What are 3D shapes?

This tutorial is from Peter Collingridge and was originally posted on his website.

If you've been programming here on Khan Academy, you've probably drawn a whole lot of 2D shapes, like with rectangles and ellipses, and maybe you've wondered to yourself how to make 3D shapes, like cubes and spheres.

Well, the ProcessingJS library that we use here isn't really designed for 3D graphics, but as it turns out, with some trigonometry, we can write our own 3D graphics engine and, in doing so, learn a bit how 3D graphics work (and why there is a reason to learn trigonometry besides making math teachers happy!).

Here's an example of the sort of program you can make - use your mouse to rotate the donut around:

What are 3D graphics?

Since computer screens are essentially two-dimensional, 3D graphics are just 2D optical illusions that trick your brain into thinking it is seeing a 3D object. Here's a simple example:

Three parallelograms pretending to be a cube

A 3D graphics engine works by calculating what 2D shapes a 3D object would project on to the screen. So to write our own 3D engine, we need to know how to do these calculations. Our program won't be as quick as most 3D engines but it should help us understand the principles of how they work.

Representing shapes

A 3D graphics engine takes a 3D object and converts into 2D graphics, but how do we represent a 3D object in code?

A single point in 3D space is easy to represent using an array of three numbers. For example, we can use [30, 80, 55] to represent a point 30 pixels along the horizontal (x) axis, 80 pixels along the vertical (y) axis, and 55 pixels along the axis that goes into and out of the screen. Play around with the point below, rotating with the mouse and tinkering with the numbers:

Representing a line is also easy: you just connect two points. One way to represent an object in 3D, therefore, is by converting it into a group of lines. This is called a wireframe, as it looks like the object is made from wire. It's obviously not ideal for representing solid objects, but it's a good place to start.

Terms

Below are some terms that we will use when refering to 3D shapes. Other terms might be used elsewhere, but these ones are pretty popular.

- **Node**: a point represented by three coordinates, x, y and z (can also be called a vertex).
- **Edge**: a line connecting two points.
- **Face**: a surface defined by at least three points.
- **Wireframe**: a shape consisting of just nodes and edges.

Creating 3D Shapes

- Hide tutorial navigation
- A simple shape to start working with is a cube. Although a tetrahedron has fewer sides, its sides aren't orthogonal, which makes things trickier. Let's start by creating a 200×200×200 pixel cube, centred at the origin (0, 0, 0).
- We will start not with actually drawing anything, but with coming up with arrays of numbers that describe our shapes in 3D shape - specifically, arrays that describe our **nodes** and our **edges**.

Nodes

- We start by defining an array of nodes, where each node is an array of three digits, the x, y and z coordinates of that node:
- `var node0 = [-100, -100, -100];`
- `var node1 = [-100, -100, 100];`
- `var node2 = [-100, 100, -100];`
- `var node3 = [-100, 100, 100];`
- `var node4 = [ 100, -100, -100];`
- `var node5 = [ 100, -100, 100];`
- `var node6 = [ 100, 100, -100];`
- `var node7 = [ 100, 100, 100];`
- `var nodes = [node0, node1, node2, node3, node4, node5, node6, node7];`
- As you may have noticed, the nodes are all 8 ways of arranging three lots of positive or negative 100.
- You can see the nodes of a 2x2x2 cube centered at the origin in the visualization below. Rotate using the mouse:

Edges

- Next we define an array of edges, where each edge is an array of two numbers. For example, `edge0` defines an edge between `node0` and `node1`. We start counting at 0 because arrays are indexed starting at zero (To get the value of the first node we type `nodes[0]`).
- `var edge0 = [0, 1];`
- `var edge1 = [1, 3];`
- `var edge2 = [3, 2];`
- `var edge3 = [2, 0];`
- `var edge4 = [4, 5];`
- `var edge5 = [5, 7];`
- `var edge6 = [7, 6];`
- `var edge7 = [6, 4];`
- `var edge8 = [0, 4];`
- `var edge9 = [1, 5];`
- `var edge10 = [2, 6];`
- `var edge11 = [3, 7];`
- `var edges = [edge0, edge1, edge2, edge3, edge4, edge5, edge6, edge7, edge8, edge9, edge10, edge11];`
- The tricky part is making sure you join the right edges together. Here's a visualization of the edges we're connecting for a cube: