



3D Graphics: Crash Course Computer Science #27

Crash Course: Computer Science

<https://youtube.com/watch?v=TEAtmCYYKZA>

<https://nerdfighteria.info/v/TEAtmCYYKZA>

===== Intro (00:00) =====

Hi, I'm Carrie Anne, and welcome to Crash Course Computer Science. Over the past five episodes, we've worked up from text-based teletype interfaces to pixelated bitmap graphics. Then, last episode, we covered graphical user interfaces and all their ooey-goey richness.

All of these examples have been 2D. But, of course, we're living in a 3D world and I'm a three-dimensional girl. So today, we're going to talk about some fundamental methods in 3D computer graphics and how you render them onto a 2D screen.

[Crash Course intro]

===== Wireframe Rendering (0:36) =====

As we discussed in episode 24, we can write functions that draw a line between any two points, like A and B. By manipulating the X and Y coordinates of points A and B, we can manipulate the line. In 3D graphics, points have not just two coordinates, but three: X, Y, and Z. Or "zee". But I'm going to say "zed".

Of course, we don't have X/Y/Z coordinates on a 2D computer screen, so graphics algorithms are responsible for flattening 3D coordinates onto a 2D plane. This process is known as 3D projection. Once all of the points have been converted from 3D to 2D, we can use the regular 2D line-drawing function to connect the dots, literally. This is called wireframe rendering.

Imagine building a cube out of chopsticks and shining a flashlight on it. The shadow it casts onto your wall, its projection, is flat. If you rotate the cube around, you can see it's a 3D object, even though it's a flat projection. This transformation from 3D to 2D is exactly what your computer is doing, just with a lot more math and less chopsticks.

There are several types of 3D projection. What you're seeing right now is an orthographic projection, where, for example, the parallel sides in the cube appear as parallel in the projection. In the real 3D world, though, parallel lines converge as they get further from the viewer, like a road going to the horizon. This type of 3D projection is called perspective projection. It's the same process, just with different math. Sometimes you want perspective, and sometimes you don't. The choice is up to the developer.

Simple shapes like cubes are easily defined by straight lines. But for more complex shapes, triangles are better - what are called polygons in 3D graphics. Look at this beautiful teapot made out of polygons. A collection of polygons like this is a mesh. The denser the mesh, the smoother the curves and the finer the details. But, that also increases the polygon count, which means more work for the computer.

Game designers have to carefully balance model fidelity versus polygon count, because if the count goes too high, the frame rate of an animation drops below what users perceive as smooth. For this reason, there are algorithms for simplifying meshes.

The reason triangles are used, and not squares or polygons or some other more complex shape, is simplicity: three points in space unambiguously define a plane. If you give me three points in a 3D space, I can draw a plane through it—there is only one single answer.

This isn't guaranteed to be true for shapes with four or more points. Also, two points aren't enough to define a plane, only a line, so three is the perfect and minimal number. Triangles for the win!

===== Scanline Rendering (3:03) =====

Wireframe rendering is cool and all – sort of retro – but of course 3D graphics can also be filled. The classic algorithm for doing this is called scanline rendering, first developed in 1967 at the University of Utah. For a simple example, let's consider just one polygon. Our job here is to figure out how this polygon translates to filled pixels on a computer screen. So let's first overlay a grid of pixels to fill. The scanline algorithm starts by reading the first three points that make up the polygon, and finding the lowest and highest Y values. It will only consider rows between these two points.

Then, the algorithm works down one row at a time. In each row, it calculates where a line running through the center of a row intersects with the side of the polygon. Because polygons are triangles, if you intersect one line, you have to intersect another. It's guaranteed.

The job of the scanline algorithm is to fill in the pixels between the two intersections. Let's see how this works. On the first row we look at, we intersect here and here. The algorithm then colors in all pixels between those two intersection. And this just continues row by row, which is why it's called scanned line rendering. When we hit the bottom of the polygon we're done. The rate at which the computer fills in polygons is called the fill rate.

Admittedly, this is a pretty ugly filled polygon, it has what's known as jaggies, those rough edges. This effect is less pronounced when using smaller pixels but nonetheless you see this in games all the time, especially in lower-powered platforms.

One method to soften this effect is anti-aliasing. Instead of filling pixels in a polygon with the same color, we can adjust the color based on how much the polygon cuts through each pixel. If a pixel is inside of a polygon, it gets fully colored. But if a polygon only grazes a pixel, it'll get a lighter shade. This feathering of the edges is much more pleasant to the eyes.

Anti-aliasing is used all over the place, including in 2d graphics like fonts and icons. If you're leaning too close to your monitor... closer... closer... you'll see all the fonts in your browser anti-aliased. So smooth.

In a 3D scene, there are polygons that are apart of objects, in the back, near the front, and just about everywhere. Only some are visible because some objects are hidden behind other objects in the scene, what's called occlusion. The most straightforward way to handle this is to use a sort algorithm, and arrange all the polygons in the scene from farthest to nearest, then render them in order. This is called the Painter's Algorithm because painters also have to start with the background, and increasingly work up to foreground elements.

Consider this scene with three overlapping polygons. To make things easier to follow, we're going to color our polygons differently. Also for simplicity, we'll assume these polygons are all parallel to the screen. Though in a real program like a game, the polygons can be tilted in 3D space. Our three polygons, A, B, and C, are at distance 20, 12, and 14.

The first thing the painter's algorithm does is sort all the polygons from farthest to nearest. Now that they're in order, we can use scan line rendering to fill each polygon one at a time. We start with polygon A, the furthest one away. Then we repeat the process for the next farthest polygon, in this case C. And then we repeat this again for Polygon B. Now we're all done, and you can see the ordering is correct. The polygons that are closer are in front.

An alternative method for handling occlusion is called Z-Buffering. It



3D Graphics: Crash Course Computer Science #27

Crash Course: Computer Science

<https://youtube.com/watch?v=TEAtmCYYKZA>

<https://nerdfighteria.info/v/TEAtmCYYKZA>

achieves the same output as before but with a different algorithm.

Let's go back to our previous example, before it was sorted. That's because this algorithm doesn't need to sort any polygons, which makes it faster. In short, z-buffering keeps track of the closest distance to a polygon for every pixel in the scene. It does this by maintaining a Z-Buffer, which is just a matrix of values that sits in memory.

At first, every pixel is initialized to infinity. Then Z-Buffering starts with the first polygon in its list. In this case, it's A. It follows the same logic as the scanline algorithm, but instead of coloring in pixels, it checks the distance of the polygon versus what's recorded in its Z-Buffer. It records the lower of the two values. For our Polygon A, with a distance of twenty, it wins against infinity every time.

When it's done with Polygon A, it moves on to the next polygon in its list, and the same thing happens. Now, because we didn't we didn't sort the polygons, it's not always the case that later polygons override high values. In the case of Polygon C, only some of the values in the Z-Buffer get new minimum distances.

This completed Z-Buffer is used in conjunction with a fancier version of scanline rendering that not only rests for line intersection, but also does a look up to see if that pixel will even be visible in the final scene. If it's not, the algorithm skips it and moves on.

An interesting problem arises when two polygons have the same distance. Like, if Polygon A and B are both at a distance of twenty, which one do you draw on top?

Polygons are constantly being shuffled around in memory and changing their access order. Plus rounding errors are inherent in floating point computations. So, which one gets drawn on top is often unpredictable. The result is a flickering effect called Z-fighting, which if you've played 3D games, you've no doubt encountered.

Speaking of glitches, another common optimization in 3D graphics is called Back-Face Culling. If you think about it, a triangle has two sides: a front and a back. With something like the head of an avatar, or the ground in a game, you should only ever see one side - the side facing outwards. So to save processing time, the back side of polygons are often ignored in the rendering pipeline, which cuts the number of polygon faces to consider in half.

This is great, except when there's a bug that lets you get inside of those objects and look outwards. Then the avatar head or ground becomes invisible.

===== Lighting (8:21) =====

Moving on. We need to talk about lighting - also known as shading - because if it's a 3D scene, the lighting should vary over the surface of objects. Let's go back to our teapot mesh.

With scanline rendering coloring in all the polygons, our teapot looks like this. Not very 3D. So let's add some lighting to enhance the realism. As an example, we'll pick three polygons from different parts of our teapot.

Unlike our previous examples, we're now going to consider how these polygons are oriented in 3D space. They're no longer parallel to the screen but rather tilted in different 3D directions. The direction they face is called the surface normal, and we can visualize that direction with a little 3D arrow that's perpendicular to the polygon's surface.

Now let's add a light source. Each polygon is going to be illuminated a different amount. Some will appear brighter because their angle

causes more light to be reflected towards the viewer. For example, the bottom-most polygon is tilted downwards—away from the light source, which means it's going to be dark.

In a similar way, the rightmost polygon is slightly facing away from the light, so it will be partially illuminated. And finally there's the upper-left polygon. Its angle means that it will reflect light from the light source towards our view, so it will appear bright.

If we do this for every polygon, our teapot looks like this, which is much more realistic. This approach is called flat shading, and it's the most basic lighting algorithm. Unfortunately, it also makes all those polygon boundaries really noticeable, and the mesh doesn't look smooth.

For this reason, more advanced lighting algorithms were developed, such as Gouraud Shading and Phong Shading. Instead of coloring in polygons just using one color, they vary the color across the surface in clever ways, which results in much nicer output.

===== Textures (9:56) =====

We also need to talk about textures, which in graphics, refers to the look of a surface, rather than its feel. Like with lighting, there are many algorithms with all sorts of fancy effects.

The simplest is texture mapping. To visualize this process, let's go back to our single polygon. When we're filling this in using scanline rendering, we can look up what color to use at every pixel, according to a texture image saved in memory. To do this, we need a mapping between the polygon's coordinates and the texture's coordinates. Let's jump to the first pixel that scanline rendering needs to fill in.

The texturing algorithm will consult the texture in memory, take the average color from the corresponding region, and fill the polygon accordingly. This process repeats for all pixels in the polygon, and that's how we get textures. If you combine all the techniques we've talked about this episode, you get a wonderfully funky little teapot. And this teapot can sit in an even bigger scene, comprised of millions of polygons.

===== Speeding Up the Process (10:47) =====

Rendering a scene like this takes a fair amount of computation. But importantly, it's the same type of calculations being performed over and over and over again for many millions of polygons - scanline filling, anti-aliasing, lighting, and texturing.

However, there are a couple of ways to make this much faster! First off, we can speed things up by having special hardware with extra bells and whistles just for these specific types of computations, making them lightning fast. And secondly, we can divide up a 3D scene into many smaller parts, and then render all the pieces in parallel, rather than sequentially.

CPU's aren't designed for this, so they aren't particularly fast. So, computer engineers created special processors just for graphics - a GPU, or Graphics Processing Unit. These can be found on graphics cards inside of your computer, along with RAM reserved for graphics.

This is where all the meshes and textures live, allowing them to be accessed super fast by many different cores of the GPU all at once. A modern graphics card, like a GeForce GTX 1080 Ti, contains 3584 processing cores, offering massive parallelization. It can process hundreds of millions of polygons every second!



3D Graphics: Crash Course Computer Science #27

Crash Course: Computer Science

<https://youtube.com/watch?v=TEAtmCYYKZA>

<https://nerdfighteria.info/v/TEAtmCYYKZA>

===== Review and Credits (11:57) =====

Okay, that concludes our whistle stop tour of 3D graphics. Next week, we switch topics entirely. I'll ping you then.

Crash Course Computer Science is produced in association with PBS Digital Studios. At their channel you can check out a playlist of shows like Gross Science, ACS Reactions, and The Art Assignment. This episode was filmed at the Chad and Stacy Emigholz Studio in Indianapolis, Indiana. And it was made with the help of all these nice people and our wonderful graphics team, Thought Cafe.

Thanks for watching, and try turning it off and then back on again.

[Theme music]