**Basic 3D Graphics in Processing**

**Carson Foster**

In this lesson, we’ll be looking at the (very) basics of using 3D graphics in Processing. The sections will cover: how to first use the 3D renderer, how to draw basic shapes in 3D, lights and cameras, and materials.

The 3D Renderer

In order to draw in 3D, you must first tell Processing that you will be using the 3D renderer. This is similar to exporting sketches to PDF or SVG, as you had to use a custom renderer there as well. This is how it works:



You can replace the width and height with the constants of your choice, but the last argument must be typed as shown in order for the 3D renderer to be used.

Basic Shapes

Now that the 3D renderer has been chosen, you are actually able to draw things in 3D now. However, the first thing to note is that, in 3D, you have to give up some fine control over exactly where things appear to be positioned, as some of the 2D space on the screen is used to give an illusion of a 3D drawing. Next, you’re likely going to have to use transformations if you are drawing with 2D shapes, and you’re definitely going to have to use transformations with the two 3D primitive shapes. Take, for example, the first 3D primitive:



Notice that both overloaded versions do not have arguments to represent the location of the box, only the size of the box. The first version creates a cube with the given side length, and the second version creates a rectangular prism with the given width, height, and depth, but neither allows you to declare a location for it. Thus, you are forced to use transformations to position the 3D primitive shapes.

In 3D, the transformations change only slightly. Let’s just run through the transformations and how they are different in 3D really quickly:



Translation is pretty similar, as you just add a z coordinate to the function. For further reference, the positive z direction is coming out from the screen (toward you), and the negative z direction is going into the screen (away from you).



For rotation, each axis gets its own function, which rotates **around** that axis. To help visualize this, grab a pencil. Hold the pencil in your right hand’s closed fist. Stick your right arm straight out in front of you so that your palm would face down. Bend your elbow to the left toward your body so that your arm forms a right angle. Rotate your forearm (do not move your wrist). The pen is rotating around the x axis. Now, put your hand palm down again, and rotate your arm up so that your palm faces to the left (keeping the right angle of your arm). Rotate your forearm again (do not move your wrist) to see the pen rotate around the y axis. Stick your arm out straight again (palm down). Rotate your forearm (do not move your wrist). The pen is (can you guess?) rotating around the z axis. For those of you who are familiar with flying/flight games, rotating around the x axis is pitch, rotating around the y axis is yaw, and rotating around the z axis is roll.



For scaling, it’s much the same as in 2D: you can either scale all dimensions by the same scale factor, or scale each dimension separately.

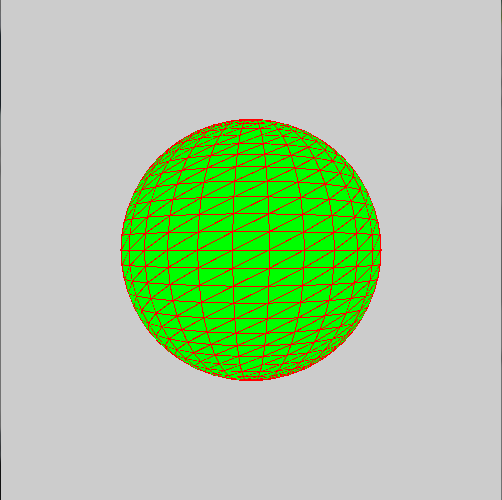
Our final 3D primitive is a sphere. The associated function is extremely simple:

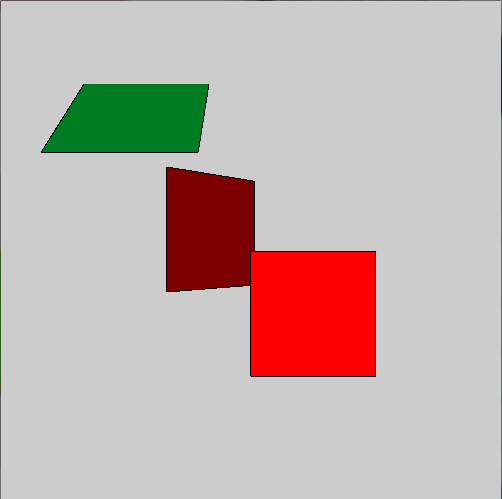


As you can see, the only thing you can change is the radius. Use transformations to set its location.

It’s also worth noting in this section that you can change the fill and stroke of 3D objects with the same functions as in 3D (but stroke can look weird for spheres):



As you can see, the stroke for spheres is entirely composed of the tessellated triangles that Processing uses to create the sphere. If you don’t want to see these lines, don’t use a stroke when drawing a sphere.

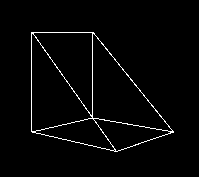
 Also worthy of note is that you can still draw 2D shapes in 3D, they just appear as 2D in the drawing (so they have no depth, as if they were drawn and cut out on an infinitely thin piece of printer paper). Here’s what some 2D rectangles in 3D look like:



Custom Shapes

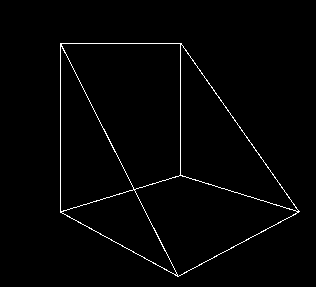
If you want to create a custom shape in 3D, you have two options: first, you can use a variety of 2D and 3D shapes composited together, or you can define a shape by its vertices. Both of these are cumbersome and annoying, but we’ll look at one example of each.

Defining a shape by its vertices is pretty much the same as in 2D, except that there’s also a z argument. Let’s look at an example:



The picture is cropped to just include the shape.

Here’s the equivalent using individual shapes:



Have fun deciding which version you hate less.

Light and the Camera

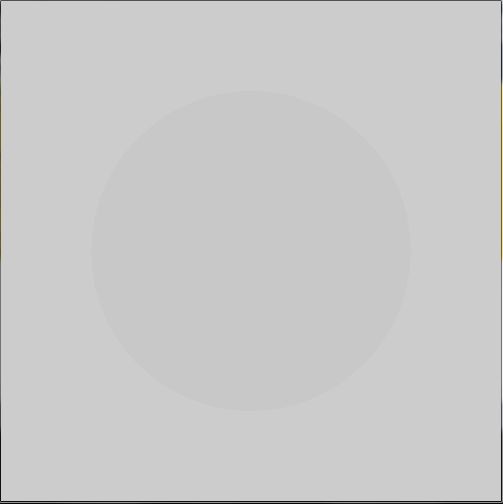
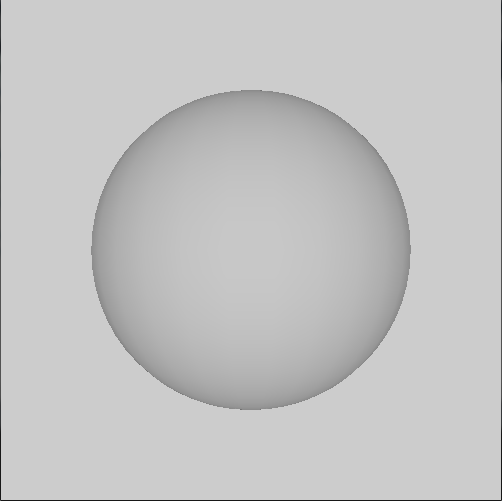
The first concept in this section is pretty simple: lights. ‘Lights’ in Processing refers to the same thing as ‘lights’ in most other contexts: things that project light onto other objects. In 3D, this means that we can have spotlights, ambient lighting, etc. All of the following functions must be placed in the draw loop to maintain their effects.

First, you can set the default lighting with one simple command. For details on what the exact values of default lighting are, check the Processing reference.



Here’s how adding the default lighting can change a sketch:

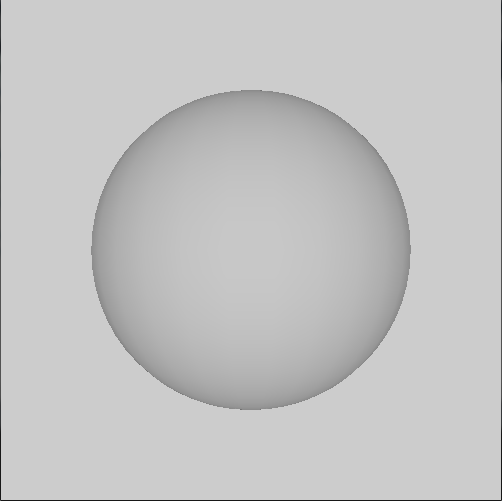


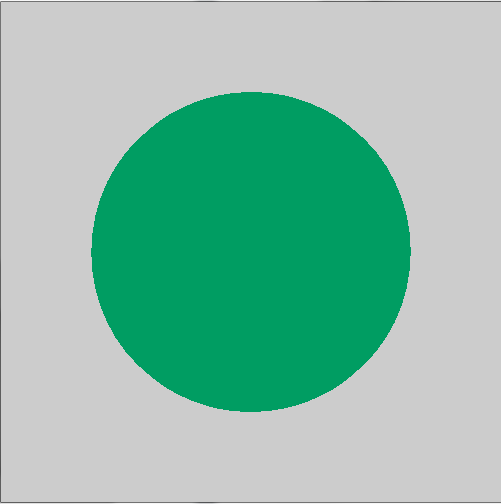
It’s pretty hard to even see the left version (with no lights), but once we uncomment the commented line and enable the default lighting, the sphere is more visible and appears more realistic.

The next lighting function adds an ambient light to the sketch, which does not come from a specific direction, and usually does not emanate from a set point.



As you can see, each ambient light has a certain color, and you can place it at a set point (although I’m not sure why you would).

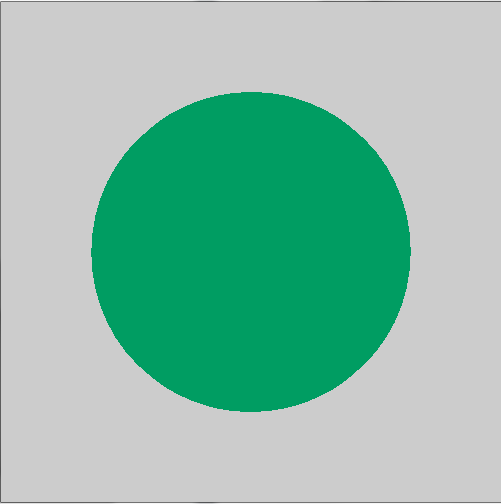
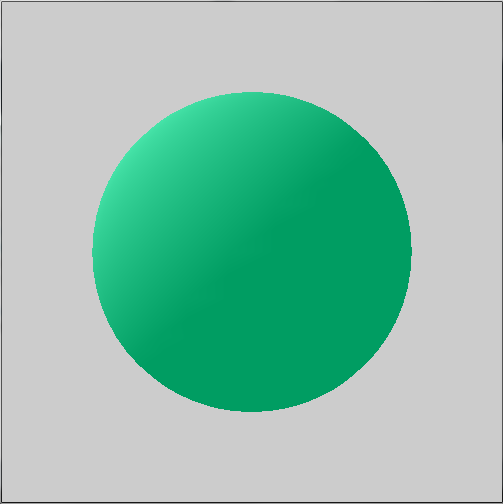
 Here’s a comparison between the sphere with the default lighting & a sphere with only green ambient lighting:

As you can see, ambient lighting essentially only changes the color of objects in the sketch; it doesn’t provide any sort of directional light, which we’ll look at next.

Directional lighting is just that: directional. It is light that has a certain color, like ambient light, but only comes from a certain direction (and isn’t limited to a cone shape, like a flash light in real life is).



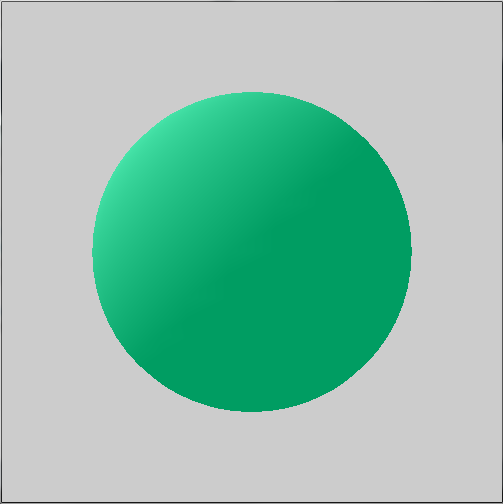
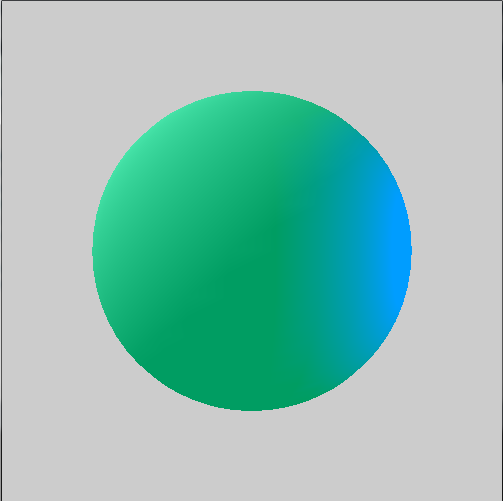
The last three argument represent the direction of the light: for each argument, put either a positive one, zero, or negative one. This represents whether the light shines in the positive or negative direction for that axis. For example, if the last three arguments are one, negative one, and negative one, then the light will be pointing in the positive x direction (right), negative y direction (up), and negative z direction (into the screen). Therefore, the light will be emanating from the bottom left (and somewhat out of the screen).

 Here is a comparison of the ambient lighting only, and the same ambient lighting with a soft directional lighting (towards the bottom right).

As you can see, the sphere is in the way of the light, meaning that only part of the sphere is illuminated.

Our next type of light is a point light, which emanates from a point, but also isn’t restrained into a specific cone shape (if you do want a cone shape, look at spotLight in the Processing reference; we won’t discuss it).



Alright, let’s move straight into our comparison, this time between the ambient + directional light sphere and a new ambient + directional + point light sphere, with the point light being off to the right side (and a little out of the screen) & blue.

The final lighting function we’ll discuss has to do with highlights, and it’s beyond the scope of this lesson to explain in detail what this does, so there will only be a simple explanation. This property of light, called specular light, is applied to light functions after it in the code (like fill or stroke), and changes how highlights appear. The format for the function is:



The most common form is just using white specular light, but feel free to toy with the values to achieve an effect you like.

Camera

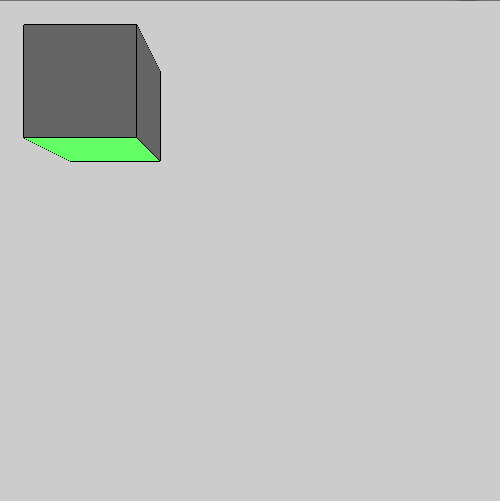
Before we can learn how to manipulate the camera, we first have to understand what the camera is. Essentially, the ‘camera’ in Processing is your point of view into the virtual 3D world. When Processing renders in 3D, you can think of it creating an imaginary 3D space, in which you can then put objects at certain coordinates. What you see, however, depends on the position and direction of your eyes in the 3D space. The camera in Processing acts as your eyes in the 3D space. Thus, by changing the position & direction of the camera, you can change what is displayed, without actually changing anything else about what you draw.

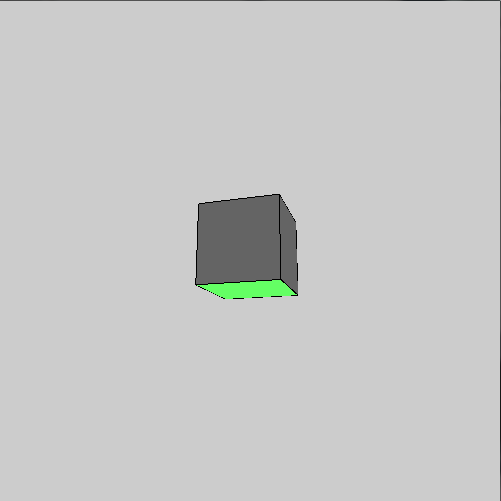
The first option to change the camera settings is this function:



The first version simply resets the camera settings to the default. The second version is more complicated. The first three arguments describe where the camera is in the 3D space. The next three arguments describe where in the 3D space the camera is pointing. The last three arguments are used for changing which direction is considered up, and their possible values are zero, negative one, and positive one. They aren’t very useful arguments, so I won’t cover them in detail. However, you should know that the default values (setting the negative y direction as up) are zero, one, zero (respectively).



Here, I’ve created a box, some light ambient lighting, and a directional light that colors the bottom of the box green. Here’s what that looks like originally (on the left), and what happens when we uncomment the line with the camera function (on the right).

In the uncommented line, I moved the camera farther away and focused it on the box, so that the box appears smaller and in the center. Despite this, I didn’t move the box at all or change its size; I just changed the properties of the camera.

The next way of manipulating the camera is quite different:



Between the begin and end functions, any transformations applied will affect the camera: therefore, you can translate, rotate, etc. the camera all you want to provide fine grain control. Remember to put this block at the beginning of your draw function if you want to use it. Most of the time, the plain camera function will be acceptable, however.

Materials

In Processing, you can also set a few of the ‘material’ properties of shapes. This refers to the properties of the material/substance a sphere/box/etc. is made of. For example, you can change the shininess of objects to either make them reflect more or less light. These functions are like fill and stroke, in that they apply to the shapes drawn after their function calls.

Our first material function changes how the material reflects ambient light:



As you can see, you can pass a color variable, a grayscale number, or 3 numbers representing an RGB color. These parameters determine how much of that color light is reflected by the material. For example, if you called ambient(255, 0, 64), then the following materials would reflect all of the red light, none of the green light, and about a quarter of the blue light. Remember, what you see is the reflected light (not the absorbed light).

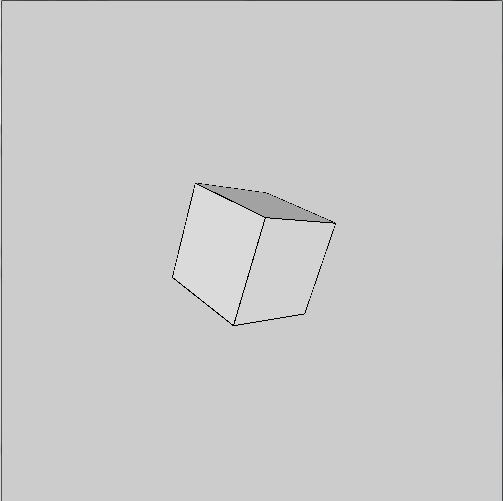
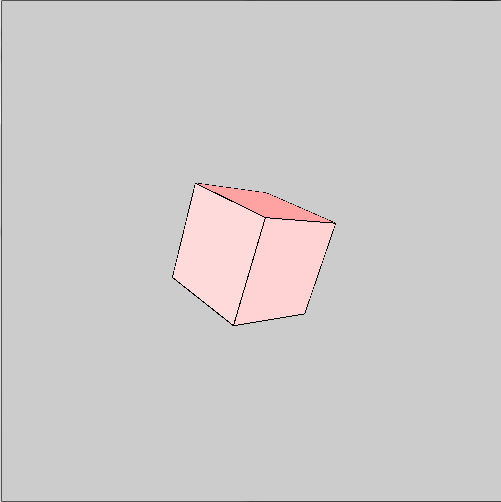
Our next function changes the emissive lighting of a material. This type of lighting refers to light that the material itself emits, which causes it to appear to illuminate itself.



The parameters are the same as the last function, allowing this function to accept a color variable, a grayscale value, or a set of 3 numbers representing an RGB color.

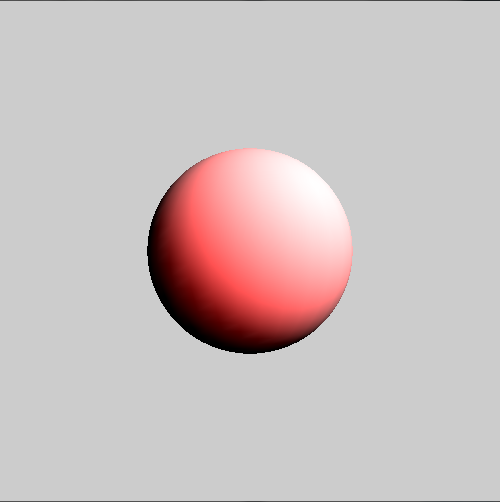


Once more, let’s look at a comparison: before we uncomment on the left, and after we uncomment on the right:

As you can see, the emissive material property causes the material itself to provide that color, even if another light (or the fill of the object) doesn’t.

The next material property that you can control in Processing is the specular color of the material. This property combines with the specular color of the light to produce a highlight. The format is the exact same as the other material functions we’ve looked at already:



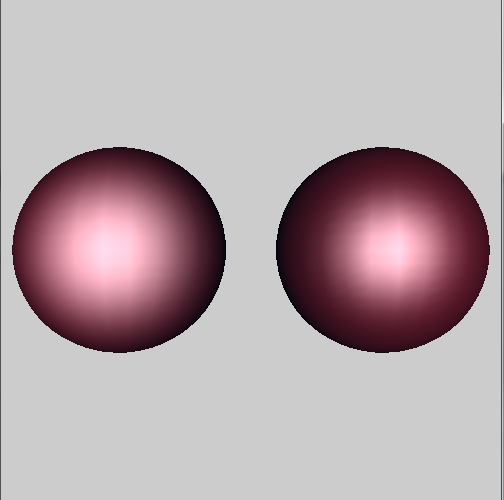
Here’s an example in which I combine a red material specular color with white specular lighting (and the light is also white).



The final material property we will discuss is shininess. This property is pretty simple: put in a low value to cause the material to reflect less light, and put in a higher value to make the material reflect more light.



Here’s the obligatory example:



Well, that’s going to wrap up the last lesson! I hope that you’ve learned a lot about Processing, and that you’ve had as much fun as I have.