



GAME DEVELOPMENT > PROGRAMMING

Let's Build a 3D Graphics Engine: Spaces and Culling

Advertisement

by [Kyle Sloka-Frey](#) 29 May 2013Difficulty: Intermediate Length: Medium Languages:

Programming

Platform Agnostic

HTML5



This post is part of a series called [Let's Build a 3D Graphics Software Engine](#).


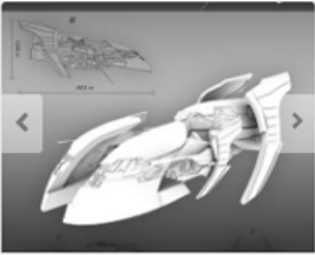


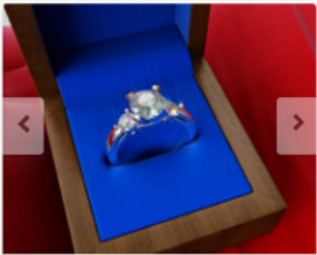



◀ [Let's Build a 3D Graphics Engine: Linear Transformations](#)

▶ [Let's Build a 3D Graphics Engine: Rasterizing Line Segments and Circles](#)

Welcome! This is the third part of [our series on 3D graphics engines](#). If you made it this far into the series, you'll be glad to know that this piece will be much lighter on the math aspect of 3D engines, and instead will take a focus on more practical things - in particular, adding a camera and a basic rendering system.

Tip: If you haven't read [the first two parts](#) yet, I highly suggest that you do before continuing.

You can also get extra help over on Envato Studio, where you can choose from a wide range of high-quality [3D Design & Modeling services](#) from experienced providers.

 3D Floor Plan Modeling and Rendering Vertex-Design 100% Recommended \$100 4 days	 3D Object Modeling WhiteX 100% Recommended \$450 10 days	 3D Logo Design Gabey005 100% Recommended \$200 5 days	 3D Character / Mascot Design Gabey005 100% Recommended \$300 7 days
 Object Modelling / Rendering meetai 100% Recommended \$150 2 days	 3d Rendering unesdesign 100% Recommended \$120 2 days	 Turntable 3D Models GuidoDesign 100% Recommended \$100 2 days	 Product Design, 3D Modeling & Visualization xpshl 100% Recommended \$250 4 days

[3D Design & Modeling services on Envato Studio](#)

Recap

First off, let's take a look at the classes that we've created so far:

```

01 Point Class
02 {
03     Variables:
04         num tuple[3]; //(x,y,z)
05
06     Operators:
07         Point AddVectorToPoint(Vector);
08         Point SubtractVectorFromPoint(Vector);
09         Vector SubtractPointFromPoint(Point);
10         Null SetPointToPoint(Point); // move point to specified point
11
12     Functions:
13         drawPoint; //draw a point at its position tuple
14 }
15
16

```

```
17 Vector Class
18 {
19     Variables:
20         num tuple[3]; //(x,y,z)
21
22     Operators:
23         Vector AddVectorToVector(Vector);
24         Vector SubtractVectorFromVector(Vector);
25         Vector RotateXY(degrees);
26         Vector RotateYZ(degrees);
27         Vector RotateXZ(degrees);
28         Vector Scale(s0,s1,s2); //params: scaling along each axis
}
```

Using these two classes on their own has proven a little messy thus far, and drawing every possible point can drain your system's memory fairly quickly. To solve these problems, we are going to introduce a new class into our game engine: the *camera*.

Our camera is going to be where all our rendering happens, exclusively; its going to *cull* all of our objects to the screen, and it is also going to manage a list of all of our points.

But before we can get to all of that, we must first talk a little bit about culling.

London Culling

Culling, by definition, is the selection of objects from a larger group of objects. For our game engine, the small selection that we take will be the points that we want to draw to the screen. The larger group of objects will be every point that exists.

Doing this drastically reduces your engine's drain on a system's memory, by drawing only what a player is actually able to see, rather than an entire world's worth of points. In our engine, we are going to do this by setting parameters for a *view space*.

Our view space will be defined across all three of the traditional axes: x, y, and z. Its x definition will consist of everything between the window's left and right boundaries, its y definition will consist of everything between the window's top and bottom boundaries, and its z definition will be between (where the camera is set) and our player's view distance (for our demonstration, we will be using an arbitrary value of).

Prior to drawing a point, our camera class is going to check to see whether that point lies within our view space. If it does, then the point will be drawn; otherwise, it will not.

Advertisement

Can We Get Some Cameras in Here?

With that basic understanding of culling, we can deduce that our class will look like this, so far:

```
1 Camera Class
2 {
3     Vars:
4         int minX, maxX; //minimum and maximum bounds of X
5         int minY, maxY; //minimum and maximum bounds of Y
6         int minZ, maxZ; //minimum and maximum bounds of Z
7 }
```

We are also going to have our camera handle all of the rendering for our engine as well. Depending on the engine, you will find that renderers are often separated from the camera systems. This is typically done to keep the systems encapsulated nicely, since - depending on the scope of your engine - the two could get quite messy if kept together. For our purposes, though, it will be simpler to treat them as one.

First, we're going to want a function that can be called externally from the class that will draw the scene. This function will cycle through each of the points that exist, compare them to the camera's culling parameters, and draw them if applicable.

Source: <http://en.wikipedia.org/wiki/File:ViewFrustum.svg>

Tip: If you wanted to separate your camera system from your renderer, you could simply create a `Renderer` class, have the camera system cull the points, store the ones to be

drawn in an array, and then send that array to the `draw()` function of your renderer.

Point Management

The final piece of our camera class is going to be its point management system. Depending on the programming language you are using, this could just be a simple array of all of the objects that can be drawn (we will be handling more than just points in later parts). Alternatively, you may have to use the language's default object parent class. If you are super unlucky, you will have to create your own object parent class and have each drawable class (so far only points) be a child of that class.

After adding that into the class, a basic overview of our camera would look like this:

```
01 | Camera Class
02 | {
03 |     Vars:
04 |         int minX, maxX; //minimum and maximum bounds of X
05 |         int minY, maxY; //minimum and maximum bounds of Y
06 |         int minZ, maxZ; //minimum and maximum bounds of Z
07 |         array objectsInWorld; //an array of all existent objects
08 |     Functions:
09 |         null drawScene(); //draws all needed objects to the screen, does not return any
10 | }
```

With these additions, let's improve a little upon the program that we made last time.

Bigger and Better Things

We are going to create a simple point drawing program, with the sample program that we created [last time](#) as a starting point.

In this iteration of the program, we are going to add in the use of our new camera class. When the **D** key is pressed, the program will redraw the screen without culling, displaying the number of objects that were rendered in the top right-hand corner of the screen. When

the **C** key is pressed, the program will redraw the screen with culling, also displaying the number of rendered objects.

Let's take a look at the code:

```

01  main{
02      //setup for your favorite Graphics API here
03      //setup for keyboard input (may not be required) here
04
05  var camera = new Camera(); //create an instance of the camera class
06  camera.objectsInWorld[100]; //create 100 object spaces within the camera's array
07
08  //set the camera's view space
09  camera.minX = 0;
10  camera.maxX = screenWidth;
11  camera.minY = 0;
12  camera.maxY = screenHeight;
13  camera.minZ = 0;
14  camera.maxZ = 100;
15
16      for(int x = 0; x < camera.objectsInWorld.length; x++)
17      {
18          //Set its location to a random point on the screen
19          camera.objectsInWorld[x].tuple = [random(-200,1000), random(-200,1000), random(
20      }
21
22  function redrawScreenWithoutCulling() //this function clears the screen and then dr
23  {
24      ClearTheScreen(); //use your Graphics API's clear screen function
25      for(int x = 0; x < camera.objectsInWorld.length; x++)
26      {
27          camera.objectsInWorld[x].drawPoint(); //draw the current point to the scr
28      }
29  }
30
31  while(esc != pressed) // the main loop
32  {
33      if(key('d') == pressed)
34      {
35          redrawScreenWithoutCulling();
36      }
37      if(key('c') == pressed)
38      {
39          camera.drawScene();
40      }
41      if(key('a') == pressed)
42      {
43          Point origin = new Point(0,0,0);
44          Vector tempVector;
45          for(int x = 0; x < camera.objectsInWorld.length; x++)
46          {
47              //store the current vector address for the point, and set the point
48              tempVector = camera.objectsInWorld[x].subtractPointFromPoint(origin);
49              //reset the point so that the scaled vector can be added

```

```

50         camera.objectsInWorld[x].setPointToPoint(origin);
51         //scale the vector and set the point to its new, scaled location
52         camera.objectsInWorld[x].addVectorToPoint(tempVector.scale(0.5,0.5,0.5)
53     }
54 }
55 if(key('s') == pressed)
56 {
57     Point origin = new Point(0,0,0); //create the space's origin as a point
58     Vector tempVector;
59     for(int x = 0; x < camera.objectsInWorld.length; x++)
60     {
61         //store the current vector address for the point, and set the point
62         tempVector = camera.objectsInWorld[x].subtractPointFromPoint(origin);
63         //reset the point so that the scaled vector can be added
64         camera.objectsInWorld[x].setPointToPoint(origin);
65         //scale the vector and set the point to its new, scaled location
66         camera.objectsInWorld[x].addVectorToPoint(tempVector.scale(2.0,2.0,2.0)
67     }
68 }
69 if(key('r') == pressed)
70 {
71     Point origin = new Point(0,0,0); //create the space's origin as a point
72     Vector tempVector;
73     for(int x = 0; x < camera.objectsInWorld.length; x++)
74     {
75         //store the current vector address for the point, and set the point
76         tempVector = camera.objectsInWorld[x].subtractPointFromPoint(origin);
77         //reset the point so that the scaled vector can be added
78         camera.objectsInWorld[x].setPointToPoint(origin);
79         //scale the vector and set the point to its new, scaled location
80         camera.objectsInWorld[x].addVectorToPoint(tempVector.rotateXY(15));
81     }
82 }
83 }
84 }

```

Now you can see, first-hand, the power of culling! Do note that if you are looking through the sample code, some things are done a bit differently in order to make the demos more web-friendly. (You can [check out my simple demo here.](#))

Advertisement

Conclusion

With a camera and rendering system under your belt, you can technically say that you've created a 3D game engine! It may not be overly impressive just yet, but it's on its way.

In our next article, we'll be looking at adding some geometric shapes to our engine (namely line segments and circles), and we will talk about the algorithms that can be used to fit their equations to the pixels of a screen.

Advertisement



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](#)