examples of Graphics APIs





# Direct3D 11

- The Graphics API we'll be using this semester
  - Powerful
  - Higher-level
  - Many examples exist
  - Integrates well with our tools
- Created by Microsoft
- Used on PCs and Xbox
- One of several APIs in "DirectX"

# Direct3D

# DirectX vs D3D

- **DirectX**: Collection of APIs allowing “direct” hardware access
- **Direct3D**: Specific API for rendering with a GPU
- The two terms are often used interchangeably

# Direct3D is…

- A library to interface with a GPU for rendering
- Not explicitly a *game engine* or a *game framework*
  - Unity is a game engine
  - MonoGame is a game framework
  - Both can use Direct3D under the hood
- Anything else we want, we must build ourselves!
  - Game loop?  Window?
  - We need to code those!

# Official helper libraries

- DirectX Math
  - CPU (C++) side, SIMD-friendly math library
  - Has structures/functions for 2D & 3D math
  - Comes with DirectX / Windows SDK
- DirectX Toolkit
  - Collection of D3D helper classes for common tasks
  - Created by Microsoft
  - Available on GitHub & NuGet
- More on these soon!

# Components of Direct3D

And most other graphics APIs

# D3D is object-oriented

- All functions are accessed through objects
  - Very few global functions
  - OpenGL/Vulkan differ here!

- Main Direct3D C++ objects:
  - **Device**
  - **Device Context** (“Context” for short)
  - **Swap Chain**

# Major components (of any graphics API)

- **Resources** – Data used during rendering
- **Graphics state** – Options for rendering
- **Issuing commands** – Change state, update resources, draw!

# Resources


- Data used during the rendering process
  - Input buffers (vertices)
  - Shader variable data
  - Textures
- Resources live in GPU memory
- Graphics API creates and manages resources
  - D3D's **Device** object creates resources
- Our job is to *bind* (activate) resources prior to each draw call

# Binding Resources

- We could have 100's or 1000's of resources in GPU memory
  - 250 vertex buffers
  - 122 index buffers
  - 7 shader variable buffers
  - 46 textures
- Which one(s) should be used for the next render?
- We specify the set of "active" resources by *binding* them
  - Call a function to make a resource active
  - Unbinds previously active resource

# Graphics state

- The current set of all graphics options and bound resources
  - Used by the API/Drivers/GPU when drawing
  - D3D's **Context** object tracks graphics state

- Examples:
  - Which vertex buffer is active?
  - Which shaders are active?
  - What are the rasterizer options?
  - Where should rendering output be stored?

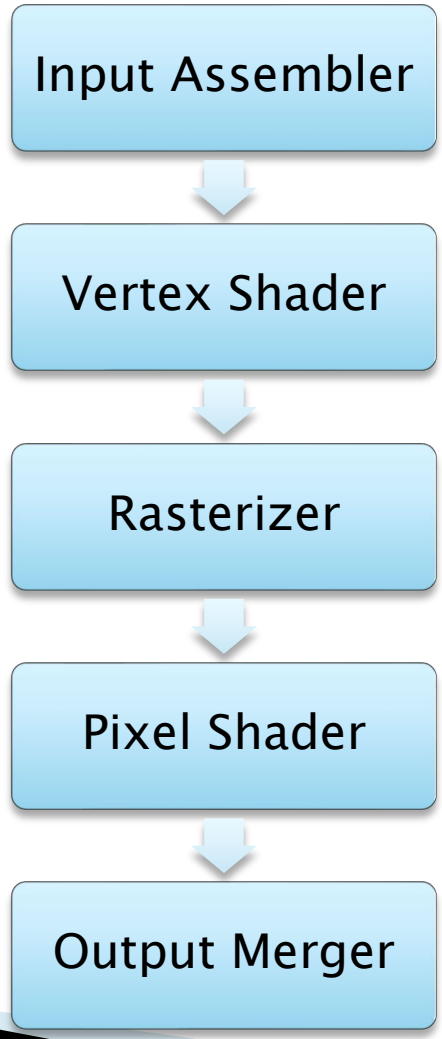
- All state persists until we change it

# Issuing commands

- D3D's **Context** object has two broad categories of functions
- Functions to "prepare for rendering"
  - Change state
  - Update resources
- Functions to "launch GPU work"
  - Clear output buffers
  - Begin rendering process ("draw calls")

# Rendering Pipeline

# Rendering pipeline stages

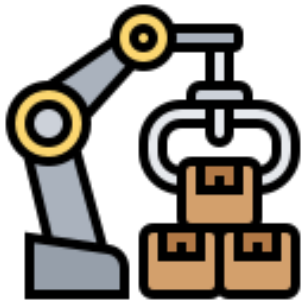



- Gets vertices from memory
- Runs Vertex Shader code for each vertex
- Turns triangles into pixels
- Runs Pixel Shader code for each pixel
- Puts pixels into a render target

# Pipeline I/O

- Input
  - Vertices (required) & indices (optional)
  - Vertices & indices must be on GPU already
- Output
  - Pixel colors in a render target (an image)
  - Default target is the “back buffer” (screen)
- Multi-step process
  - Once started, all steps performed by Direct3D, drivers & GPU
  - Once started, we **do not** get results back in C++!

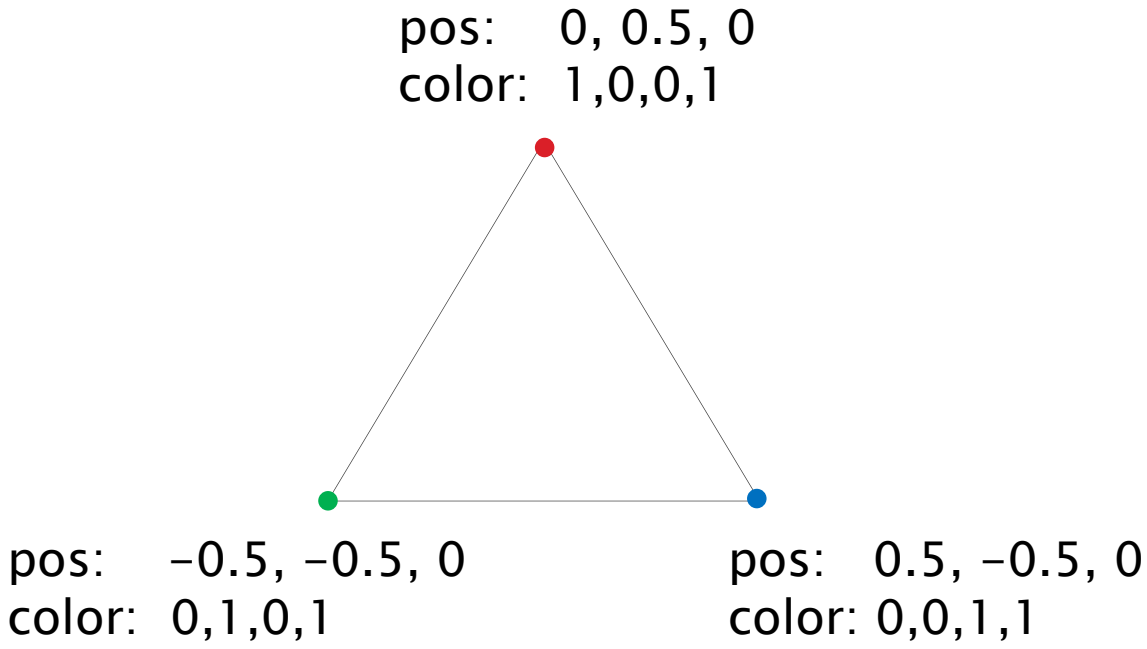
# Stage 1: Input Assembler



- Several tasks
  - Gathers information from the buffers
  - Assembles data into geometric primitives
  - Feeds primitives into the next stage
- Essentially: launches vert shader once per vertex
  - Each of which is processed in parallel (at the same time)
- Options for this stage include:
  - **Input layout**
  - **Primitive topology**

# Data after Input Assembler

- Vertices assembled into primitives (triangles)

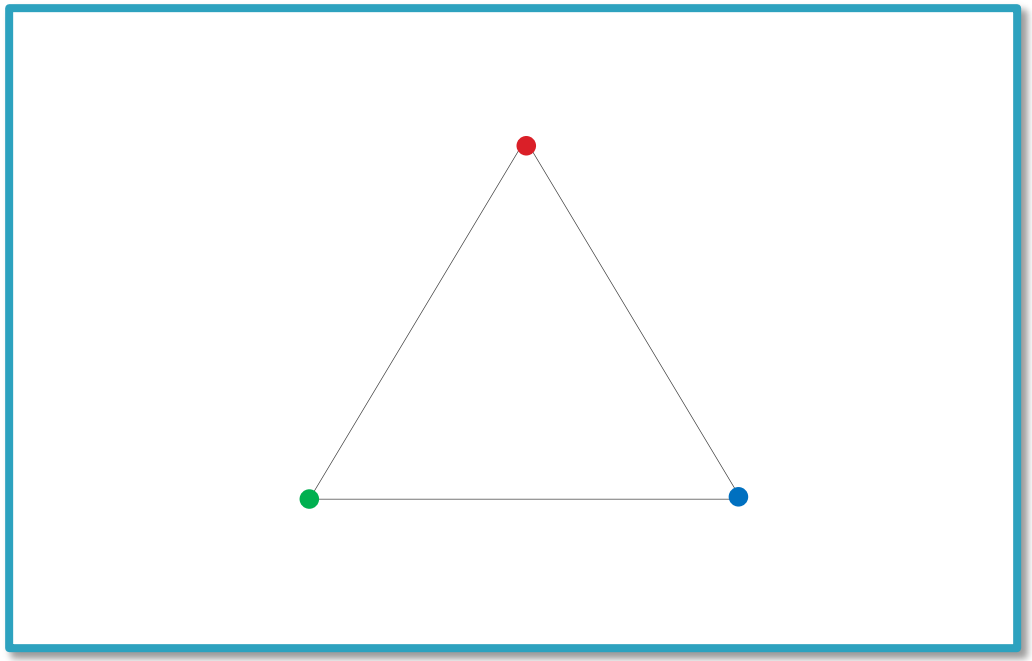


# Stage 2: Vertex Shader

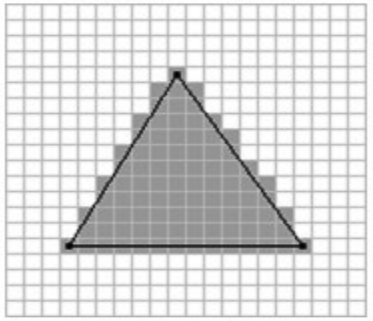
- Programmable and required
- Performs operations on a single vertex
  - Transformations, lighting, skinning (animation), etc.
  - Every vertex goes through this stage
- Main goal is to transform vertex positions:
  - Local to world space
  - World to view space
  - View to screen space

# Data after Vertex Shader

- Triangle is now in screen space
  - Exact location depends on transformation
  - And potentially camera matrices, too



# Stage 3: Rasterizer



- Converts primitives to pixels
  - Feeds those pixels into the next stage
  - Also clips primitives that are not visible

- Interpolates vertex data into per-pixel data
  - UV coords, colors, normals, etc.
  - Takes data at the vertices and calculates an adjusted value for each pixel of the triangle

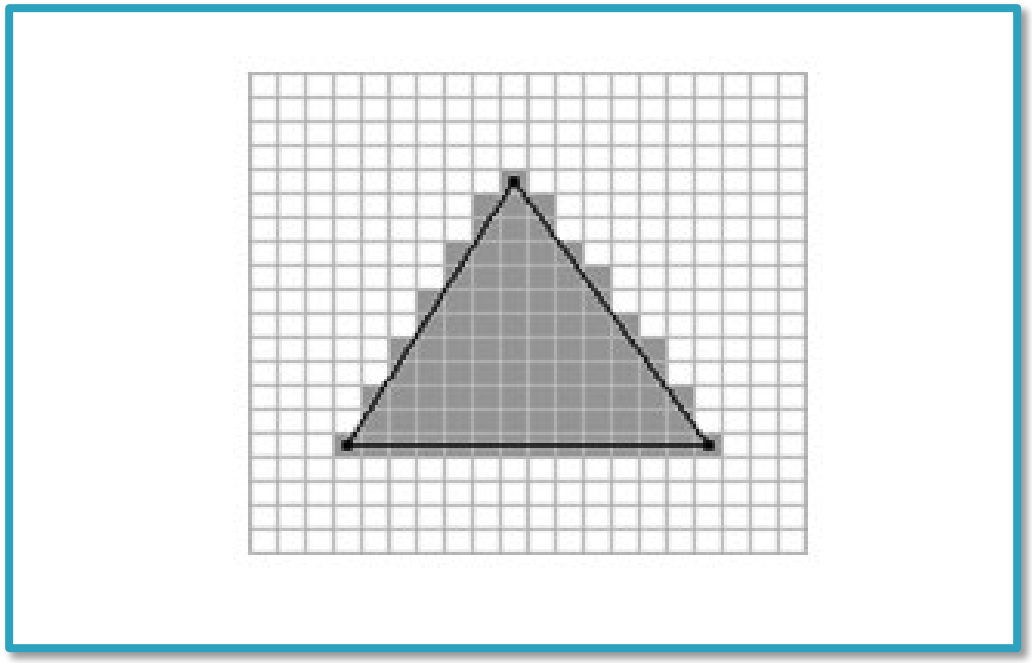
# Rasterizer

- Automatically culls back-facing triangles
  - Triangles you “can’t see” are skipped
  - Optimization – assumes most models are “shells”
  - Can be customized

- Options for this stage include:
  - **Viewport** for rendering/clipping
  - **Fill mode** – wireframe or solid
  - **Cull mode** – triangle front, back or neither

# Data after Rasterizer

- Triangle now rasterized

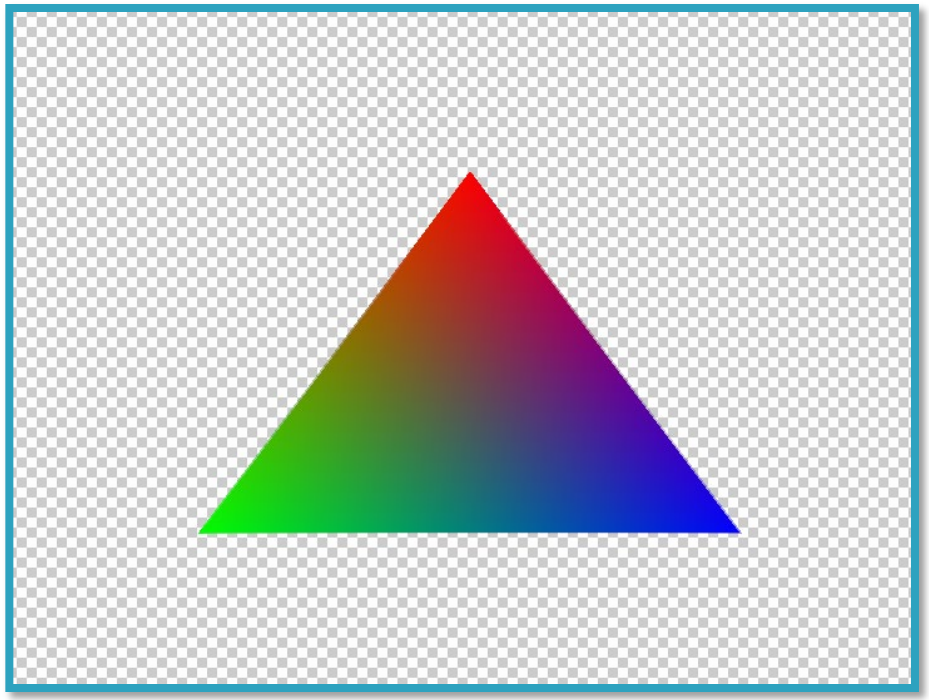


# Stage 4: Pixel Shader

- Programmable and required
- Calculates final pixel color
- Where most special effects are calculated
  - Lighting, normal mapping, textures, etc.
- Return a color (4 numbers): R, G, B, A

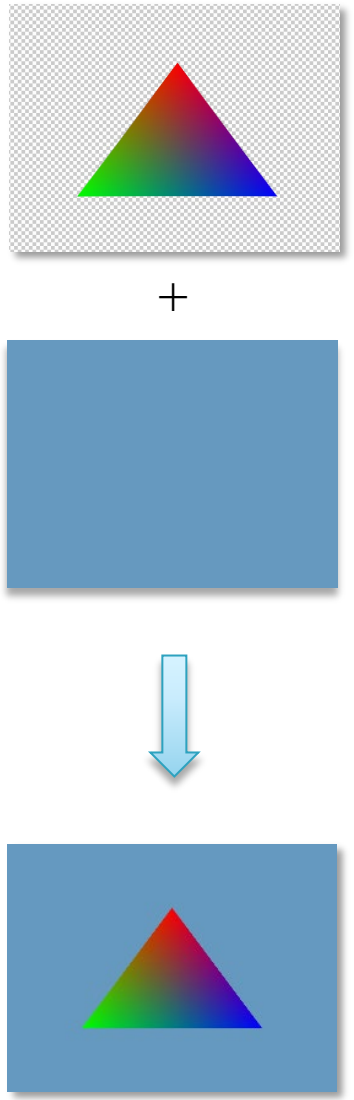
# Data after Pixel Shader

- Just this triangle's pixels have been "shaded"



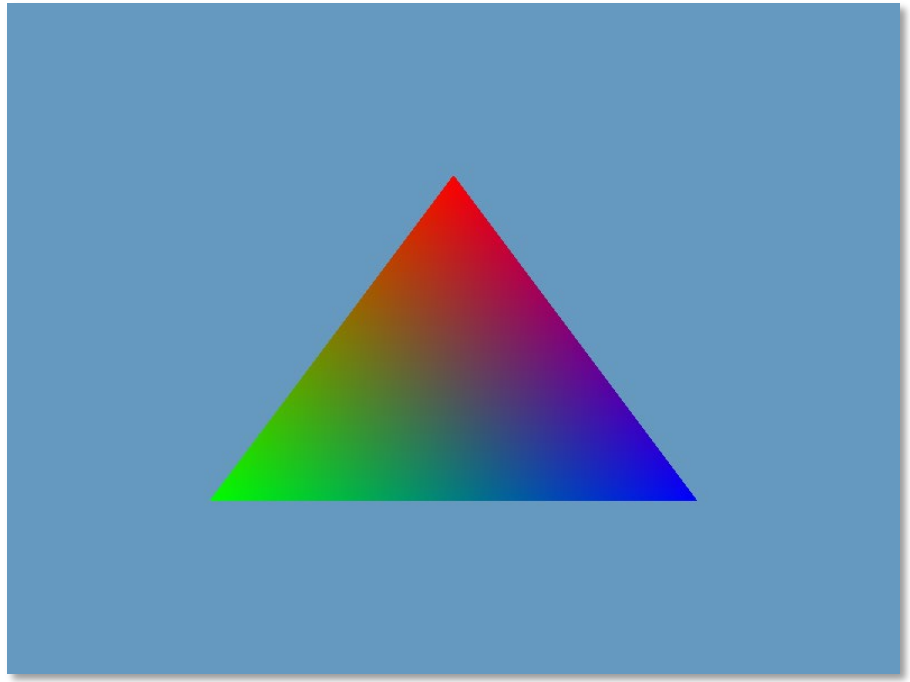
# Stage 5: Output Merger

- Merges output data (pixels)
  - Handles alpha blending if necessary
  - Combines colors together based on settings
- Writes to depth buffer
  - Records “depth” (distance) of each pixel
  - Can discard pixels that are “behind” others
- Options for this stage include:
  - **Blending options** – How to combine colors
  - **Render targets** – Where to put pixels
  - **Depth buffer** – Can turn read/write on and off

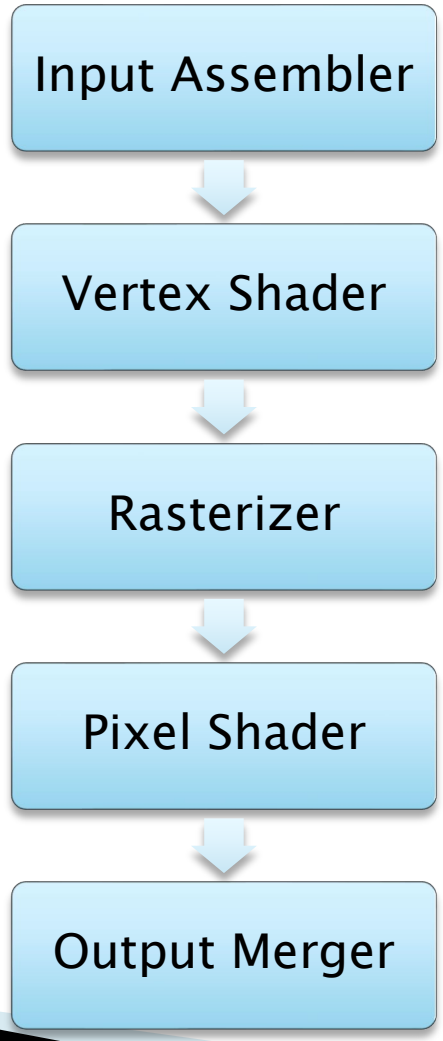


# Data after Output Merger

- Pixels merged with existing background



# Pipeline stages – Recap




- Reads from input buffers (vertex & index)
- Arranges vertices into primitives
- Supplies vertex data to the pipeline

- Processes each vertex of current draw
- Vertices processed in parallel (at the same time)
- Passes transformed vertices to next stage

- Determines which pixels are covered by triangles
- Calculates per-pixel data from per-vertex data

- Processes each pixel for current draw
- Pixels processed in parallel (at the same time)
- Passes pixel colors to next stage

- Merges pixel shader results with previous color output
- Handles blending (if enabled)
- Writes depth data to depth buffer

# Other Pipeline Details

# Rendering Pipeline is asynchronous

- Draw calls are not blocking!
  - C++ does *not* wait for a draw call to complete
  - Pipeline starts
  - Takes time to complete
  - Our code continues queueing up commands for Direct3D
- Common to call Draw() many times before the first is finished
  - Rendering is usually ½ - 1 frame behind game loop
- Part of Direct3D's job is to handle that synchronization for us

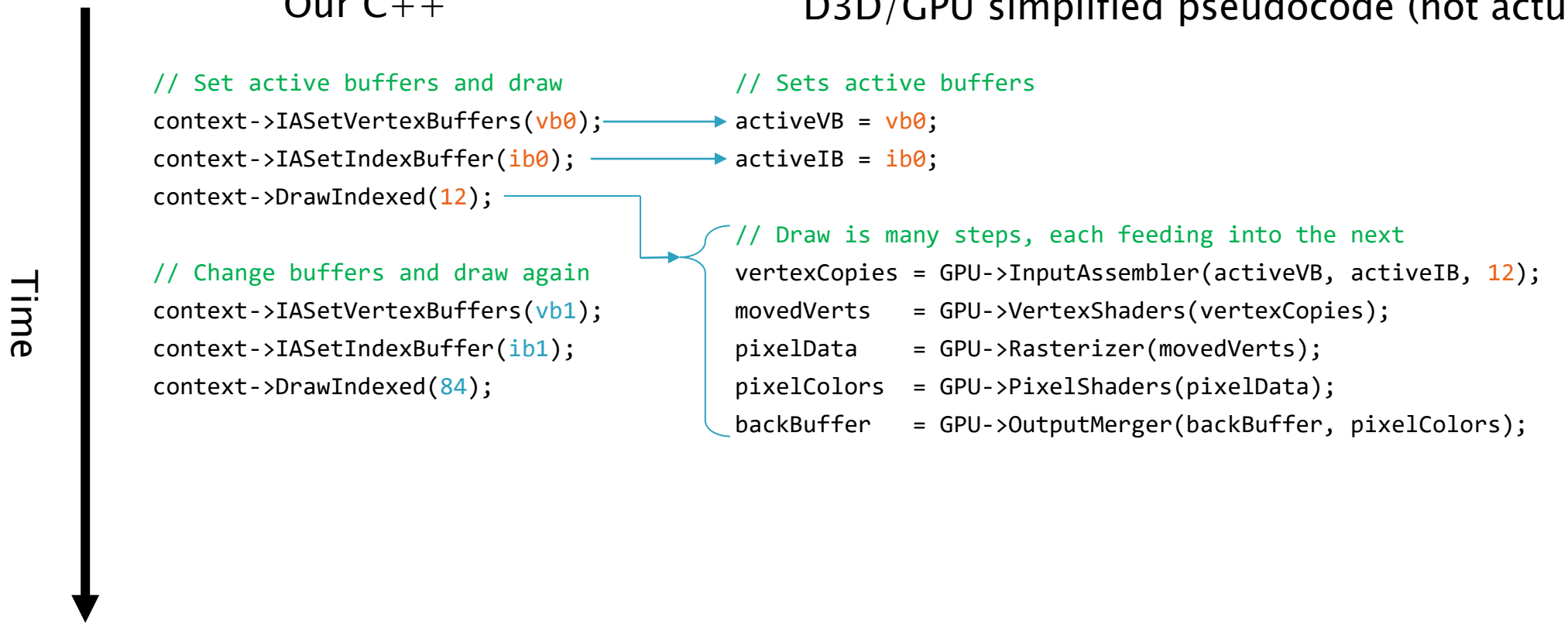
# CPU-Side vs. GPU-Side

Our C++

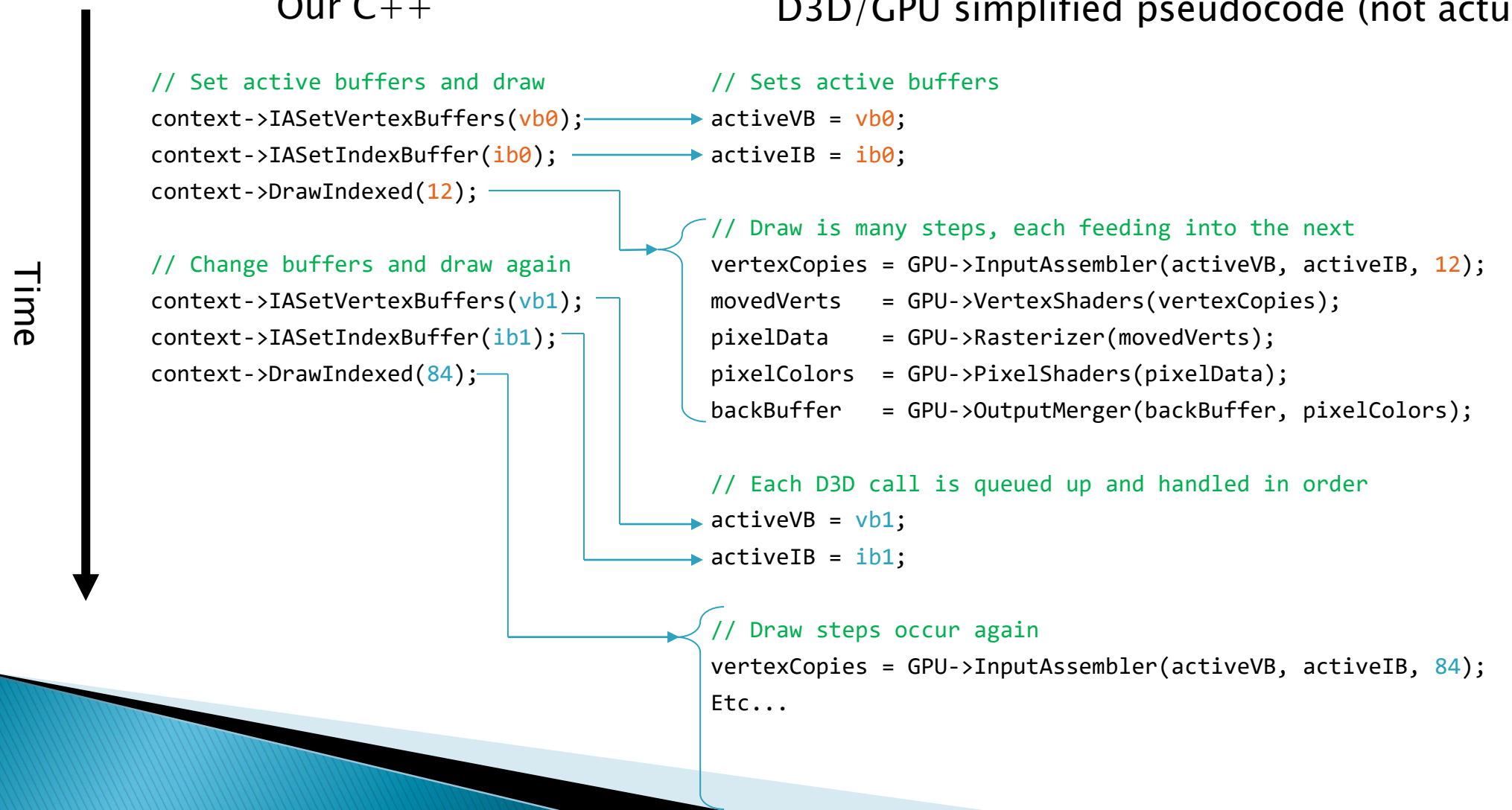

```
// Set active buffers and draw
context->IASetVertexBuffers(vb0);
context->IASetIndexBuffer(ib0);
context->DrawIndexed(12);

// Change buffers and draw again
context->IASetVertexBuffers(vb1);
context->IASetIndexBuffer(ib1);
context->DrawIndexed(84);
```

# CPU-Side vs. GPU-Side



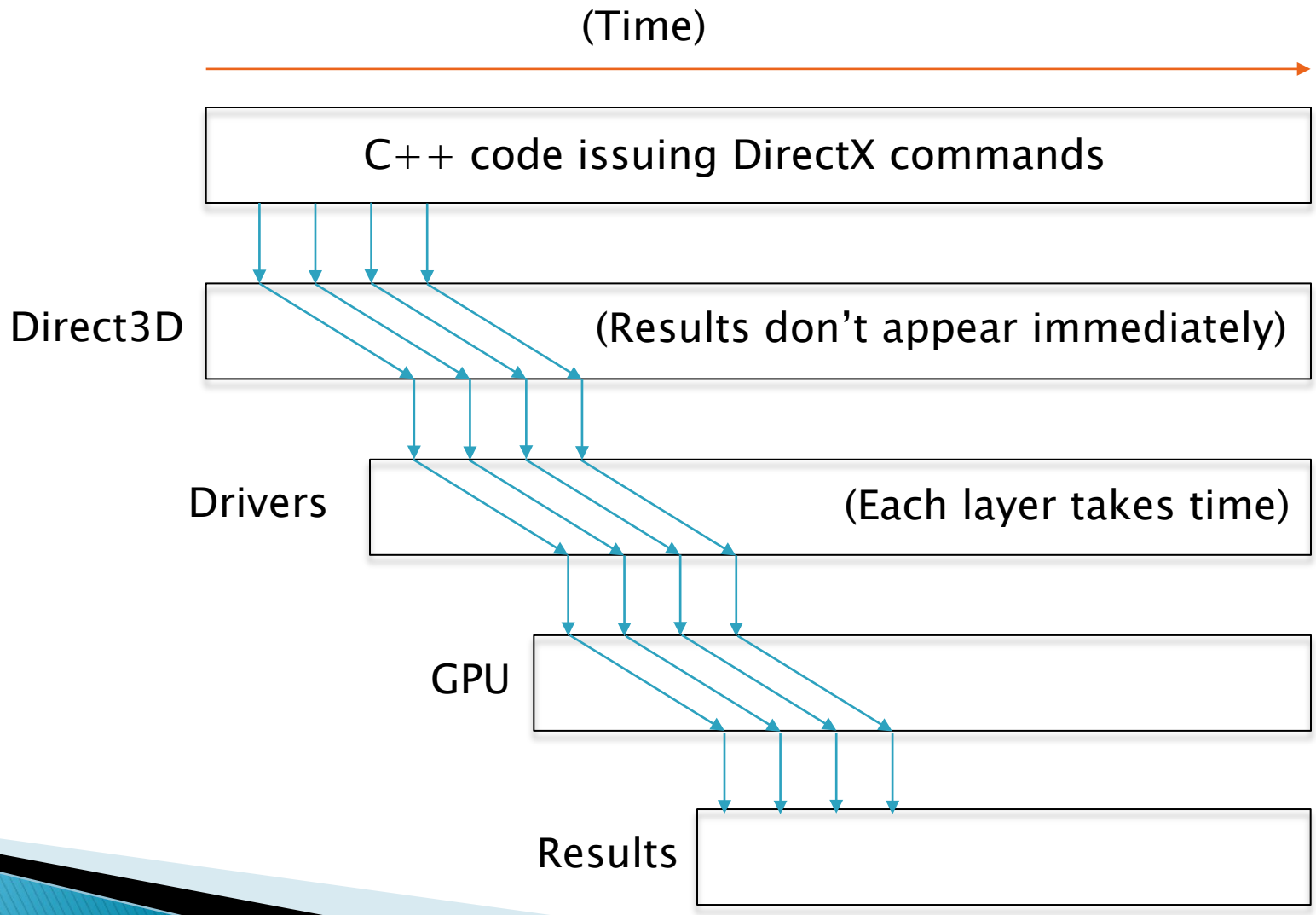
# CPU-Side vs. GPU-Side



# One-way pipeline

- Pipeline is optimized in ONE DIRECTION: CPU → GPU
- After a frame is rendered & displayed, it is **thrown out**
  - Clear the frame
  - Replace with the next one
- Attempting to get *any* results back causes a **stall**
  - All CPU code must WAIT for GPU to catch up
  - Framerate drops **dramatically**

# One-way pipeline example



# CPU readback = Pipeline stall