Let's Build a 3D Graphics Engine: Points, Vectors, and Basic Concepts

by Kyle Sloka-Frey 21 May 2013

Difficulty: Intermediate Length: Medium Languages: English

Platform Agnostic | HTML5 | Programming | Simulation | 3D Games | Mathematics

This post is part of a series called Let's Build a 3D Graphics Software Engine.

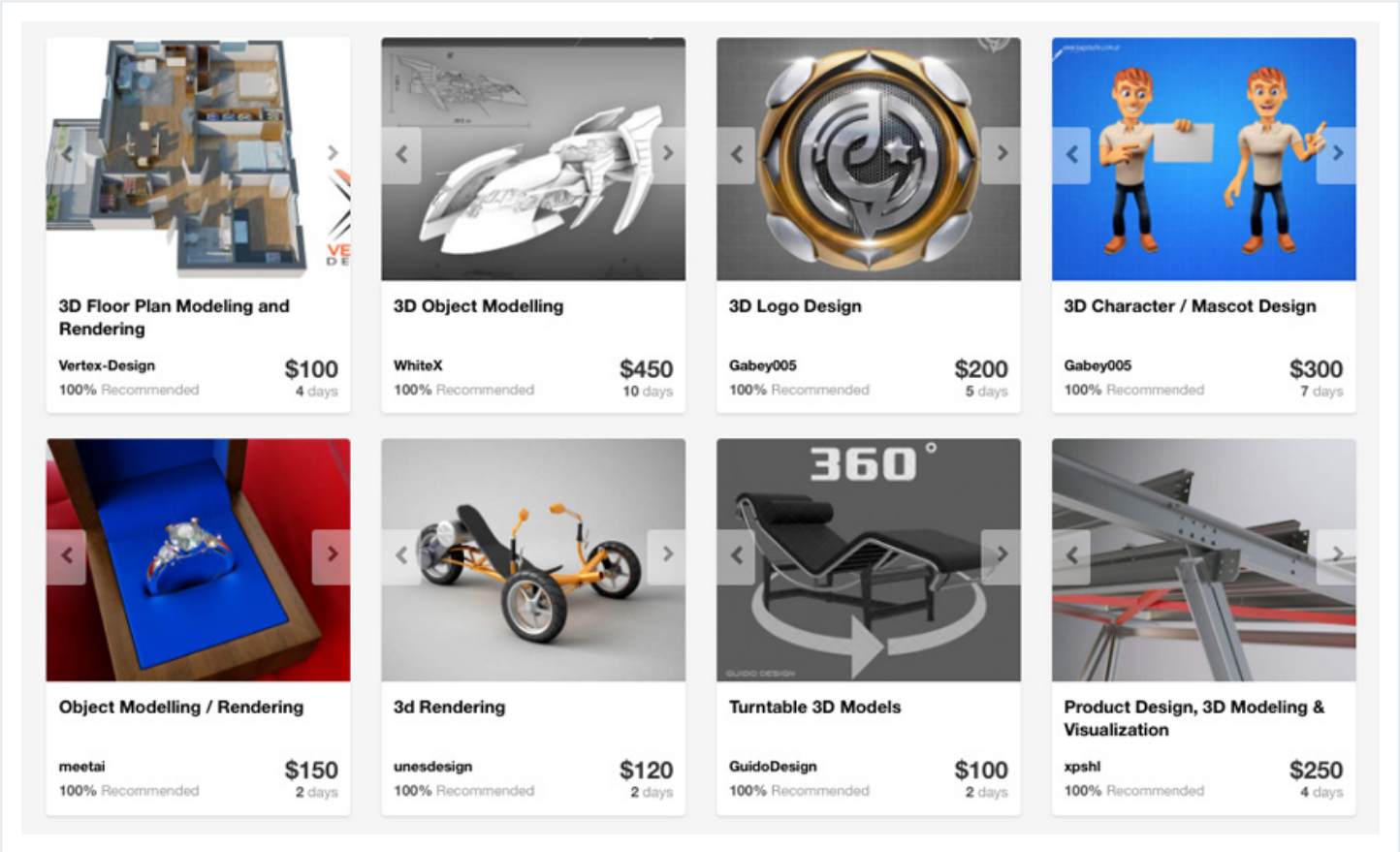
Let's Build a 3D Graphics Engine: Linear Transformations

The 3D game engines that are behind today's biggest games are staggering works of mathematics and programming, and many game developers find that understanding them in their entirety is a difficult task. If you are lacking in experience (or a college degree, like myself), this task becomes even more arduous. In this series, I aim to walk you through the basics of graphics systems in 3D engines.

More specifically, in this tutorial we will be discussing points and vectors, and all of the fun that comes with them. If you have a basic grasp of algebra (variables and variable math) and Computer Science (the basics of any object-oriented programming language), you should be able to make it through most of these tutorials, but if you're having trouble with

any of the concepts, please ask questions! Some of these topics can be terribly difficult.

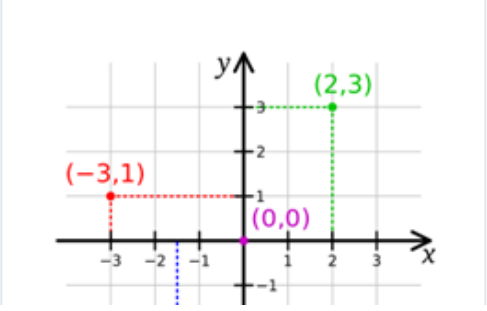
You can also get extra help over on Envato Studio, where you can find plenty of fantastic 3D Design & Modeling services for affordable prices.

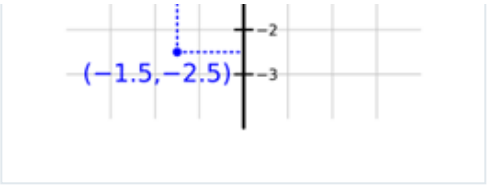


3D Design & Modeling services on Envato Studio

Basics of Coordinate Systems

Let's start from the basics. Three-dimensional graphics require the concept of a three-dimensional space. The most widely used of these spaces is called the Cartesian Space, which gives us the benefit of Cartesian coordinates (the basic (x, y) notations and 2D grid-spaced graphs that are taught in most high schools).

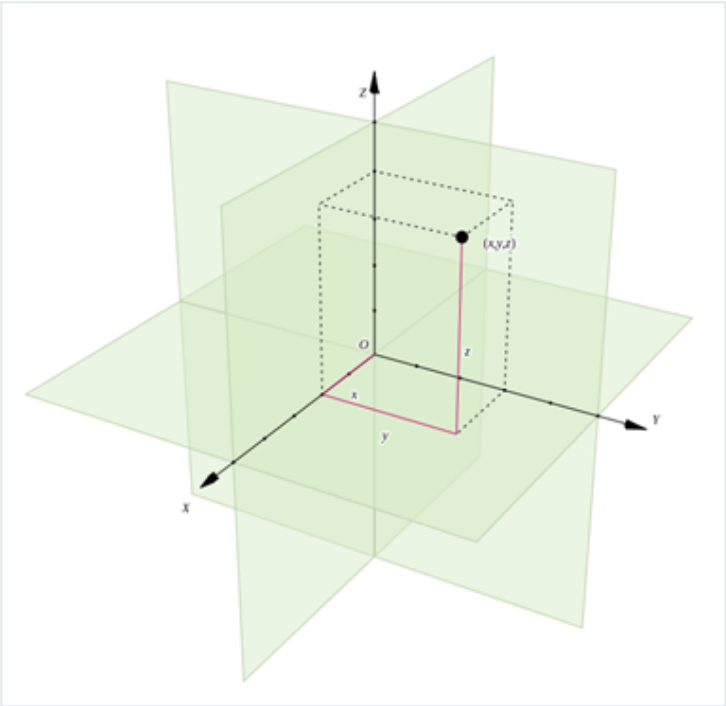




Pictured: the bane of many high schoolers' existence.

3-dimensional Cartesian space gives us an x, y, and z axis (describing position based on horizontal placement, vertical placement, and depth respectively). The coordinates for any point within this space are shown as a *tuple* (in this case a 3-tuple, since there are three axes). On a 2-dimensional plane, a tuple could be depicted as (x, y), and in 3-dimensional plane, it is depicted as (x, y, z). The use of this 3-tuple is that it shows a point's location relative to the space's origin (which itself is typically shown as $(0, 0, 0)$).

Tip: Tuple: an ordered list (or sequence) of elements in computer science or mathematics. So, (K, y, l, e) would be a 4-tuple, showing a sequence of characters that make up my name.



Within this space, we are going to define a point as a representation of a 3-tuple. This can also be shown as:

$$P = (x, y, z)$$

In addition to this definition of a point, we must define its parts.

Each of the elements within this 3-tuple is a *scalar* (number) that defines a position along a *basis vector*. Each basis vector must have a unit length (that is, a length of exactly 1), so 3-tuples such as $(1, 1, 1)$ and $(2, 2, 2)$ could not be basis vectors as they are too long.

We define three basis vectors for our space:

$$X = (1, 0, 0)$$
$$Y = (0, 1, 0)$$
$$Z = (0, 0, 1)$$

Source: https://www.thefullwiki.org/Arithmetics/Cartesian_Coordinate.

The Coordinate System

Now let's talk about the mathematical definition of our coordinate system, how it affects our graphics system, and the calculations we can make.

Representing Points

The *origin point* of our coordinate system can be depicted as the point O, which represents the 3-tuple (0,0,0). This means that the mathematical representation of our coordinate system can be depicted as:

$$\{O; X, Y, Z\}$$

By this statement, you can say that (x, y, z) represents a point's position in relation to the origin. This definition also means that any point P, (a, b, c), can be represented as:

$$P = O + aX + bY + cZ$$

From here on, I will reference scalars in lower-case and vectors in upper-case - so a, b, and c are scalars, and X, Y, and Z are vectors. (They are actually the basis vectors we defined earlier.)

This means that a point whose tuple is (2,3,4) could be represented as:

$$\begin{aligned}(2,3,4) &= (2,0,0) + (0,3,0) + (0,0,4) \\ &= (0,0,0) + (2,0,0) + (0,3,0) + (0,0,4) \\ &= (0,0,0) + 2(1,0,0) + 3(0,1,0) + 4(0,0,1) \\ &= O + 2X + 3Y + 4Z\end{aligned}$$

So, we've taken this abstract concept of "a point in 3D space" and defined it as four separate objects added together. This kind of definition is very important whenever we want to put any concept into code.

Mutually Perpendicular

The coordinate system that we will be using also has the valuable property of being *mutually perpendicular*. This means that there is a 90 degree angle between each of the axes where they meet on their respective planes.

Our coordinate system will also be defined as "right-handed":

Source: http://viz.aset.psu.edu/gho/sem_notes/3d_fundamentals/html/3d_coordinates.html.

In mathematical terms, this means that:

$$X = Y \times Z$$

...where $\times$ represents the cross product operator.

In case you aren't sure what a cross product is, it can be defined by the following equation (assuming you are given two 3-tuples):

$$(a, b, c) \times (d, e, f) = (bf - ce, cd - af, ae - bd)$$

These statements might seem tedious, but later on, they will allow us to do a variety of different calculations and transformations much more easily. Luckily, you don't have to memorize all of these equations when building a game engine - you can simply build from these statements, and then build even less complicated systems on top of that. Well, until you have to edit something fundamental within your engine and have to refresh yourself on all of this again!

Points and Vectors

With all of the basics of our coordinate system locked down, its time to talk about points and vectors and, more importantly, how they interact with one another. The first thing to note is that points and vectors are distinctly different things: a point is a physical location within your space; a vector is the space between two points.

To make sure that the two don't get confused, I'll write points in capital italics, as P, and vectors in capital boldface, as $\mathbf{V}$.

There are two main axioms that we are going to deal with when using points and vectors,

and they are:

- Axiom 1: The difference of two points is a vector, so $\mathbf{V} = P - Q$
- Axiom 2: The sum of a point and a vector is a point, so $Q = P + \mathbf{V}$

Tip: An *axiom* is a point of reasoning, often seen as evident enough to be accepted without argument.

Building the Engine

With these axioms stated, we now have enough information to create the building block classes that are at the heart of any 3D game engine: the `Point` class and the `Vector` class. If we were going to build our own engine using this information, there would be some other important steps to take when creating these classes (mostly having to do with optimization or dealing with already existing APIs), but we are going to leave these out for the sake of simplicity.

The class examples below are all going to be in pseudocode so that you can follow along with your programming language of choice. Here are outlines of our two classes:

```
Point Class
{
        Variables:
                num tuple[3]; //(x,y,z)
```

```
        Operators:
                Point AddVectorToPoint(Vector);
                Point SubtractVectorFromPoint(Vector);
                Vector SubtractPointFromPoint(Point);

        Function:
                //later this will call a function from a graphics API, bu
                //this should just be printing the points coordinates to
                drawPoint;
}

Vector Class
{
        Variables:
                num tuple[3]; //(x,y,z)

        Operators:
                Vector AddVectorToVector(Vector);
                Vector SubtractVectorFromVector(Vector);
}
```

As an exercise, try to fill in each of these class's functions with working code (based on what we've gone over so far). Once you've gotten all of that finished, put it to the test by making this example program:

```
main
{
    var point1 = new Point(1,2,1);
    var point2 = new Point(0,4,4);
    var vector1 = new Vector(2,0,0);
    var vector2;

    point1.drawPoint(); //should display (1,2,1)
    point2.drawPoint(); //should display (0,4,4)
```

```
    vector2 = point1.subtractPointFromPoint(point2);

    vector1 = vector1.addVectorToVector(vector2);


    point1.addVectorToPoint(vector1);
    point1.drawPoint(); //should display (4,0,-2)

    point2.subtractVectorFromPoint(vector2);
    point2.drawPoint(); //should display (-1,6,7)
}
```

Conclusion

You made it to the end! It may seem like an awful lot of math to only make two classes - and it definitely is. In most cases you will never have to work on a game at this level, but having an intimate knowledge of the workings of your game engine is helpful nonetheless (if only for the self-satisfaction).

If you thought that this was fun, then make sure to check out my next tutorial on graphics system basics: transformations!

Of course, once you've created the graphics engine, you'll need to add some content, so feel free to check out the wide selection of game assets and 3D models on Envato Market.

Kyle Sloka-Frey

Kyle Sloka-Frey

Data Scientist, Father, Game Dev, Developer, Futurist

Kyle is a data scientist, father, game dev, and all around analytics and organization lover. He enjoys space and all things Futurism.

kyleslokafrey

50 Comments Gamedevtuts+ Login

Recommend 7 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS

Name