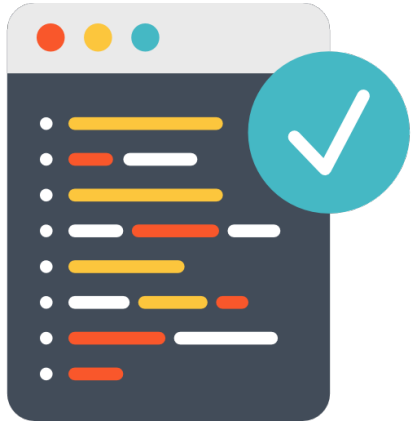
Introduction to Shaders

In Direct3D 11

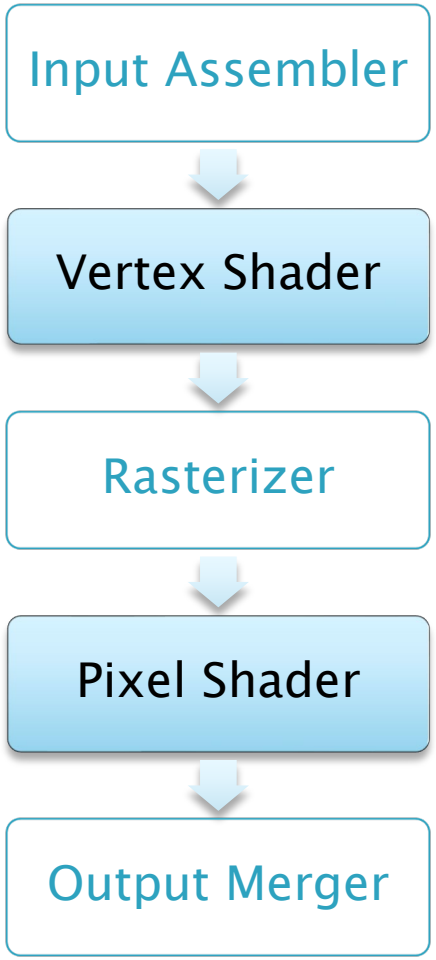
Shaders



- Small programs executed by the GPU
  - During the rendering pipeline
  - Need to draw something to run a shader!
- Written in HLSL (for Direct3D)
  - High-Level Shading Language
- The core of a “programmable pipeline”
  - Required by the pipeline
  - Can’t draw if you don’t have shaders!

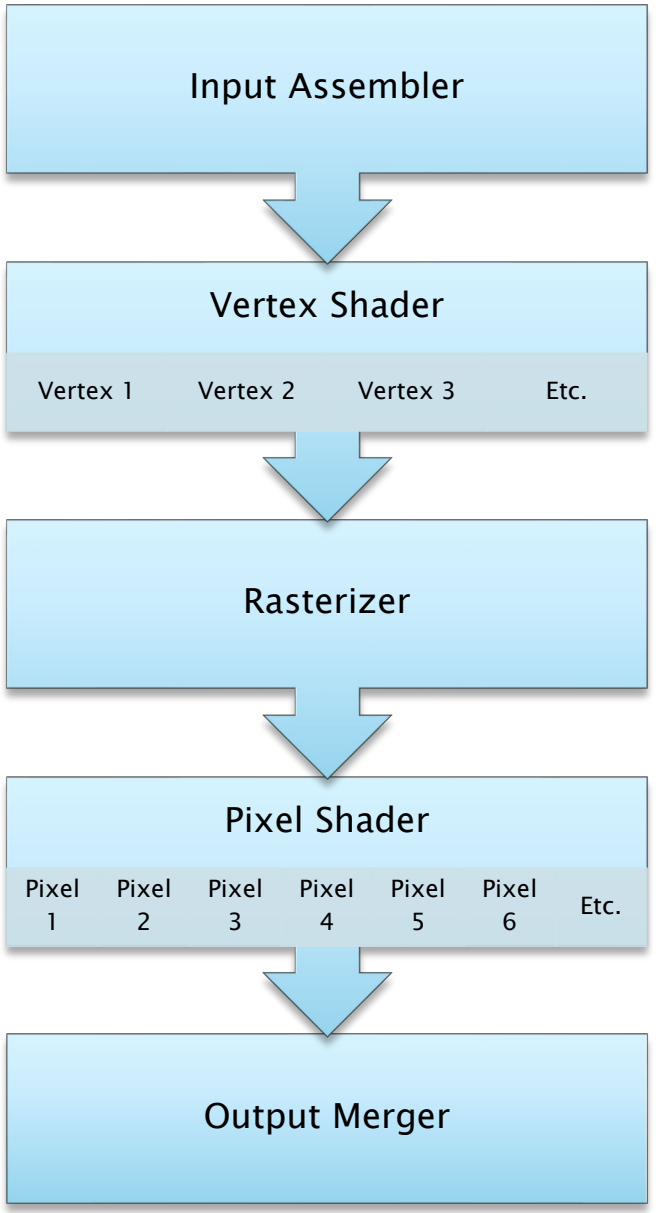
Shaders & the pipeline

- Two required shader stages during pipeline
  - Vertex Shader
  - Pixel Shader
- Shaders receive data from previous stages
  - The shader's *input*
- And must send data to the next stage
  - The shader's *output*



Shaders & parallelism

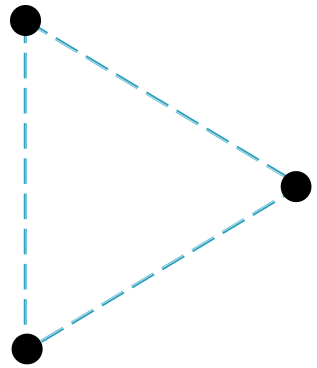
- Each shader stage represents multiple *shader instances* running in parallel
  - Copies of the shader program
- All shader instances run in lock-step
  - All execute the same instruction at once (SIMD!)
  - Each instance is fed different input
  - Allowing instances to produce different output
- All instances must finish before next stage



Direct3D shader types

- **Vertex** and **pixel** shaders required for basic rendering
- Several other varieties
  - Hull
  - Domain
  - Geometry
  - Compute (not part of the pipeline)
- We won't cover these much this semester
  - Might have a neat demo of compute shaders later on

Vertex shader

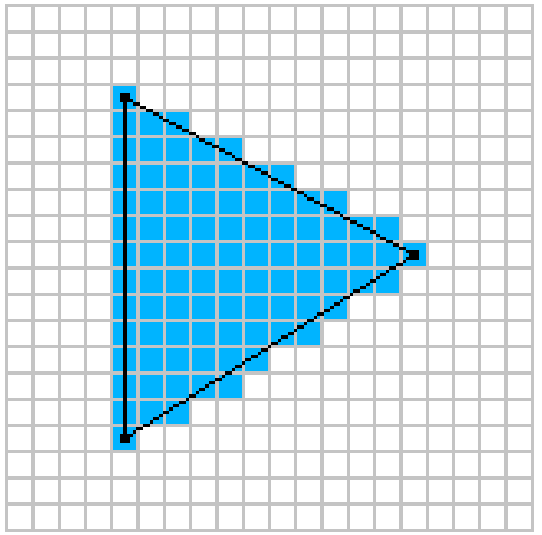


- **Main goal**: Transform local vertex position to screen-space position

- **Inherent Input**: All data related to a single vertex
  - Each instance gets a different vertex from bound vertex buffer
  - From input assembler stage

- **Minimum output**: 4-component screen-space vector
  - Must be tagged as “SV_POSITION”
  - Can also output any other per-vertex data

Pixel shader



- **Main goal**: Determine the color of this pixel
- **Inherent Input**: Interpolated vertex output data
  - Each instance gets slightly different data
  - From rasterizer stage
- **Minimum output**: 4-component color vector (RGBA)
  - Must be tagged as “SV_TARGET”

Shaders – Details & Syntax

HLSL: High-Level Shading Language

- HLSL is the shading language for Direct3D
- C-like language
  - Originally, shaders were written in pure assembly
  - These days: compiled from HLSL to assembly
- Need to be on the GPU before execution
- Generally, individual shaders written for specific effects
  - Rather than an *all-in-one mega shader*

“Shader model” = Version

- The “version” of HLSL you’re using
  - Direct3D 11’s shader model is 5.0
- Higher shader models generally have:
  - More features
  - Higher instruction limits
- Can compile shaders to older models
  - For backwards compatibility
  - Purposefully limits language features

Common programming constructs

- HLSL has:
  - Local variables (defined in a function)
  - Global variables (from constant buffers)
  - Functions
  - Structs
  - Loops
  - If statements
  - Switch statements

- Note: Branching *may* negatively impact performance!
  - Worst case when different instances take different branches

Common data types, operators & functions

- Common data types
  - Scalar: int, bool, float
  - Vector: float2, float3, float4
  - Matrix: float3x3, float4x4 (matrix)
- Operators
  - Standard mathematical, relational, boolean & bitwise operators
- Intrinsic functions
  - Basic math and trig: abs(), pow(), min(), max(), sin(), cos(), etc.
  - Vector math: reflect(), refract(), dot(), cross()

Vector initialization examples

```
// Basic initialization
float4 color = float4(1.0f, 0.5f, 0.0f, 1.0f);
float2 two   = float2(5.5f, 6.2f);

// Initialize a larger vector with a smaller one
float4 large = float4(two, 1.0f, 1.0f);
float4 again = float4(1.0f, two, 1.0f);
float4 twice = float4(two, two);
```

Vector Components

- A vector's components may be accessed with:
  - r, g, b, a
  - x, y, z, w

- Accessing data in colors (or ANY vector):

```
float4 color      = float4(1,0,0,1);
float redAmount   = color.r;   // or color.x
float greenAmount = color.y;   // or color.g
```

Swizzling

- Syntactical shortcut for rearranging vector components

```
float4 color     = float4(1.0f, 0.5f, 0.1f, 0.0f);
float4 foSwizzle = color.xxyx;  // 1, 1, 0.5, 1
float4 worksFine = color.agbb;  // 0, 0.5, 0.1, 0.1
float4 error     = color.xbya;  // Can't mix notations
```

Swizzling – Automatic repetition

- If fewer components than necessary specified, last is repeated

- When a float4 is expected:
  - .xy really means .xyyy
  - .z really means .zzzz
  - .xxx really means .xxxx

- Code examples:

```
float4 color = float4(1, 0.5f, 0.25f, 0);
float4 allZ  = color.z;  // 0.25, 0.25, 0.25, 0.25
float4 someY = color.xy; //    1,  0.5,  0.5,  0.5
```

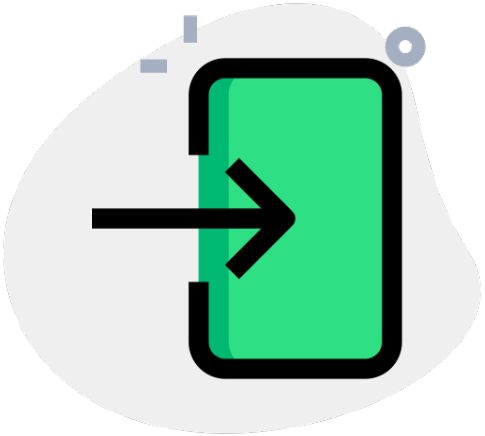
Functions

- C-like functions
  - Require return types (or void)
  - Can have parameters
  - Support *out* parameters
- Can write any number of functions
- Always inline'd by compiler
  - No function call overhead!
  - And no recursion

Shader entry point

- `main()` function
  - Where a shader's execution begins
- Must have a return type
  - Represents data to send to next stage of pipeline
  - Each type of shader has unique requirements for its return value(s)
- Can (and will) have parameters
  - Represents data from previous pipeline stage

Parameters to main()



- Represent input from previous pipeline stages
  - All shaders will have one or more params
- Option 1: Comma-separated variables

```
// Pixel shader with 2 parameters
float4 main(float2 uv : TEXCOORD, float4 c : COLOR) : SV_TARGET {…}
```

- Option 2: Custom struct with multiple members

```
// Pixel shader with 1 struct parameter
float4 main(VS_OUTPUT psInput) : SV_TARGET {…}
```

Return value from main()

- Shaders require a return type for output
  - Sends data to next pipeline stage

- Option 1: A single value

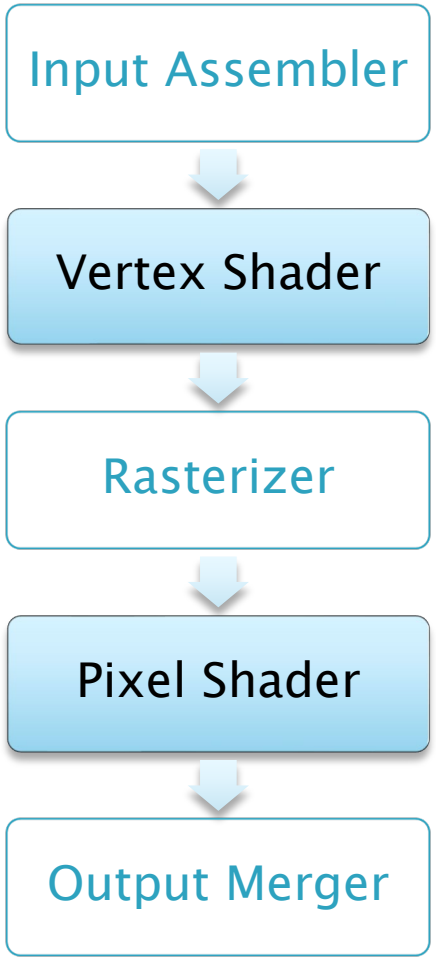
```
// Pixel shader returning one variable worth of data
float4 main(VS_OUTPUT psInput) : SV_TARGET {…}
```

- Option 2: Custom struct with multiple members

```
// Pixel shader returning a whole struct
PS_OUTPUT main(VS_OUTPUT psInput) {…}
```

Basic shader input & output

- Vertex shader
  - **Parameters**: Individual per-vertex data
  - **Must return**: Transformed per-vertex data
- Pixel shader
  - **Parameters**: Per-pixel interpolated VS output
  - **Must return**: Color (a single float4)
- VS output must match PS input
  - Can define common structs in external files
  - And use #include as necessary



Semantics

- Tags for shader input & output
  - Additional meta-data for the pipeline
  - Informs Direct3D how to process this data
  - Required on parameters & return types

```
struct VertexShaderInput
{
    float3 localPosition : POSITION;
};
```

Semantic

Required semantics

- Some semantics are **expected** and **required**
  - Begin with SV_ (“System Value”)
- Vertex shader
  - Output *must* include a float4 tagged as SV_POSITION
  - Rasterizer expects this data
- Pixel shader
  - Output *must* include a float4 tagged as SV_TARGET
  - Output Merger expects this data

Semantic names are flexible

- `float3 color : HORSE;`
  - Technically works
  - But not great – lacks actual meaning/intent
- Common conventions
  - POSITION, NORMAL, TANGENT, TEXCOORD, COLOR, etc.
  - These had meaning in Direct3D 9.0
- When in doubt: use a string that shows intent
  - CAMERAPOSITION is better than LOLWTFPLEASECOMPILE

Shader include files

- Can create files to be #include'd into shaders for organization
- Common extension is `.hlsli`
  - "HLSL Include"
  - But any extension works – it's all just text
- Must use C-style include guards (no "#pragma once" here)

```
#ifndef UNIQUE_NAME_FOR_THIS_FILE
#define UNIQUE_NAME_FOR_THIS_FILE
<all code goes here>
#endif
```

Global shader data from C++

- Data that is shared (constant) among all shader instances
- Constant buffers!
  - Global shader variables
  - Good old cbuffer definitions
- No semantics on cbuffer variables
  - This is data straight from GPU memory
  - Not “flowing through” pipeline stages

Resources & registers

- **Registers:** Predefined “slots” for resources

- Common resource types:
  - Samplers (*s* registers)
  - Textures (*t* registers)
  - Constant buffers (*b* registers)

```
cbuffer externalData : register( b0 )
{
	matrix worldMatrix;
};
```

Available shader data

- Data for a shader comes from one of three places
  - 1. Input from previous pipeline stage
  - 2. Constant buffers
  - 3. Texture resources
- In all cases, the data must be on the GPU before calling Draw()
- It's our job to put it there

Shader Possibilities

What *can* we do with shaders?

- Technically…anything we want!
  - It's all numbers
  - It's all math

- Perform lighting equations to mimic photo-realism
- Procedurally generate abstract art
- Create music visualizations

- https://www.shadertoy.com/ – WebGL shader playground!

What *will* we do with shaders?

- A game engine usually isn't generating abstract art in shaders
- Our goal:
  - Mimic surface materials
  - Under various lighting conditions
  - To create photo-realistic 3D worlds
- Plus a few other interesting effects

How many shaders do we need?

- Different effects generally have unique shaders
  - Static meshes
  - Animated meshes
  - Photo-realistic materials
  - Stylized materials (e.g. toon shading)
  - The skybox
  - Particle systems
  - Post processing passes (glow, depth of field, etc.)
  - User interface
- Each of these would use different shaders

Managing Shaders

Managing shaders

- Each shader gets its own .hlsl file
  - Visual Studio will compile your .hlsl files
  - And do syntax highlighting
  - And give you errors & line numbers if necessary

- While your application is running, you should:
  - Load pre-compiled shaders at start-up
  - Set (“activate”) individual shaders when necessary
  - Combine them in whatever ways make sense

Combining Shaders

- Shaders must be compatible to work together
- You can't just throw any two (or more) shaders together
- At a minimum, the following must match:
  - Output of your vertex shader
  - Input of your pixel shader

Drawing with Shaders

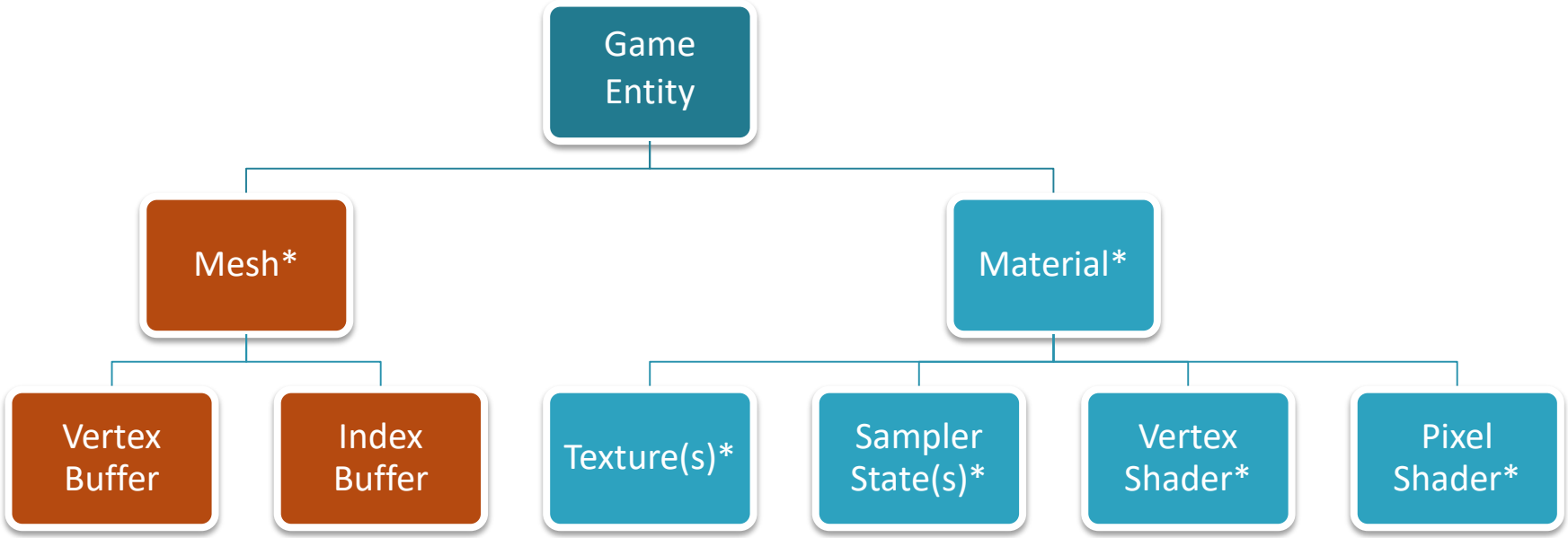
- When drawing, you **need**:
  - A vertex shader
  - A pixel shader

- You might also **want**:
  - One or more *textures*
  - One or more *samplers* (options for using textures)
  - External data (numbers) – colors, lights, time, etc.

Material System

- What's a material?
  - Combination of shaders, textures and other data
  - Describes the look of a mesh
  - Think of materials in Unity as an example
- Material definition should include shaders
- When "using" a material
  - Activate material's shaders
  - Draw one or more meshes

Basic game entity hierarchy



Example Shaders

From the Starter Code

Example shaders

- We'll look at the starter project shader code
  - VertexShader.hlsl – Vertex shader code file
  - PixelShader.hlsl – Pixel shader code file

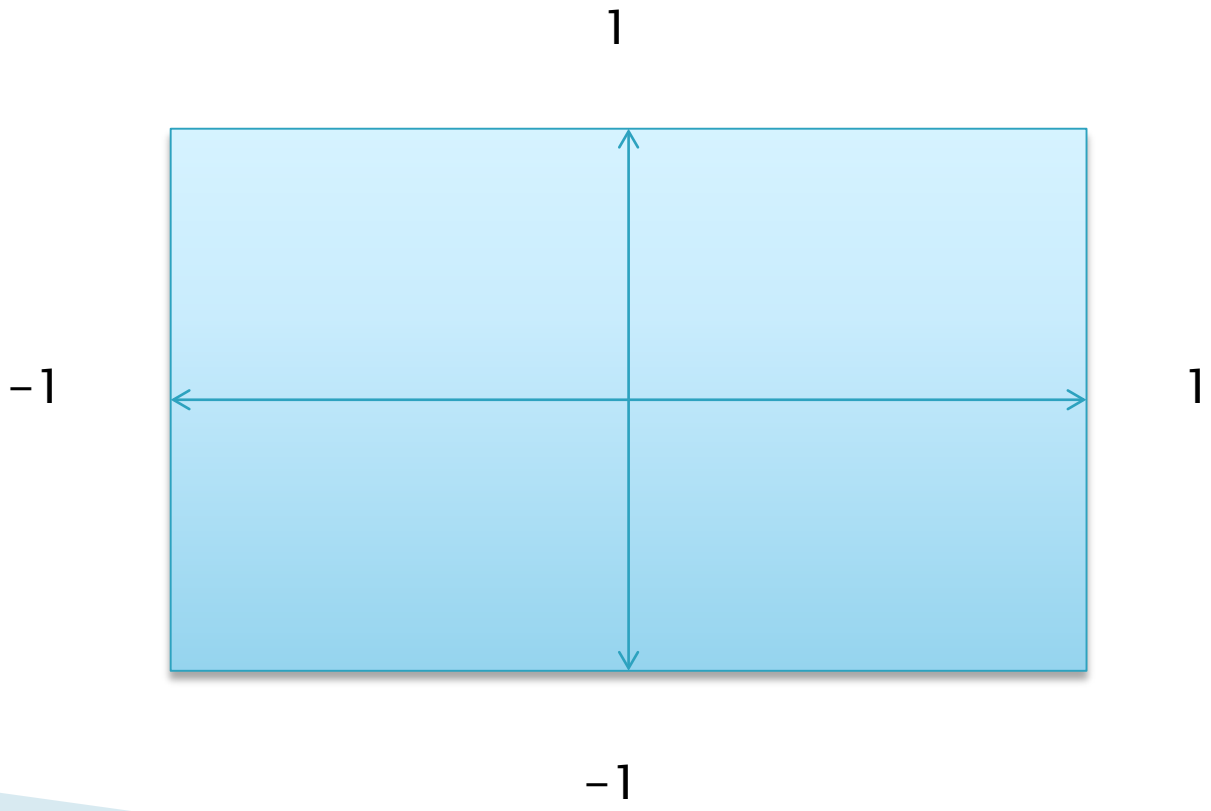
- Each is compiled by Visual Studio
  - At build time
  - Outputs .cso files

Vertex shader

- Performs per-vertex processing
- At a minimum:
  - Output vertex position in homogenous clip space
- Homogenous clip space
  - Normalized device coordinates
  - Resolution-agnostic coordinate system
  - -1 to 1 on each axis
- Can also output other data

Normalized device coordinates

- Vertex shader output screen coordinates:



VS: Constant buffer

- Constant buffer (cbuffer) holds external data
  - “Per model” data can be set from CPU-side

```
// Constant Buffer
cbuffer externalData : register( b0 )
{
	matrix world;
	matrix view;
	matrix projection;
};
```

Binds this variable to a specific constant buffer index

- **b** means Constant Buffer
- **0** is the index

- Built-in data type
- Same as float4x4

VS: Input struct

```
// Represents a single vertex worth of data
// - Must match vertex layout in your buffers
struct VertexShaderInput
{
    float3 localPosition   : POSITION;
    float4 color           : COLOR;
};
```

- Built-in data types
- Multi-component vectors

- These are called “semantics”
  - Define the usage of the variables
  - Some names have meaning
  - Others can be custom
- Can also have an index – Examples:
  - `COLOR0, TEXCOORD1, etc.`
- HLSL handles some types differently

VS: Output struct

- “VertexToPixel” – my own naming convention
  - Used as the VS output & PS input struct

```
// Represents data we send down the pipeline
// - Should match pixel shader’s input
struct VertexToPixel
{
  float4 screenPosition   : SV_POSITION; // SV = System Value (required)
  float4 color            : COLOR;
};
```

VS: Entry point

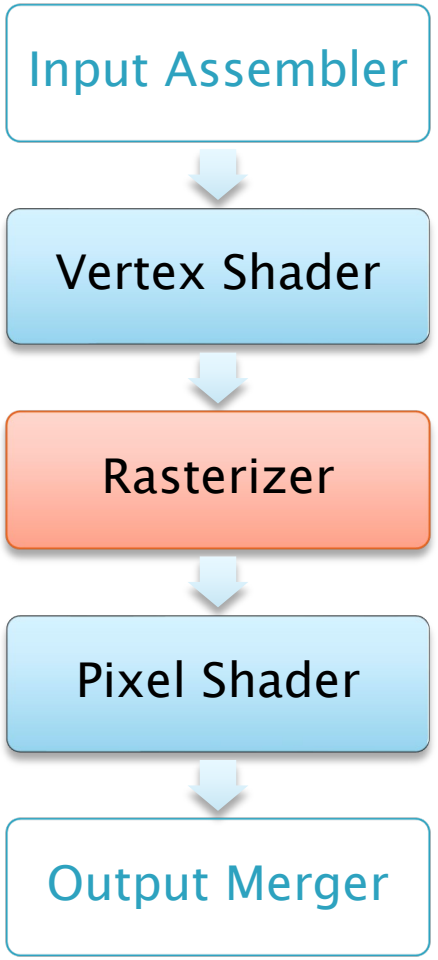
```
VertexToPixel main( VertexShaderInput input )
{
  // Create output variable
  VertexToPixel output;

  // Pass data through (in correct format)
  output.screenPosition = float4(input.localPosition, 1.0f);
  output.color = input.color;

  // Return entire struct (pass to rasterizer)
  return output;
}
```

Between VS and PS

- Rasterization occurs
- Rasterizer turns triangles into pixels
  - After the vertex shader is run on all vertices
- Rasterizer interpolates all VS output
  - Each pixel gets an altered version of VS output
  - Sometimes called “per-pixel” data



Pixel Shader

- Performs per-pixel processing
- Input struct must match VS output struct
- At a minimum, must output a color

Pixel Shader – Input Struct

- Could #include external file defining this struct

```
// Defines the input to this pixel shader
// - Represents data we expect to receive
//   from earlier pipeline stages
struct VertexToPixel
{
    float4 screenPosition       : SV_POSITION;
    float4 color                : COLOR;
};
```

PS – Actual Shader "Program"

This pixel shader returns a single color as a 4-component vector

This pixel shader's output data has a semantic: SV_TARGET

Results go to current render target

```
// Entry point for this pixel shader
float4 main(VertexToPixel input) : SV_TARGET
{
    return input.color;
}
```