Graphics Programming & GPUs

What is Graphics Programming?

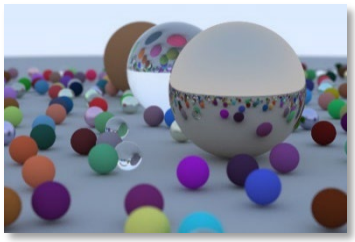
At its core, **graphics programming** is when we write code to create some sort of visual output, whether it be the individual frames of a real-time video game, a static image file, an entire feature film or some other artifact. This overarching field of study is called "**computer graphics**" (CG for short). It includes topics such as ray-tracing, rasterization, vector graphics, real-time graphics, computer vision, animation, film, data visualization and much more. These topics are also built upon other areas of study, such as geometry, linear algebra and physics, among others.

What Do I Need for Graphics Programming?

Technically, not much! If you want to do rudimentary graphics programming, all you need is a **programming language**, some **math** and a way to **show off the results**. Practically any language will do, assuming it has the capability to display an image on the screen and/or write data to a file.

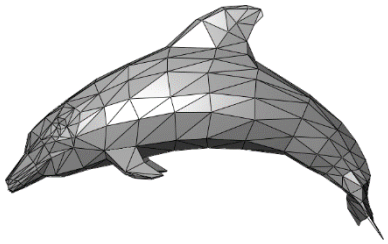
If you can create a 2D array of color values, then loop and fill each element with some meaningful color, you're programming graphics; no fancy GPU or complex real-time graphics API required! You could absolutely begin applying basic computer graphics topics like geometry rasterization (turning 3D models into pixels) and shading (approximating lighting on virtual surfaces) with little else. And doing so can be a great introduction to the field.

> *Peter Shirley of Nvidia has written an excellent online book series called Ray Tracing in One Weekend. This book starts with a blank C++ project and builds a ray tracing application from the ground up, including all of the required math and mathematical structures (like vectors).*
>
> *Running the program produces next to no actual screen output as it spends its time calculating the final color of each pixel in the image, one by one, and then saves those results to a file, as seen to the right. However, the file can take **minutes or hours** to produce, depending on how detailed you want it to be.*



What you will find, however, is that doing this at scale – hundreds of 3D models, with thousands of triangles each, turned into millions of pixels and colored by approximating the physics of light interacting with varying surface materials – with a big loop in your code is going to be extremely slow. Too slow for a real-time app like a game. That is, unless we can utilize special hardware to speed things up. This is where a GPU, which is controlled through a graphics API like Direct3D or OpenGL, comes in. But before that, let's look at the major approaches to 3D rendering.

3D Rendering

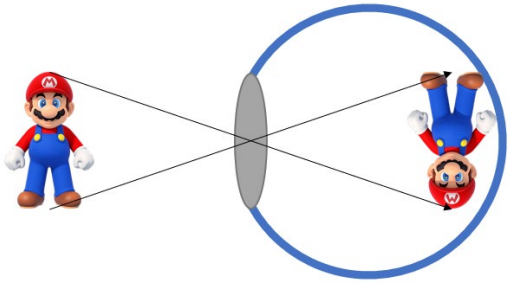


Our focus is going to be on rendering for real-time 3D games, so we'll need a way to perform **3D rendering**. What do we mean by "3D rendering"? Generating a 2D image from one or more 3D models. A 3D model in this context is simply an arrangement of triangles. The shape of each triangle is defined by its vertices (the corners of the triangle). Each vertex represents, at a minimum, a position in 3D space, though each could have other useful rendering data associated with it such as a color, texture mapping coordinates and/or a surface normal for lighting calculations.

How can we take a 3D model – which is a list of 3D positions, in groups of three – and turn it into pixels in an image? There are two major techniques for this: **raytracing** and **rasterization**.

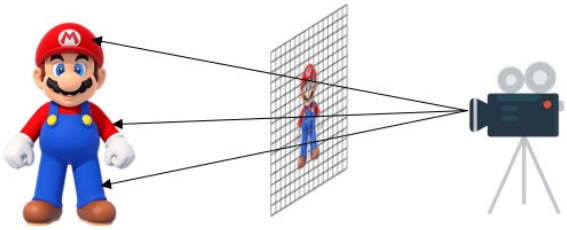
Raytracing

What are we really trying to do when rendering a 3D model? We're trying to mimic the way we as human beings might "see" a real object: light strikes the object, bounces off and hits our eye. More specifically, the light is focused through the lens of our eye and projected (upside down) onto our retina.

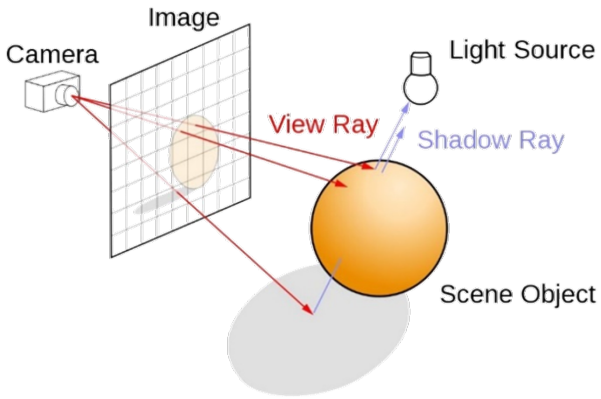


There is far, far too much light in the real world to attempt to simulate it all, especially since most of it wouldn't contribute to the final image we're rendering. Instead, we only need to handle the light that would hit our virtual "eyeball" (or "camera"). How do we know which light would hit our eye without simulating all of it? We work backwards.

For each pixel of the image we're rendering: starting from the camera's position, we trace a mathematical ray out into the 3D scene, passing through this pixel's position, until the ray hits a surface. In this setup, the image is effectively our retina. To simplify even further, we can write our code such that the image (our simulated retina) is *between* the camera (our simulated lens) and the 3D scene, resulting in the image being rendered rightside up.



Does this “backwards” technique really work? It sure does, and it’s the basis for **raytracing**. As you might imagine, there is *much* more work involved in creating a photorealistic image. Knowing a ray hit a surface only tells us where the surface is; it doesn’t tell us the perceptual color of the surface, or where the light came from *before* striking this object (as in reflections).



Of course, raytracing can solve these issues, too. However, this requires much more work and access to much more data. For instance, we can recursively send out another ray from this surface to see where the light might have come from (simulating the light bouncing or reflecting). And if that ray hits another surface before a light? Do it again, and again, and again. Not only do we need access to the object we’re currently rendering, but we also need access to the *entire 3D scene*, as the light could have bounced off any number of other objects. Tracing the full path of light, rather than a single ray, is a type of raytracing called “path tracing”.

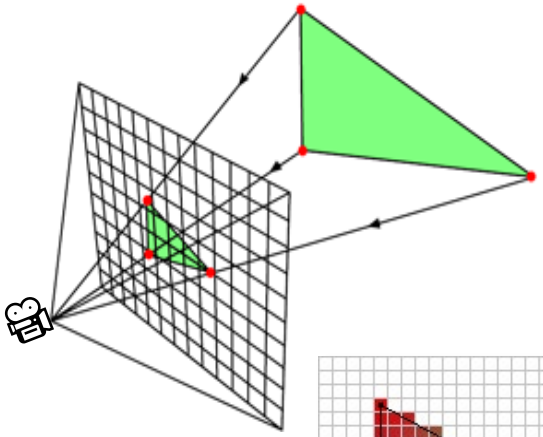
In summation: raytracing works well and can produce high quality results, but is generally quite slow, especially as we account for more optical phenomena. The higher quality, the longer it takes. This has traditionally made raytracing too slow for real-time applications like games.

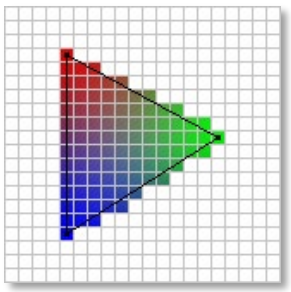
Rasterization

The goal of 3D rendering is to figure out which pixels represent each object and then color those pixels accordingly. Raytracing is just one way of determing those pixels. Another approach is called **rasterization**, which is the process of turning a **vector** shape (a shape defined by geometry, like curves or triangles) into a **raster** image (a set of pixels).

The basic steps of rasterization are as follows:

1. Using matrix math, project the 3D position of each vertex onto the 2D image.
2. Given these three 2D positions, determine the pixels within the triangle’s bounds.
3. Optional: Interpolate any other vertex data (such as color) across the triangle.
4. Determine the final color of each pixel.





While any geometric shape can be rasterized, modern 3D graphics generally focuses on 3D models comprised of triangles. Why? Because triangles, the simplest 2D shape, are always flat (since their vertices are always co-planar). This greatly simplifies step 2 in the process above, ensuring rasterization is fast.

One big advantage of rasterization is that the only data we need is the object currently being rendered. As we'll see later, this makes the process of rasterization a prime candidate for being sped up by specialized hardware (GPUs), allowing it to be done in real-time.

The downside here is that we don't have lighting information from the whole scene. We *do* have all of the object's data (color, texture mapping, etc.), so we can still create a high quality image. However, the final pixel colors won't inherently be influenced by the rest of the 3D world.

In the examples to the right, the same scene is raytraced and rasterized. The raytraced scene has obvious reflections on the teapot and spoon, but also much softer lighting on the bottom of the teacup, as light hitting the saucer is reflecting up towards the cup. In the rasterized version, we lose all of that *global illumination* information, but gain performance. With the right assets and techniques, some of these effects, such as the reflections on the teapot, can be approximated well enough to look plausible (though not necessarily physically accurate).

Rasterization vs. Raytracing

If raytracing produces better results, why isn't it the default? Because it's just too slow to use exclusively. Modern computer graphics hardware – video cards (or "GPUs") – are built specifically to speed up *rasterization*. And, as stated above, we can add additional techniques to the rendering process to further improve the final image. For example, these screenshots of Star Wars Battlefront 2 from 2017 look pretty darn good! Next, we'll look at how that specialized hardware came about.

What the Heck is a GPU?

A video card, also known as a graphics card or a GPU (Graphics Processing Unit), is a core feature of all modern computers. Any visual output to a monitor generally goes through a GPU. If we can perform graphics programming in any language, why do we need this special graphics hardware? What does a GPU really do for us?

First: A Quick Step Backwards

Long before modern GPUs, video cards in early computers were called *display adapters*. They did little more than turn a digital signal into an analog signal displayed on what was essentially a simple television.

Applications (like games) would change the colors in a chunk of memory – a *buffer* – that the video card would read and send to the display. The relative speed (or lack thereof) of computers in that era, combined with low-resolution displays and very little memory, meant early graphics were quite limited.

Video Card Improvements

Over time, common tasks that programmers had to do, such as displaying ASCII text on a screen, drawing simple primitive shapes and storing the buffer (memory) for the display, were embedded into video cards themselves. Being able to send an instruction to the hardware to perform a specific task meant an application could spend more time doing other work like physics, AI, etc. Having the display buffer embedded in the video card meant the card itself could write directly to this memory, freeing up the system's memory for other applications.

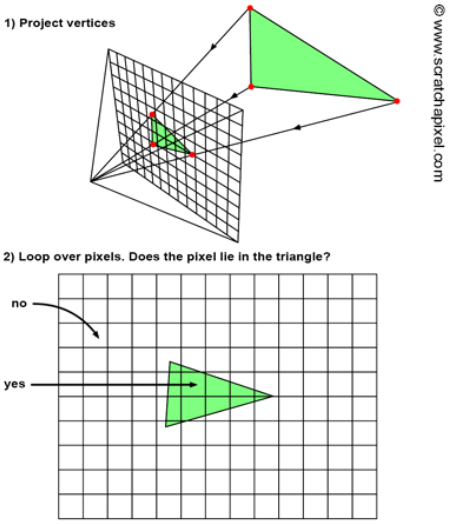
As time went on, the amount of memory on video cards increased, as did the resolution of displays. Rather than being limited to 320x200 pixels, displays could be 640x480, 1024x768 or even higher. More pixels, however, meant more work to display a game across the entire screen. Computers were getting faster (luckily), but at the same time games were getting more and more complex.

From 2D to 3D

2D games require 2D images. During the game, pixels of those images are copied into the display buffer (which is effectively its own image – a 2D array of color data). Every frame of the game, different images are copied into the buffer, resulting in movement and animation.

In the 90's, 3D games (games in a virtual 3D space, like Doom and Quake) were becoming popular. No longer was it enough to just copy 2D images around the screen; we needed some sort of actual 3D rendering.

Rasterization



Let's look at **rasterization** again, where mathematical representations of triangles (3 points in space) are converted to specific pixels in an output image. If you start with triangles in the shape of a car, you'll see that car in the final image.

Rasterization requires transforming (projecting) each vertex of each triangle from 3D space to a 2D plane through the use of matrix multiplication. Once projected, the exact pixels that the triangle overlaps are determined. Lastly, each pixel has to be colored in some meaningful way, often by performing a lighting calculation and/or pulling colors from a texture.

The overall workload of rasterizing a group of triangles to create an interesting image is much higher than plastering several 2D images onto a screen. There was only so much you can do with a big loop that iterates over every vertex and/or pixel. A new paradigm was required to push the boundaries of real-time computer graphics.

A New Challenger Approaches: 3dfx

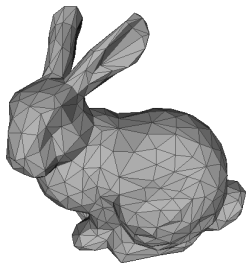
Near the end of 1996, a company called 3dfx Interactive released the Voodoo graphics add-on card and practically revolutionized real-time 3D graphics overnight. This card, which had to be plugged into an existing graphics card, could perform triangle rasterization and basic texturing on the card itself. This was a huge win for game programmers, as a lot of the heavy lifting in 3D rendering was done for them. The release of the Voodoo card caused several other companies, namely Nvidia, to step up their game. Throughout the late 90s and early 2000s, video cards were being released with more and more features that could speed up rendering, freeing applications up to do more as well. It was obvious that this kind of hardware was going to become a mainstay of home computers.

Where Are They Now?

If 3dfx was so amazing, where is it now? The company was eventually outpaced by competitors and was ultimately acquired by Nvidia. But its legacy as one of the early hardware-accelerated 3D graphics giants remains. We could spend many more pages delving into the history of graphics cards, but will instead focus on how they help us with graphics programming today.

If you want to know more about the history of graphics cards, see The History of the Modern Graphics Processor on TechSpot. And for a more detailed discussion of exactly how the Voodoo card worked and how video cards evolved over the next decade, check out Appendix C. History of PC Graphics Hardware from *Learning Modern 3D Graphics Programming* by Jason L. McKesson.

So…What Does a GPU Really Do for Us?



A lot has to happen when we render a 3D model. First, each vertex of each triangle has to be transformed and projected onto a 2D plane (the screen) using matrix multiplication. Then each of those triangles needs to be rasterized to figure out which pixels they cover. Lastly, each of those pixels needs to be colored, often with a combination of a texture and lighting approximations.

Let's map these steps out in pseudocode with, say, 100 triangles:

```
Triangle[] allTriangles;      // An array of 100 triangles
Vertex[]   projectedVertices; // Projected vertices to be rasterized
Pixel[]    finalPixels;       // All pixels to be output

// Process each vertex of each triangle
foreach(Triangle t in allTriangles) // 100 triangles
{
        foreach(Vertex v in t.vertices) // 3 vertices per triangle
        {
                // Perform transformations & projection on each vertex
                projectedVertices.Add(v * transformAndProjectionMatrix);
        }
}

// Rasterize each group of three vertices
for(int i = 0; i < projectedVertices.Length; i += 3)
{
        // This step itself can be slow, as rasterization
        // could result in any number of pixels!
        Pixel[] trianglePixels = Rasterize(
                projectedVertices[i],
                projectedVertices[i + 1],
                projectedVertices[i + 2]));
        finalPixels.Add(trianglePixels);
}

// Process all pixels that have been rasterized
foreach(Pixel p in finalPixels)  // Could be upwards of millions of pixels!
{
        p = Texture();           // Apply image
        p *= Lighting();         // Apply lighting
        p *= OtherEffects();     // Apply other effects
}
```

If you're thinking "wow, that seems like a lot of work", you're right. It absolutely is. However, this pseudocode reveals a few things about the process. First, there are three distinct phases: **vertex transformation**, **rasterization** and **pixel coloring**. Second, notice how each vertex is processed in isolation; the results of any one vertex's transformation have no bearing on any other vertex. Likewise, the same is true for rasterization and for pixel coloring.

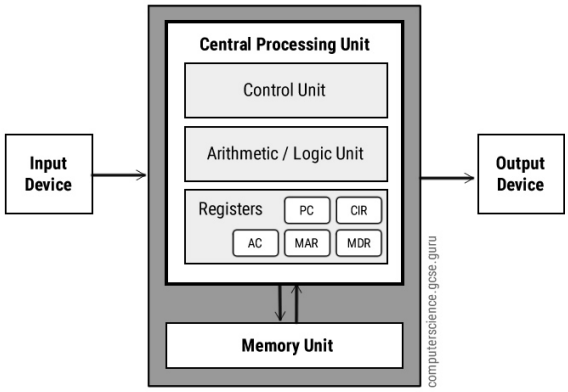
Modern GPUs are designed to handle this exact process. They can store the triangle data we intend to draw, manipulate copies of that data as it progresses through the phases of rendering and, most importantly, turn each of the loops in the above pseudo-code into parallel processes. In other words, instead of finishing vertex 1, then moving on to vertex 2, then on to vertex 3 and so on, a modern GPU will process each vertex literally **at the same time**. This is a *massive* speedup to the rendering process!

CPUs & GPUs

When you think of a simple computer program, you probably (correctly) picture it as a set of steps that occur in order. The first line runs, then the next and so on until the program ends. In between the first and last lines, there is a combination of simple statements & expressions, function calls, branching (*if* and *switch* statements that might cause different paths or branches of code to execute) and/or loops that repeat the same code blocks multiple times. Each of these statements is made up of one or more instructions that the computer carries out. Most simple programs truly do execute one instruction at a time and they don't move on to the next step until the previous one is finished.

This is exactly what a **processor**, or Central Processing Unit (CPU), does. To facilitate the execution of instructions, modern CPUs contain several important components. First, the **arithmetic-logic unit** (ALU) performs math and logic operations, though it can only act on data that physically resides on the CPU. This is where **registers** come in: they're very small amounts of *very fast* data storage. Registers hold the values being read and written by the ALU; an ALU can only operate on data in its registers. Lastly, the **control unit** (CU) is responsible for fetching instructions (code) from RAM, and then decoding & executing those instructions in order. It can also copy data into the ALU's registers.

Examples of instructions include tasks like "copy this variable's data from memory into the first register", "perform addition on the numbers in the first two registers", and "store the result in the third register". Note: These are the kinds of instructions that compilers generate from the higher-level code we write.



CPUs contain several other components, too. For instance, CPUs also have several levels of onboard memory, called **cache**, to hold frequently used data and instructions so that less time can be spent fetching them from RAM. This is a pretty big oversimplification, but hopefully it sheds some light on what is going on inside the devices we use every day.

Multi-Core Processors

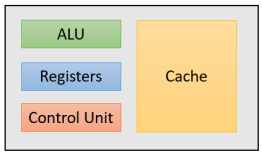
Computers can perform tremendous amounts of work incredibly quickly. But how do we make them even faster? Making the components of a CPU – the ALU, registers and CU – faster can make the overall computer quicker, but there are physical limits to this approach. However, the last few decades have seen a shift in processor design: rather than just getting *faster* CPUs, we're getting *more* CPUs.

Most modern processors actually contain multiple physical CPUs, called **processors cores**. A quad-core processor, for example, contains four processors on a single chip. When a simple program runs, it's really just running on one of these cores. However, the operating system can distribute the workloads of different programs onto different cores, allowing the computer to literally perform different tasks at the same time. That is a pretty big speedup! We can even write more sophisticated multithreaded programs to exploit this hardware, distributing the workload of a single program over multiple cores.

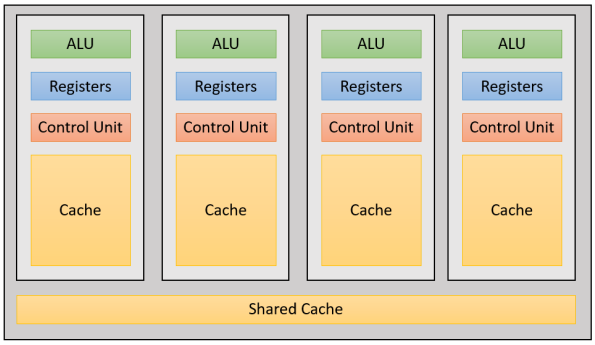
Parallel Processing for Graphics

So, what does this have to do with triangles? Think back to the rasterization pseudocode from earlier: we had hundreds of vertices and upwards of millions of pixels to process *every frame*. If we could distribute that work among multiple CPU cores, we could get a decent speedup. Say, for instance, we had a 4-core CPU; we could get an almost 4x speed up if we were able to multithread the code to distribute the work over the entire processor.

Here are extremely simple diagrams of what a single core and a 4-core CPU might look like.



Single-Core CPU

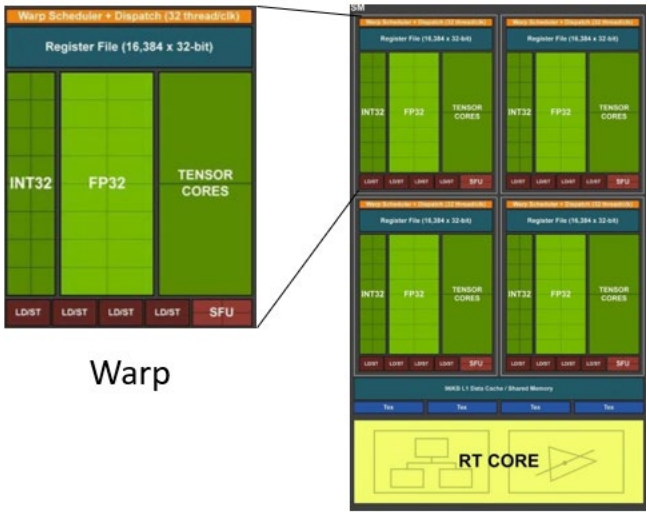


Quad-Core CPU

Unfortunately, that still won't be fast enough for the scale of work we'll be dealing with. We need a piece of hardware that can process not just tens or hundreds but *thousands* of units of work simultaneously. For that, we need hardware that is custom built to be able to do massive amounts of work at once. This isn't just about adding more cores; this requires an entirely different architecture.

GPU Parallelism

So how is a GPU different? First, modern GPUs have different types of cores: some perform integer math, some perform floating-point math, and so forth. These cores are arranged into processing blocks called *warps*. On the right, you'll see a *warp* from Nvidia's Turing series of GPUs, which contains sixteen 32-bit integer cores, sixteen 32-bit floating-point cores, and several other components.



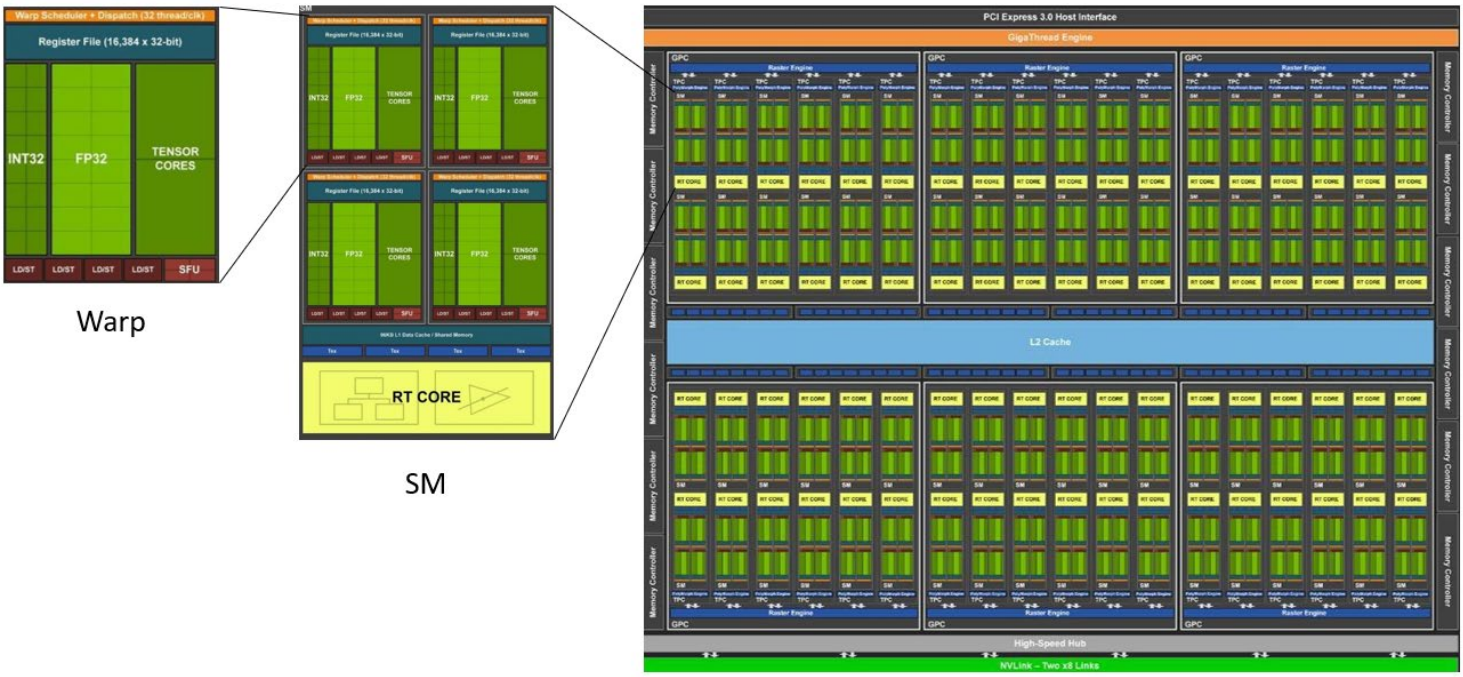
Warp

SM

These warps are then arranged in groups of 4 called Streaming Microprocessors (SMs), also seen to the right. These also include memory that can be shared among the warps, as well as cores for performing other types of work (like texture sampling and raytracing). Dang, that seems like a lot of cores!

Well, yes and no…

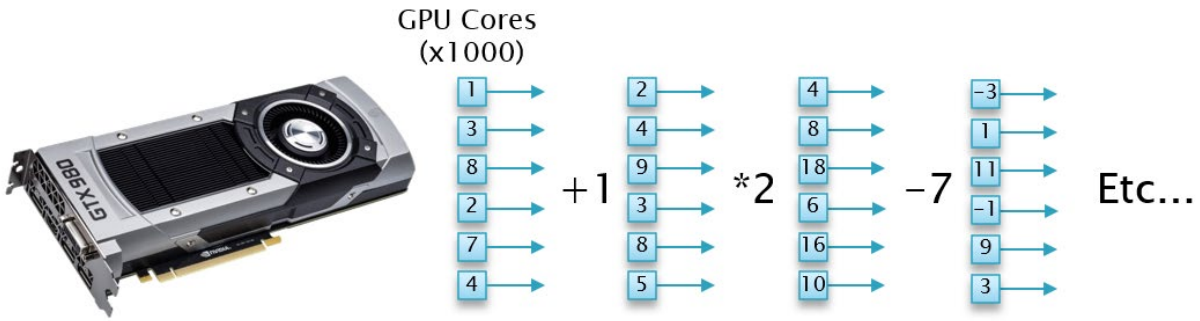
Here's an entire GPU diagram. **4,608 cores**. Now *that's* a lot, each of which can work simultaneously!



SIMD: Single Instruction, Multiple Data

GPUs are designed to be massively parallel; they're not just a bunch of CPUs duct taped together. The architecture of a GPU allows for thousands of operations to be performed simultaneously, but with a big caveat: all of the cores of a warp need to be performing **the exact same operation at the same time**. If one core is adding, they're all adding. If one core is multiplying, they're all multiplying.

The paradigm that drives this architecture is called "Single Instruction, Multiple Data", or SIMD (pronounced SIM-Dee). In SIMD architecture, each core must be performing the same, singular instruction (like adding, subtracting, etc.) at any given time, but each can be operating on different pieces of data. One instruction, different data.



Keep in mind that this is an oversimplification. Technically, each warp can be performing different tasks, but is SIMD internally. However, multiple warps often work in tandem.

SIMD & 3D Rendering

At first glance, this might seem less-than-ideal. What good are all these cores if they have to be doing the same thing? Let's think back to the pseudocode of rasterizing 100 triangles. The first step was to transform each vertex's 3D position to a position on the screen. This required us to multiply 300 different positions by the same matrix. This means we need to repeat the exact same instructions (the algebra required for vector/matrix multiplication) 300 times using different data. Same instructions, different data.

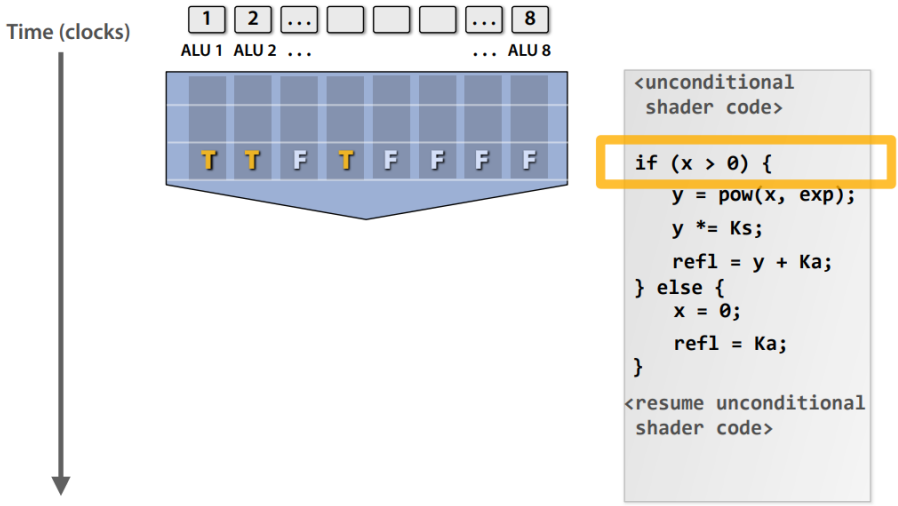
Moving on to the rasterization phase, the same thing had to happen. Process 100 triangles by performing the same work on each. Same instructions, different data. And then the pixel coloring phase? Same thing again: same instructions for lighting, texturing and so forth, simply using different input data.

As it turns out, modern 3D rendering via rasterization maps perfectly to GPU architecture. Almost as if these GPUs were designed to perform, and thereby speed up, these exact tasks!

GPU Drawbacks

GPUs seem crazy powerful! Why don't we just use them for everything? Due to their SIMD architecture, there are some tasks that they aren't great at. One in particular is dealing with branching.

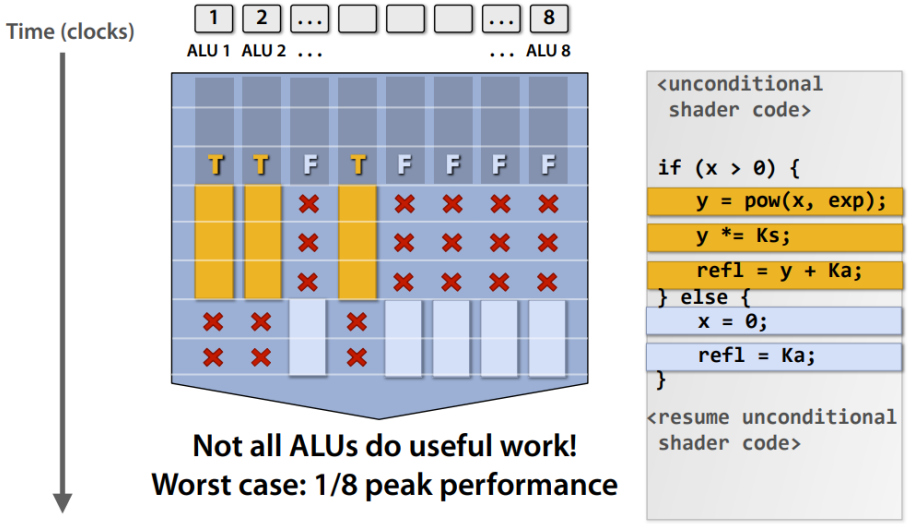
Branching is when your code does one thing or another depending upon some condition; in other words, when your code hits an if statement or a switch statement. Let's visualize this a bit:



(Slide 48 from "How a GPU works" at CMU)

As work happens on a GPU, each core is running in lockstep: each one adds, then subtracts, etc. However, when a branch is encountered, different cores may need to take different paths (some go into the *if* while others go into the *else*). How does this work if they all need to be executing the same instructions? Some cores just sit and wait for their turn!

As shown below, if one set of cores should be executing code A while the other set of cores should be executing code B, one set will literally *sit idle* while one block is executed, and then swap to handle the other block. The result is that the conditional statement – the branch – might cause the overall code to take *longer*! High performance shader code is often optimized, by hand, to remove branches.



(Slide 49 from "How a GPU works" at CMU)

Now, this isn't to say that we can't or shouldn't use branches. Branches are *fine*, especially if we know all cores will take the same branch, or when we're writing code to learn. But this does illustrate a non-obvious consequence of modern GPU architecture.

> *Since GPU cores can pause threads of execution rather than always execute the same instruction, GPU architecture is sometimes called SIMT: Single Instruction, Multiple Threads*

What About Raytracing?

Despite what the marketing slogan "RTX On" might imply, moving from rasterization to raytracing is not as simple as flipping a switch. It's a fundamental change in the work necessary for rendering, requiring sweeping changes to rendering engines. And, even with those changes *and* the right hardware, it's still somewhat slow. In fact, many games utilize a hybrid approach: rasterization for the bulk of the 3D scene, and raytracing "on top" for the pieces that are most noticeable, such as reflections.

What's Next?

GPUs are the key to real-time 3D graphics. Hopefully I've shed a little light on how they work, why they work that way and how we got here. So, what's next? We'll be getting into how we can take advantage of the power of our GPUs to perform real-time 3D rendering. And for that, we'll need an API specifically designed to speak the language of a GPU, to allow us to control it and to help synchronize this otherwise-asynchronous piece of computing hardware.

Soon we'll be looking at graphics APIs. Specifically: Direct3D 11.