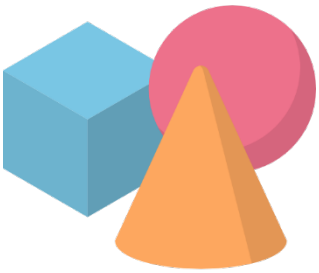
Graphics APIs

What is an API?

APIs, or Application Programming Interfaces, allow computer *applications* or hardware to communicate with each other. An API generally provides a public set of functions a programmer can use (when *programming*) to send data to, retrieve data from, or otherwise control another application or piece of hardware. This public set of functions is known as an *interface*, as it allows a programmer to use specific features without revealing all of the internal details. This is similar to how the interface of a real-world object – such as the dashboard of a car, or the keypad on an electronic lock – can hide the innerworkings of that object.

If you've ever used the C# game library *MonoGame* or created scripts for the *Unity game engine*, you've used an API. It's the set of functions you, the programmer, can call to make the underlying system do something useful. In both of these cases, you don't have unfettered access to all of the internal data of the system, but you can retrieve and alter some of that data and also tell the system to perform a pre-defined set of tasks.

What is a Graphics API?



As the name implies, a Graphics API allows a programmer to communicate with a graphics card (GPU) to perform common rendering tasks. In other words, it allows us to tell the GPU to draw things to the screen. However, as you might imagine, it's much more complicated than just calling a Draw() function with a bunch of parameters over and over again.

As we've seen, a GPU is a complex piece of hardware, and is architecturally different than a CPU. One of the tasks of a Graphics API, in conjunction with the *driver software* provided by the GPU's manufacturer, is **translating** our API calls into actual hardware instructions for the GPU to carry out. But this isn't the only thing a Graphics API does for us. Another is **managing the memory** of the GPU, allowing us to copy data onto the graphics card while ensuring that data meets all of the rules and requirements of our particular GPU.

Another huge task that the API handles is the **synchronization** of our C++ code (the CPU-side of our application) and the GPU's workload (the GPU-side). These two processors (CPU and GPU) run asynchronously; they're completely independent processors. Any software that wants to control the GPU must handle synchronizing the two pieces of hardware. Without this caretaking, our application might, for example, overwrite data that is currently being use for a rendering operation or display a frame to the user before it is fully rendered, resulting in a completely malformed result.

What Are Drivers?

A *driver* is a piece of software that controls a hardware component attached to a computer, like a GPU. Drivers are the software that literally makes the hardware *do stuff* (they "drive" it, or make it *go*). Drivers are generally provided by the manufacturer of the associated hardware component and often abide by a specific API (this is separate from a Graphics API). Any program that needs to communicate with a hardware component – a GPU, a webcam or just a simple mouse – is utilizing the drivers for that particular piece of hardware.

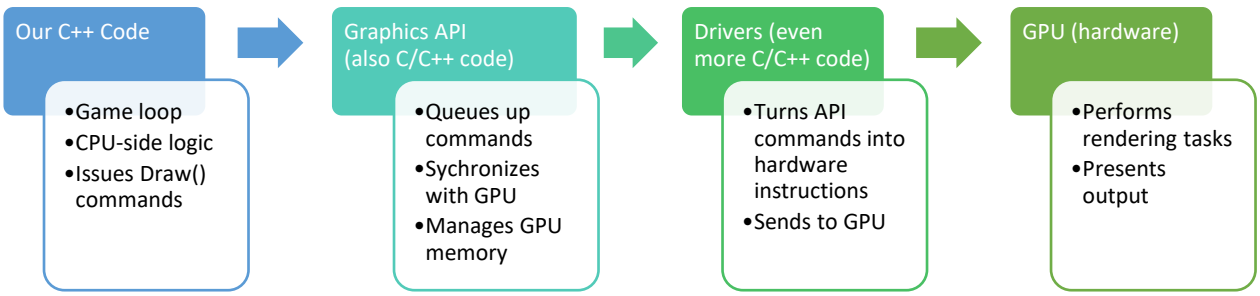
Why do we need drivers if we have our Graphics API? As stated above, a Graphics API is written with a common interface for programmers, and it needs to perform a common set of steps on the hardware. However, each and every GPU is a different; graphics cards from the same manufacturer often have different feature sets and cards from different manufacturers (like AMD, Nvidia and Intel) execute completely different instructions. It'd be impossible for a Graphics API to "speak the custom language" of every single video card on the market and also somehow anticipate all future GPU features.

So instead, the manufacturer provides drivers for each video card it makes. The same is generally true for any computer peripheral. It doesn't matter if you have an Nvidia RTX3080, an equivalent AMD card or an Intel integrated graphics card; your Graphics API can handle them all, as it simply tells the drivers what it wants to do. The drivers then make their specific hardware do it. What does this mean for us?

Layers

There are *several layers* our commands go through before the GPU actually executes them:

1. Our **C++ code** issues commands, such as "make this set of data the active buffer" and "draw using the triangles in the active buffer", to the Graphics API.
2. The **Graphics API** needs to queue these commands, synchronize them with what the GPU is doing at that exact moment, and tell the drivers to execute them once the GPU is free.
3. The **drivers** actually create and send the hardware instructions to the physical GPU.
4. The **GPU** performs the current task.



We're writing the C++ code that interacts with the Graphics API, which means we're not directly invoking the drivers or GPU. But seeing the rest of the equation can help us understand how best to use the API, why the API is structured the way it is and what the API's limitations are.

Graphics API Choices

There are several popular Graphics APIs in use today. The big five are DirectX 11, DirectX 12, OpenGL, Vulkan and Metal. They all have the same end goal: connect your code to the drivers/GPU to perform various rendering tasks. And, in general, they can all work with any modern graphics card, meaning they generally all support the same features, give or take. That being said, there are some key differences, especially between what are called **higher-level APIs** and **lower-level APIs**.


In a **higher-level API**, like DirectX 11 or OpenGL, much of the behind-the-scenes work like synchronization and memory management is done for you. The APIs try to make the best all-around choices for how to handle the details of hundreds or thousands of rendering commands each frame. The result is a mostly straightforward and easy to use API, though in some cases there are slight performance bottlenecks due to the API trying to automate things like memory management and synchronization. Some uncommon rendering techniques might benefit from the API doing things a slightly different way, but the API architects needed to make it as general purpose as possible.


In contrast, **lower-level APIs**, like DirectX 12, Vulkan or Metal, explicitly *don't* handle as much synchronization and memory management, leaving those tasks up to the application (C++ code) itself. This results in a big tradeoff: a much higher complexity at the application level, but *potentially* fewer performance bottlenecks since we can make the hardware do exactly what our unique game or engine requires. I say *potentially fewer bottlenecks* because this puts a lot of pressure on our application to do things correctly and efficiently. In fact, a naïve use of a lower-level API can actually result in *worse* performance than a high-level API, as we need to essentially re-implement much of what the high-level API was doing in the first place. You might liken this to the difference between C# and C++: potentially more powerful but a whole lot more to worry about.

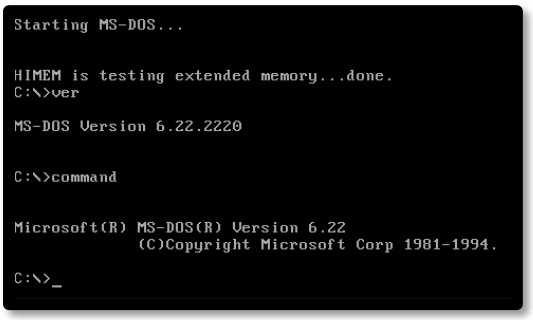
Our Graphics API: Direct3D 11

We'll be focusing on Direct3D 11 for a number of reasons. First, as a higher-level API, we can get up & running with it quickly, and end up covering a larger number of graphics topics overall. Second, it's a well-known API that has been in use for over a decade, meaning there are a significant number of articles, examples, professional talks, blog posts and other nuggets of knowledge available. Third, it integrates extremely well with Visual Studio, allowing us to compile our shaders at build-time and use a built-in "graphics debugger" to track down bugs.

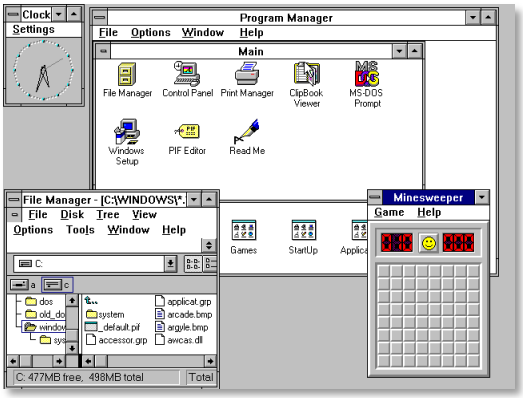
The hope is once you get the hang of a higher-level API like Direct3D 11, you should have an easier time transitioning to another API. Anything you learn here will be applicable to future endeavors as well. And you'll get to make some very impressive looking things instead of ending up with mountains of C++ code to just get a few triangles on the screen.

What is DirectX? How does it differ from Direct3D?

Technically, DirectX is an umbrella term for several different but related technologies. One of these technologies, **Direct3D**, is the actual Graphics API that we'll be using. However, the names DirectX and Direct3D are often used interchangeably. If a game is noted to "use DirectX", it generally means it is using Direct3D as its Graphics API. The next question is: where did DirectX come from?



Long before DirectX was a thing, even before the Windows OS, there were command-line operating systems. Microsoft's version, MS-DOS (Microsoft Disk Operating System), was one of the most popular. Game developers targeted the most widely-used operating systems and so many games were made for MS-DOS. Since the operating system was quite simplistic – allowing only a single program to run on the screen at any time – games could be guaranteed to have full control over the display. They essentially had *direct access* to the underlying hardware, as nothing else on the system needed it while a game was active.



Microsoft eventually created the Windows operating system, which provided a graphical user interface that allowed multiple programs to run simultaneously. In its earliest incarnations, Windows was really just a program that ran on top of MS-DOS. These early versions of Windows were tricky for game developers, as they no longer had direct hardware access; they had to go through the operating system itself to draw things to the screen. This added layer meant games were clunky and slow in early versions of Windows.

In the mid 90's, Microsoft was poised to release their brand-new Windows 95 operating system. But an operating system is only as good as the software it runs; if they wanted to court game developers, they needed to ensure games could work, and work well, on Windows 95. To that end, Microsoft created a team to build the *Windows Games SDK*, a set of libraries developers could use to more easily make games for Windows. The initial SDK (Software Development Kit) included libraries such as DirectDraw, DirectInput and DirectSound that allowed direct hardware access to various peripherals from within Windows. One journalist at the time mocked the naming convention, referring to the SDK as 'DirectX' due to the naming convention; the team liked it so much they decided to just use it going forward!

While early versions of DirectX were libraries developers would include with their games, modern DirectX is baked into the operating system itself. It is present on all Windows PCs and is the Graphics API for all Xbox game consoles. It was also used for the short-lived Windows Phone operating system.

Direct3D 11

This section contains an overview of Direct3D and how we use it.

Object-Oriented

First and foremost, Direct3D is an object-oriented C++ API. This means each time we want to use it, we'll be calling a function that belongs to an object. You'll see this throughout the starter code, like so:

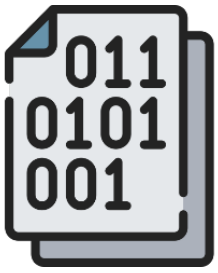
```
context->DrawIndexed(3, 0, 0);
```

This line Draws a set of primitive shapes by referencing the currently active Index buffer (hence, *DrawIndexed*). It explicitly uses the first 3 indices in that index buffer (the 3 is *how many* and the two 0's are offsets) to look up 3 vertices in the currently active vertex buffer. This function is part of the *context* object (short for *Device Context*). The exact type of primitive shape that will be drawn is defined by an earlier function call – IASetPrimitiveTopology() – which is also part of the *context* object.

The *context* object is one of three main objects we use to control the API. These objects were created when the starter code initialized Direct3D. You can think of these as our "doorways" into Direct3D:

- **Device** – Has functions for creating both *resources* (data on the GPU) and *API state objects* (groups of options for rendering)
- **Device Context** – Has functions for *setting rendering options* and *issuing rendering commands*.
- **Swap Chain** – Has functions for *presenting* the final rendering results to the user.

Resources



If we want to render something to the screen, we need geometry to rasterize. That geometry, whether it's a set of lines, a single triangle or a complex 3D model, is defined by vertices. Vertices are made up of one or more sets of numeric data, such as a *position* (3 float values for x, y and z) and a *color* (4 float values for red, green, blue and alpha). For the GPU to use vertices, they need to physically be within the GPU's memory. One of the major tasks of any Graphics API is to help us put necessary rendering data, like vertices, into GPU memory so that our rendering commands have data to work with.

Each set of rendering data on the GPU is called a *resource*. The most common rendering data we'll store in GPU memory includes **vertex** data, **index** data, **texture** data and **shader variable** data. In all four cases, these are just *buffers* (chunks of memory) holding sets of numbers. **Vertices**, and optionally **indices**, define our geometry – the shape of the object we want to draw. **Texture data** is used if we want access to an image as part of rendering, most often to apply to the surface of a 3D model. **Shader data** refers to the values we give to variables in our shader programs for the current rendering operation. Since shaders can read these values but not change them, they're often called *shader constants* and the corresponding GPU resource is called a *constant buffer*.

Resources, such as vertex buffers, index buffers, constant buffers (which hold shader variable data) and textures, are relatively slow to create, so we generally make them at start up (the *initialization* or "loading" phase of our engine). Creating any type of resource always follows a pattern: describe the resource you want, optionally provide the resource's initial data and ask the API to create it. All resources are created by calling corresponding functions on the Direct3D *device* object.

Here we see an example of a resource being created in the starter code, specifically a vertex buffer:

```
// First, describe the buffer we want Direct3D to make on the GPU
D3D11_BUFFER_DESC vbd   = {};
vbd.Usage               = D3D11_USAGE_IMMUTABLE; // Data will NEVER change
vbd.ByteWidth           = sizeof(Vertex) * 3;    // 3 = number of verts
vbd.BindFlags           = D3D11_BIND_VERTEX_BUFFER;
vbd.CPUAccessFlags      = 0;
vbd.MiscFlags           = 0;
vbd.StructureByteStride = 0;

// Struct to point to the initial vertex data
D3D11_SUBRESOURCE_DATA initialVertexData = {};
initialVertexData.pSysMem = vertices; // pSysMem = Pointer to System Memory

// Actually create the buffer on the GPU with the initial data
device->CreateBuffer(&vbd, &initialVertexData, vertexBuffer.GetAddressOf());
```

The description struct is filled out, specifying not just how large the resource needs to be, but also how we intend to use it (D3D11_BIND_VERTEX_BUFFER means we'll be *binding*, or setting, this resource as a set of vertices for rendering commands). We also provide the initial data by passing in, through a second struct, a pointer to it. Lastly, we ask the *device* object to create this resource.

One thing to note is that some resources, like vertex buffers, index buffers and textures, are generally flagged as *immutable* when created: our application can never change the resource's data. This helps the API & drivers optimize how they handle the resource under the hood. Luckily, it's quite uncommon to change most of these resources at run time anyway. When we use a resource for rendering, we're using copies of its data; we generally aren't able to write back to a resource while rendering.

> *The one exception to the general "resources should be immutable" rule is shader constant buffers. Shader constants are the variable data our shaders will use each time we render. This kind of data – transformations, camera matrices, light colors, etc. – changes every single frame and therefore needs to be re-sent to the GPU prior to rendering each object. This is one of the slower operations we'll perform, but it must be done. As you'll see in a future assignment, constant buffer creation has slightly different parameters to facilitate changing data at run time.*

These different resources types are used by various stages of the rendering process. Some, like vertex and index buffers, define *what* we're rendering – they're literally the input to the process. Constant buffers and textures define *how* we're rendering it. In any case, the overall rendering process – the steps of drawing one object – are referred to as the **Rendering Pipeline**.

Graphics State

Like all Graphics APIs, Direct3D requires us to answer questions before rendering: Which geometry is being used? Which shaders are active? Which primitive (points, lines or triangles) should be rasterized? Should we cull (ignore) the back sides of triangles? Should we use a depth buffer? Should triangles be filled-in or wireframe?

Part of our job when using the API is to answer these kinds of questions by setting associated options or *state*. All state changing is done through the *device context* object (just *context* for short). Some of these options have default values (such as filled-in triangles, and back-face culling), while others default to null: there is no default geometry and no default shaders. Sometimes, changing a setting is a single line of code. Other times it requires us to create an entire *state object*, containing several related options, by first filling out a description struct then calling a function to create the object.

Here is an example from the starter code of a single-line state change:

```
context->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
```

This changes the kind of primitive we'll be rendering. Once set, it remains until we change it again. If we wanted to render some geometry as lines and some other geometry as triangles, we'd need to change this setting, render the lines, change it again, render the triangles, and so forth.

Here is an example, *not* found within the starter code, of a more complex set of options being created and set. This example is defining options to control the rasterization phase of rendering. These options are grouped together into a *state object* which contains all settings for this particular feature. The steps below will probably look familiar: fill out a description struct with all necessary options and call a corresponding function on the *device* object to create the state object. This may seem like overkill, but the API is designed like this for a reason: the API & drivers can more quickly generate GPU instructions as they have more details about how a particular group of settings should interact. (Note that there are more options than presented below; these are the minimum variables that need non-zero values.)

```
// Rasterizer state to render back faces
D3D11_RASTERIZER_DESC rastDesc   = {};
rastDesc.DepthClipEnable         = true;
rastDesc.CullMode                = D3D11_CULL_FRONT;
rastDesc.FillMode                = D3D11_FILL_SOLID;
device->CreateRasterizerState(&rastDesc, backfaceRasterState.GetAddressOf());

// Change the state to use the new set of rasterizer options
context->RSSetState(backfaceRasterState.Get());
```

Regardless of whether we're changing a single setting or a group of settings, what we're doing is altering the *graphics state*. You can think of Direct3D as a large *state machine*: behind the scenes it tracks which resources (GPU data) are active (or *bound*) and which rendering options are set. (Note that "active" simply means those resources are denoted to be used for future rendering commands. Once created, resources remain on the GPU; we just flag which ones are to be used at any given time.) Whenever we issue a render command, such as DrawIndexed(), the API uses the entirety of the current state to generate the actual GPU instructions for rendering. It's our job to change the state as necessary to produce the results we want. That is a task we'll be rolling into our architecture as we build our engines.

Issuing Commands

There are many tasks we'll need Direct3D to perform. Some of them are simply changing state, such as "set this particular geometry to be used from now on" or "change the rasterizer options to only draw wireframe instead of filled-in triangles". Others, like "draw *N* indices worth of geometry", actually cause the GPU to perform rendering work. Function calls that specifically cause the GPU to render (or *draw*) something to the screen are often referred to as *Draw Calls*.

These types of commands – rendering and state changing – are all issued through functions of Direct3D's *device context* object (or *context* for short). You'll find examples of these kinds of functions during the **initialization** phase of our engine – for setting options that will persist for the entirety of our application – and during the **draw** phase – for preparing and rendering specific objects. Here are a few:

```
// Init() - State changes
context->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
context->IASetInputLayout(inputLayout.Get());
context->VSSetShader(vertexShader.Get(), 0, 0);
context->PSSetShader(pixelShader.Get(), 0, 0);

// Draw() - State changes
context->IASetVertexBuffers(0, 1, vertexBuffer.GetAddressOf(), &stride, &offset);
context->IASetIndexBuffer(indexBuffer.Get(), DXGI_FORMAT_R32_UINT, 0);

// Draw() - A "draw call"
context->DrawIndexed(3, 0, 0);
```

However, we generally do not interact with the API at all during the **update** phase of our engine, as that is primarily where we alter the game state: take user input, move the player, simulate physics, apply A.I., play sound effects, etc. As many of these tasks have an effect on what the final frame will look like, we don't want to begin the rendering process until the entire scene has been updated.

Common Commands

Here is a listing of several common Direct3D functions. All of these are part of the **context** object.

Draw Calls & Other GPU Work

Function	Purpose
`ClearRenderTargetView()`	Sets all pixels of the specified render target to the same color
`ClearDepthStencilView()`	Sets all pixels of the specified depth buffer to the same depth value
`Draw()`	Draws the first *N* vertices in the active vertex buffer(s)
`DrawIndexed()`	Draws the vertices from the active vertex buffers(s) corresponding to the first *N* indices in the active index buffer
`DrawInstanced()`	Draws the first *N* vertices in the active vertex buffer(s) *M* times each
`DrawIndexedInstanced()`	Draws the vertices from the active vertex buffers(s) corresponding to the first *N* indices in the active index buffer *M* times each
`Map()`	Locks a GPU resource, temporarily allowing the CPU to directly access it and preventing the GPU from using it until it is unlocked
`Unmap()`	Unlocks a resource, allowing the GPU to start using it again

State Setting & Resource Binding

Function	Purpose
`IASetPrimitiveTopology()`	Sets the type of primitive (points, lines, triangles) to draw
`IASetInputLayout()`	Tells the API how to interpret the numbers in a vertex buffer
`IASetVertexBuffers()`	Sets one or more vertex buffers as the active vertex buffer(s) *Note that it is rare to use more than one vertex buffer simultaneously, as corresponding data is pulled from all active vertex buffers when drawing.*
`IASetIndexBuffer()`	Sets the active index buffer, of which there can be exactly one at a time
`VSSetShader()`	Sets the active vertex shader, of which there can be exactly one at a time
`VSSetConstantBuffers()`	Sets one or more constant buffers for the vertex shader stage
`RSSetState()`	Sets all options for the rasterizer stage using a *rasterizer state object*
`PSSetShader()`	Sets the active pixel shader, of which there can be exactly one at a time
`PSSetConstantBuffers()`	Sets one or more constant buffers for the pixel shader stage
`PSSetShaderResources()`	Sets one or more shader resources (textures) for the pixel shader stage
`PSSetSamplers()`	Sets one or more sets of sampler options for the pixel shader stage
`OMSetRenderTargets()`	Sets the active render target(s) and, optionally, depth buffer

*Notice how each function above begins with a two-letter abbreviation: IA, VS, RS, PS, and OM. Each of these abbreviations corresponds to a phase (or stage) of the rendering process known as the **Rendering Pipeline**. IA stands for Input Assembler, VS stands for Vertex Shader, and so forth. This lets us know at a glance the step of the process that each function affects.*

Rendering Pipeline

The GPU doesn't perform any true rendering work – rasterizing geometry primitives and outputting the corresponding pixels – until we call a Draw() function (a "draw call"). Once we do, the multi-step rendering process known as the **Rendering Pipeline** begins. This process uses the current *graphics state* and all *bound (active) resources* to rasterize geometry from the active vertex buffer into pixels in the active output buffer. Once our application makes a draw call, we can no longer affect what happens during this invocation of the pipeline from C++; this means we need to ensure all relevant graphics state has been set and all necessary resources have been bound *before* calling Draw().

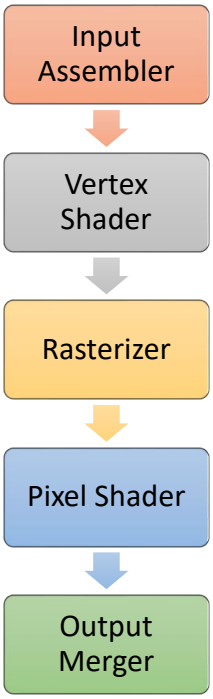
This pseudocode shows the general C++ process for arranging our state changes and draw calls:

```
// Change all settings necessary for the next Draw Call
SetVertexBuffers();
SetOtherGraphicsState();

// Perform rendering work (start the Rendering Pipeline)
Draw(someNumberOfTriangles);

// Change one or more settings for the next Draw Call
SetVertexBuffers();
SetDifferentGraphicsState();

// More rendering work (start the Pipeline again using the current settings)
Draw(moreTriangles);
```

Pipeline Stages

Triangles go in, pixels come out. What does the pipeline actually look like? What are the stages and what do they do?

There are five basic phases (or *stages*) to Direct3D's rendering pipeline. There are also several other optional, advanced stages that we won't need for the moment.



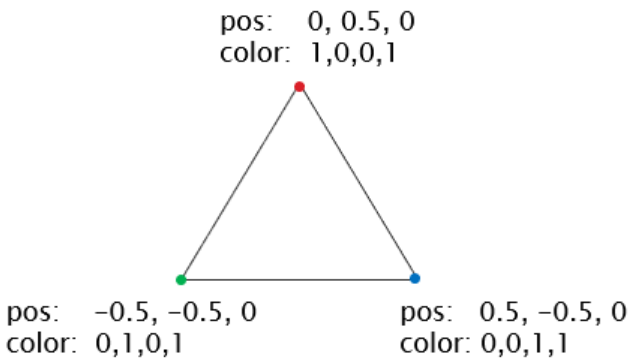
To the right, you'll see these basic pipeline stages, several of which correspond to rendering tasks we've discussed before. The Vertex Shader and Pixel Shader stages are *programmable*; they require us to provide the actual code for the GPU to execute. The other stages are *fixed-function*; we can change some of their settings and state, but they always perform the same work.

> *Before shaders existed, rendering pipelines were entirely fixed-function, which led to many games of that era looking remarkably similar. These days, all APIs have programmable pipelines and some, like Direct3D, have forgone the fixed-function-ness of shader stages entirely.*

Input Assembler

The input assembler stage is responsible for **reading vertex data** from the active vertex buffer (and optionally an index buffer). These vertices are then **arranged into primitives** based on the current *primitive topology* setting (points, lines, triangles) and then fed into the next stage. The input assembler launches *N* instances of the vertex shader (spread across our GPU cores), where N is the number of unique vertices retrieved from our buffers. In this way, the vertices are the *input* to the pipeline, hence the name "input assembler".

There is no concept of the screen or pixels at this stage. The vertex data is simply passed through to the next stage. Note that the input assembler works with *copies* of the vertex data; the original data in the vertex & index buffers remains unchanged.



Vertex data after the input assembler

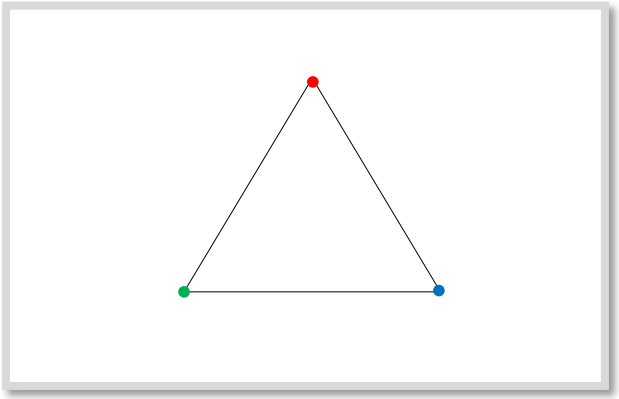
Vertex Shader

A **shader** is a small program that can be executed by a GPU. The exact code of a shader is up to us to write, though there are minimum requirements for what a shader must output to the next stage of the pipeline. If our shader doesn't output the necessary data, the rest of the pipeline might not produce any useful results. Note that there is no default code for this stage; if we don't provide a shader, nothing happens and the pipeline cannot proceed.

At a minimum, the purpose of the vertex shader stage is to **transform and project** vertex positions from their original locations (called "local space") into their final screen coordinates (called "screen space"). This is accomplished by multiplying vertex positions by one or more matrices representing these transformations and projections. These matrices, along with any other necessary variables, are provided to a shader through *constant buffers*: buffers that hold variable data a shader can read but not alter (hence "constant"). Since these matrices change often, our C++ code will be continually updating the contents of our constant buffers between draw calls.

> *There is still no concept of pixels at this stage. The coordinates we need to return are what we call normalized device coordinates. This coordinate system goes from -1 to 1 on the X and Y axes, making (0,0) the very center of the screen. Essentially, we're resolution-agnostic here, as we're describing positions as percentages across the screen.*

Another common task is passing other per-vertex data (colors, UV texture coordinates, surface normals, etc.) to the next stage as well. Some of this data might also need to be transformed or otherwise altered here based on the particular object we're drawing.



Vertices after the vertex shader stage

Rasterizer

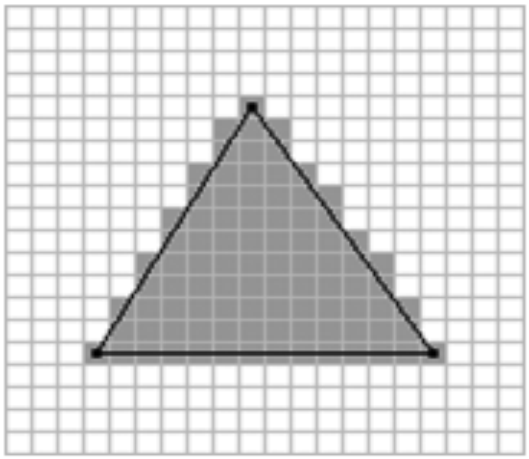
The rasterizer's primary task is to, you guessed it, **rasterize**: determine the set of pixels comprising each projected triangle. The rasterizer then launches *N* instances of the pixel shader (spread across GPU cores), where N is the total number of rasterized pixels.

To speed this process up, the rasterizer can be configured to ignore any triangles whose backsides are facing the camera (known as "back-face culling"). Note that if two or more triangles of a single model end up covering the same pixel in the output buffer, that pixel will be processed separately *for each*

triangle. This is why back-face culling is so useful; we only want to spend time processing pixels we can actually see!

Additionally, the rasterizer will interpolate vertex data (position, texture coordinates, normals, etc.) across the face of each triangle, providing unique versions of that data to each pixel shader invocation. This allows each pixel of a single triangle to have slightly different data, which is important for proper texturing and shading calculations (otherwise each pixel of a single triangle would look the same).

> *This is the first stage where the actual pixels are taken into account. The rasterizer maps the normalized device coordinates to the pixel dimensions of the output buffer. Any triangles that partially hang off the edge of the buffer are clipped (we ignore those pixels). Any triangles that are fully outside the buffer are completely skipped.*



Pixels after rasterization
Each has unique data associated with it (not pictured)

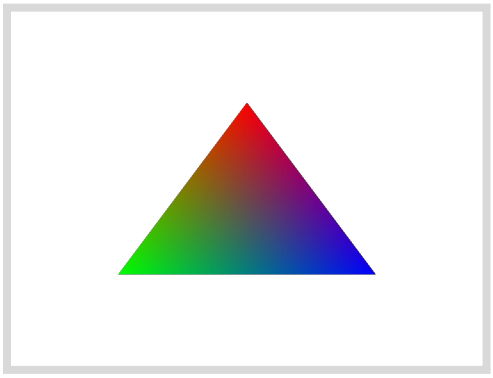
Pixel Shader

The pixel shader stage is another programmable stage. Like the vertex stage, there is no default, so we must provide a pixel shader. At a minimum, a pixel shader must **output a color** as four 0-1 float values: red, green, blue and alpha (RGBA). Note that the pixel shader only runs for rasterized pixels within the bounds of our window, *not* each and every pixel of our entire window (unless the object really is that large). The smaller an object appears, the fewer pixels need to be processed.

The simplest (and least interesting) possible pixel shader would output the exact same color for each pixel regardless of any input data. Our starter code's pixel shader is ever so slightly more interesting: it outputs whatever color value it received from the rasterizer. Since the rasterizer interpolated the vertex colors (red, green and blue) across the triangle, we see that each pixel ends up with a unique color.

This is where most of the heavy lifting occurs in the rendering pipeline: texturing, lighting calculations, approximated reflections, color correction and so forth. This means the pixel shader stage is usually the

costliest stage in terms of performance. Manually optimizing this code and cutting down on the total number of pixels to process are often of the utmost importance in high-performance games.



Output of the pixel shader
Only pixels corresponding to the triangle(s) are actually processed

Output Merger

Each object we render goes through the entire pipeline in isolation; one object is fully finished before the next begins. Within a single frame, we might draw one, ten or a thousand objects. The pixel colors from each draw call are placed in the output buffer (a 2D grid of pixel colors – essentially an image) as we render the frame object-by-object. At the end of the frame, this buffer is presented to the user.

> *We often have two of these buffers, one that collects rendering results and one that is being displayed to the user. We call the particular buffer that collects our rendering results the back buffer and the one currently being displayed the front buffer. Once a frame is completed, the two buffers swap roles. This setup is called double buffering.*

The output merger's job is to **merge the results of a draw call** into the back buffer. By default, this is done by *overwriting* whatever is currently in the back buffer with the results of the latest draw call. The new object appears "on top" of everything else, regardless of the alpha values of the pixels.

New pixel shader output + *Current back buffer contents* = *Merged results*

If a *blend state* is enabled for this stage, instead of overwriting pixel colors, the new and existing colors are *blended* together based on the pixel's alpha value. This is how transparency is achieved, though there are numerous issues that make transparency quite tricky to get perfect, so it's disabled by default.

Optionally, we could create and activate a *depth buffer* for this stage. This allows the depth (or distance from the camera) of each rasterized pixel to automatically be stored by the pipeline. If a depth buffer is active, the output merger first compares the depth of each new output pixel with the existing depth at that pixel in the depth buffer. The new pixel color is only kept if its depth is smaller than the existing depth value, meaning the pixel represents a surface that is closer. This gives us per-pixel occlusion, meaning triangles can accurately clip through each other rather than one always being "on top".

What Next?

Whew! There's a lot that goes into rendering a single triangle. Our main tasks are ensuring the necessary data is in the correct place (buffers on the GPU), the correct graphics state is set up for the object we want to render and then issuing a draw call.

Luckily, rendering a set of triangles is effectively the same amount of work for us; we just need to provide a larger number of vertices to the rendering pipeline. Things get tricky when we want to render distinct objects, especially if we want unique transformations for each one. That requires placing transformation data in the correct place on the GPU so our shaders can utilize it. That's where we're heading next.