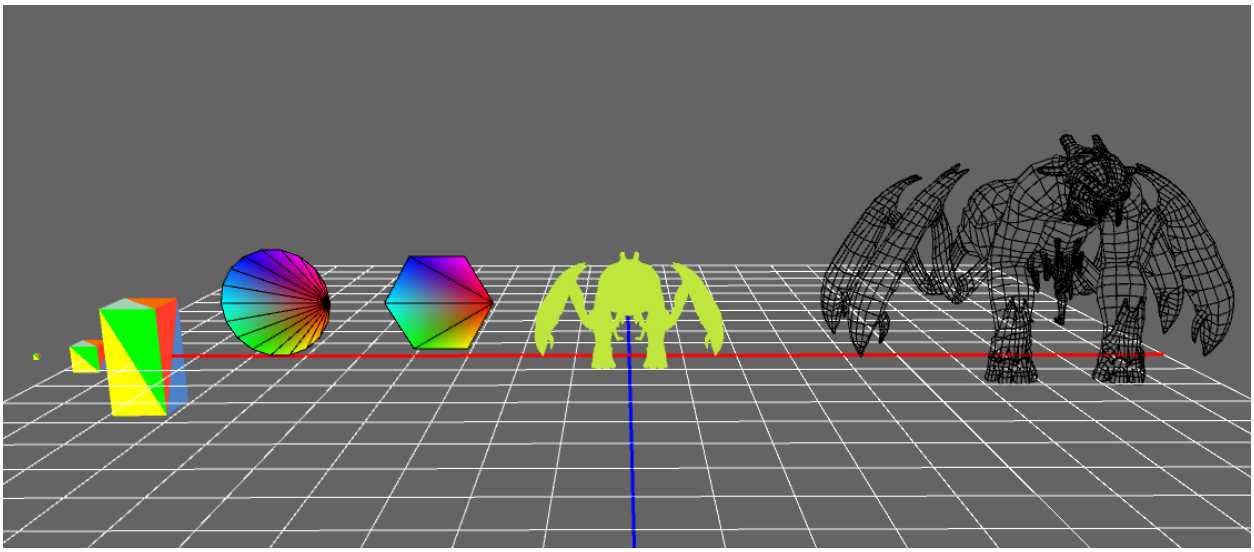
Project 3 – 3D Rendering

Overview

For this project, you are going to create, load and render some basic 3D models. In addition, you will create a simple orbit camera to help navigate a 3D scene.

Description

In the end, your program will look something like the following:



Setup

When rendering in 3D, you have to change the size() function that you use to start up a Processing application, by adding P3D as a third parameter:
size(1600, 1000, P3D);

Perhaps the most important aspect of any 3D environment is the camera; without one (even a simple one), you can't see anything.

Camera Class

The camera affects how the user views the information, and the implementation details of a camera system can make or break an application. In this assignment your camera is going to be a simple **orbit camera** which rotates around, and looks at, a specific point (an x, y, z location). This will be accomplished by using **spherical coordinates**, which is very similar to the concept of polar coordinates, just with an additional component to calculate.

Whether we're talking about Processing, DirectX, or any other rendering system, you will need two main pieces of information for a camera. A **projection** matrix, and a **view** matrix. Most graphics APIs have functions to allow the programmer to easily create these, and Processing is no different.

Projection Matrix

The projection matrix can be created (and set for you, behind the scenes), by calling this function:

```
perspective(radians(50.0f), width/(float)height, 0.1, 1000);
```

Details about the parameters of this function can be found here:
https://processing.org/reference/perspective_.html

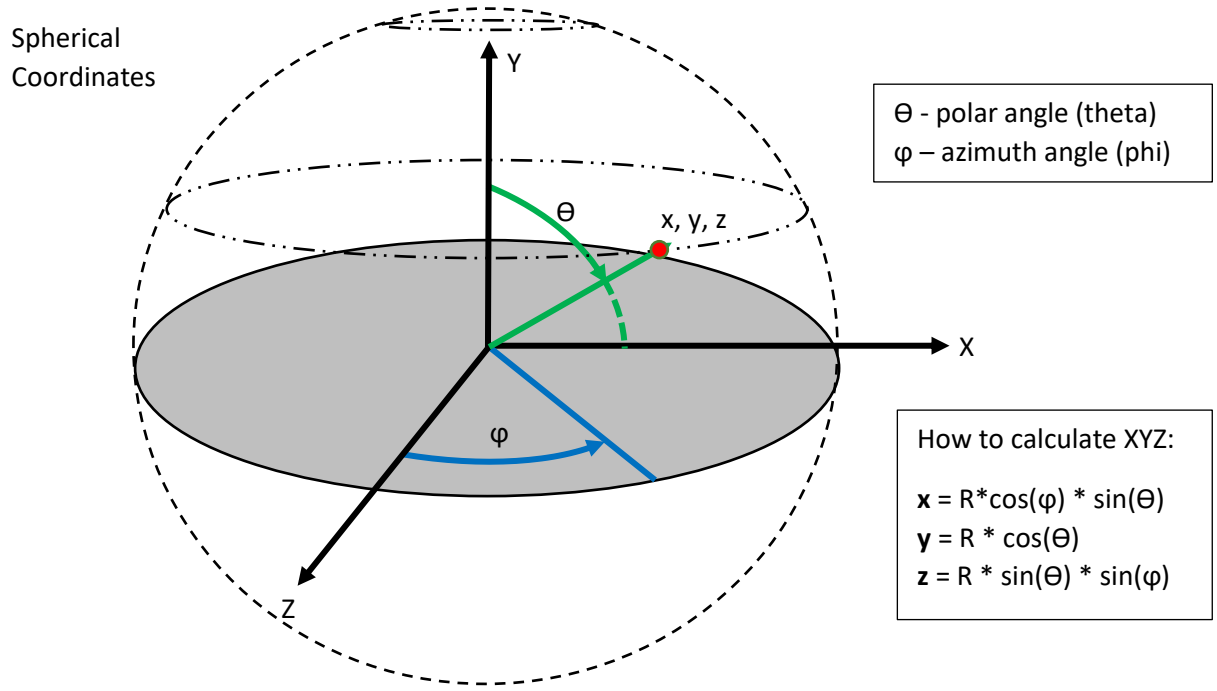
This function only needs to be called once, unless you want or need to change some of the values. The first value is typically the one that would change—a smaller angle is a narrower field of view, which can simulate zooming in on a target. 50 might be a "normal" field of view, while 10 would be zoomed in, and 90 would be much wider.

The second function, one that you will typically call every frame (unless you have a fixed camera that never needs to change position or look at something else):

```
camera(positionX, positionY, positionZ, // Where is the camera?
       target.x, target.y, target.z,    // Where is the camera looking?
       0, 1, 0); // Camera Up vector (0, 1, 0 often, but not always, works)
```

In this assignment you are going to create a class for the camera functionality. There are existing camera libraries available for use in Processing; **YOU MAY NOT** use them for this assignment. Your class should contain the following functions:

Update() - Called every frame from within the main draw() function, calculates values to pass to the camera() function
AddLookAtTarget(PVector) - Add a target to the list of positions to cycle through
CycleTarget() - Move to the next target in the list
Zoom(float) – Move toward or away from the look at target



Theta – has a range of 0 to 180 degrees (or 0 to **π** radians). If the angle is 0, that refers to straight up, along the Y axis. If the angle is 90 degrees, or **π**/2 radians, the vector would lie flat along the X/Z plane, and if the angle were 180 degrees, the final point would lie somewhere on the -Y axis.

Phi – has a range of 0-360 degrees (or 0 to 2π radians)

The **radius** in this application, will be the camera's offset from the target. For this application the range is 30 to 200, but your own program could use any value you like. (Though generally you wouldn't want to have a negative radius as some of the controls would then feel inverted.)

A basic implementation of getting the angles would be to use the map() function for the mouse X and Y positions

Initialize Variable	**Range**	**Map to…**
mouseX	0 -> width-1	Phi, 0 to 360
mouseY	0 -> height - 1	Theta, 1 to 179 (Why not 0-180? Short answer: Cameras can break in a variety of ways, one of which is looking directly along an "up" vector. Often this is avoided by limiting the camera to 1 degree shy of that direction)

The X, Y, and Z positions are relative to wherever the sphere is centered—in this assignment, that will the "look at" target. So the final position of the camera will be:

```
cameraPosition.x = lookatTarget.x + derivedX;
cameraPosition.y = lookatTarget.y + derivedY;
cameraPosition.z = lookatTarget.z + derivedZ;
```

Camera Usage

The intended use of the camera class would be to create an instance, add look at targets in the setup function, and then call Update() every frame in the draw function to calculate the proper location

Cycling Through Targets

In Processing, an easy way to handle input is with the keyPressed() function. This function gets called when ANY key is pressed, and then you can check a variable called keyCode to see what the specific key happened to be. For this assignment, any time the SPACE key is pressed, you can call the CycleTarget() function of your camera to switch to the next target.
https://processing.org/reference/keyPressed_.html

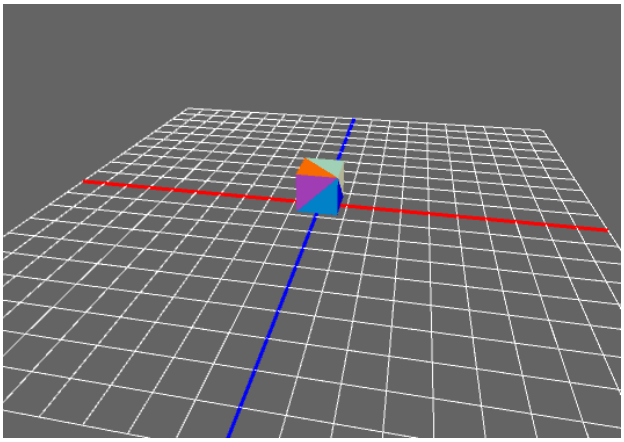
Moving Toward/Away From the Target

In addition to the keyPressed() function, there exists a function called mouseWheel() that will fire whenever the user scrolls a mouse wheel, or zooms in using touch gestures from a trackpad.
https://processing.org/reference/mouseWheel_.html

The value retrieved in that function can be sent to your camera's Zoom function. You may want to scale the value, as one "tick" of a mouse wheel might need to be many units in a 3D environment.

Grid

Navigating 3D space without some frame of reference can be an exercise in frustration. While a simulation or game will have a lot of rendered content to represent the details of some environment, many stages of development won't yet have that content. So what do you do? Create some! A simple grid to represent a ground of sorts, centered on the origin (0, 0, 0) will suffice.



To create one, you can use the line() function and a couple of loops to create something like the image on the right. That grid has minimum and maximum values of -100 and 100 along the X and Z axes, with lines every 10 units. Depending on what you were working on you might use other values for a larger or denser grid, and you might have some colors or other indicators of which axis is which. (It's very common in 3D applications to color-code the axes such that X is red, Y is green, and Z is blue—just think XYZ -> RGB.)

Shapes

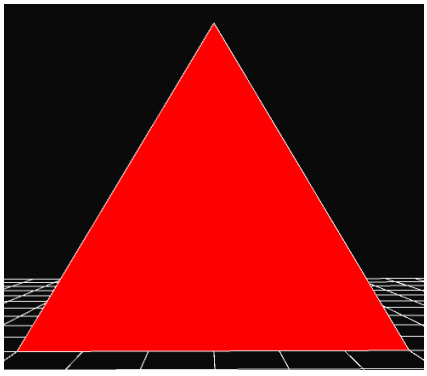
Processing handles collections of vertices in a class called PShape. The data for such an object can be created manually, or loaded from a file. In addition, data can be dynamically created and rendered on-demand, but this process is slower, and shouldn't be used where application performance is critical.

Creating shapes manually – Immediate mode

It's possible to create and render shapes immediately, as needed—this is something referred to as **immediate mode** rendering. This is typically a slower process, performance-wise, but it can be very helpful for the programmer as it allows them to get something up and running with minimal effort. In Processing this can be accomplished by using the beginShape() and endShape() functions, which you have already used in previous assignments. The only difference here is that in 3D, the vertex() function will need a 3rd component.

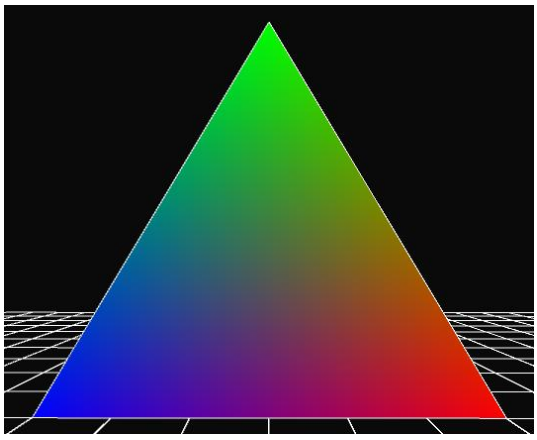
For example, if you wanted to create a single triangle, you might write:

```
beginShape();
vertex(-30, 0, 0);
vertex(0, -50, 0);
vertex(30, 0, 0);
endShape();
```



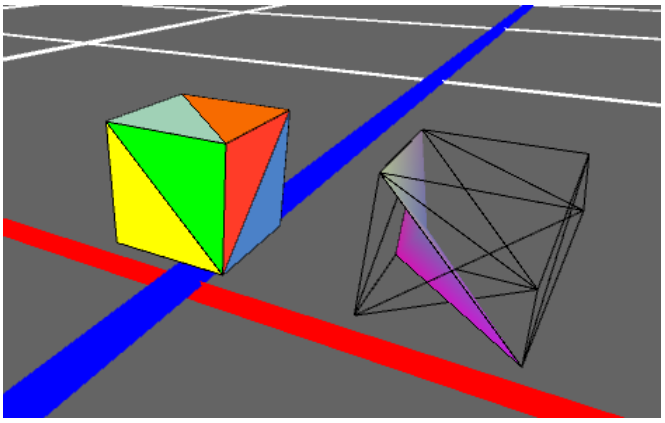
You can also set per-vertex colors when creating a custom shape. This is done by calling the fill() function before each call to vertex.

```
beginShape();
fill(255, 0, 0);
vertex(-30, beginShape();
fill(255, 0, 0);
vertex(-30, 0, 0);
fill(0, 255, 0);
vertex(0, -50, 0);
fill(0, 0, 255);
vertex(30, 0, 0);
endShape();0, 0);
fill(0, 255, 0);
```



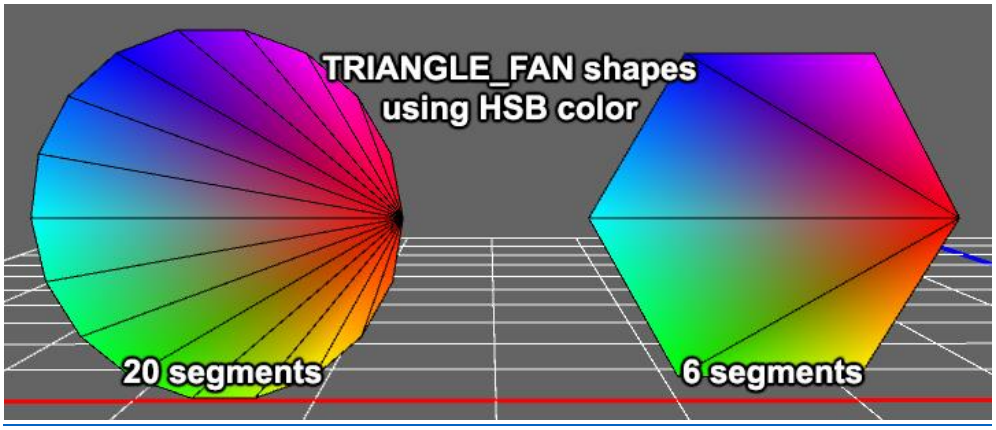
This process can be used to create arbitrarily complex shapes, including shapes which are made of multiple polygons.

When creating complex shapes, it's important to use the beginShape() function properly. Up until this point you haven't had to pass it any parameters, but you can pass a variety of values to it, which will determine how the data you store in the shape gets processed at render time.



Left: beginShape(TRIANGLE) Right: beginShape()

For example, the image on the right is the same data for a cube, but in one instance beginShape() is called with the parameter TRIANGLE passed to it, which will treat all vertex() data sent as though it should be grouped in batches of 3, to create a triangles. There are many ways to use beginShape()—the documentation has some additional examples: https://processing.org/reference/beginShape_.html



Creating shapes manually – Retained mode

Creating shapes manually is fine, but if you need to render a lot of objects, immediate mode will be a bit of a drag on the application's performance. You can create an instance of the PShape object, initialize it using the same beginShape(), endShape(), vertex() functions you've used before—this time, however, they are members of the PShape object, and must be called as such.

```
PShape triangle = createShape();
triangle.beginShape();
triangle.vertex(-30, 0, 0);
triangle.vertex(0, -50, 0);
triangle.vertex(30, 0, 0);
triangle.endShape();
```

```
// Elsewhere…
void draw()
{
   // Draw the shape
   shape(triangle);
}
```

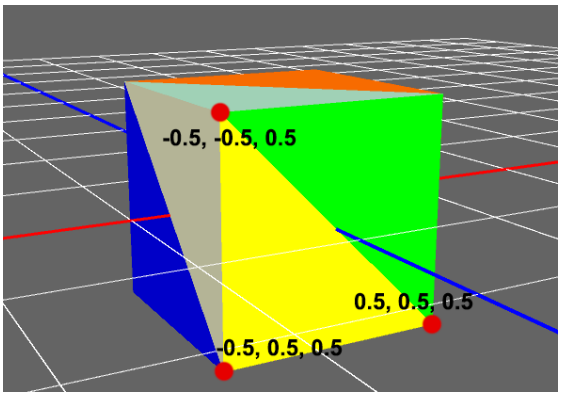
In addition to setting the vertex data, a PShape can store its own render settings like stroke, and fill color, which you set by calling various functions. For example, if you wanted the shape to use a stroke, you could set it by writing the following:

```
someObject.setFill(color(255, 0, 255));
someObject.setStroke(true);              // Use a stroke at all
someObject.setStroke(color(0));          // Set the stroke color
someObject.setStrokeWeight(2.0f);
```

Once the shape has been created, you can draw it by simply calling the shape() function, and passing in the PShape object.

Creating a Cube

A cube is a simple shape: 6 sides, 2 triangles per side, 3 vertices per triangle—36 vertices in total. A single cube with length/width/height of 1 is sometimes called a Unit Cube, and is often centered on the origin, like this:



Each vertex is half a unit offset from the origin along each axis, depending on which triangle you are setting up. You can set the fill() function once per triangle; no need to call it once per vertex (unless you want them to be different).

Loading shapes from a file

Creating a complex mesh by hand would be painful, to say the least. For that, we load files containing the data that created from some external source (typically a 3D modeling program). To do that, Processing has a super-simple function, loadShape(). Simply give it the name of the file you want to load (which has to be of type .SVG or .OBJ), and the relevant data is read into a new PShape. So if you wanted to load a model of, say a tree, and it was called “tree.obj” you would just write:

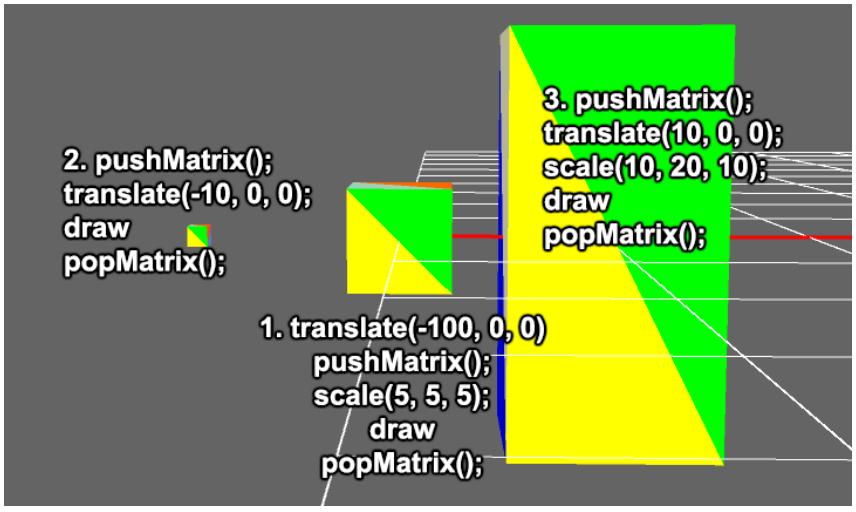
PShape someObject = loadShape(“monster.obj”);

The file to load must be in a folder called **data** inside your sketch folder (you can create this manually). Once you have this object, the same rules for setting properties or drawing the shape still apply.

Transformation Frames

Once you have everything created, drawing it in the proper location is the next step. The translate(), rotate() and scale() functions will allow you to control where something is drawn, how it is oriented, and the size of the object. It can often be convenient to do so from some particular frame of reference. The default frame is the origin. Draw something, and it appears relative to the origin. Call translate(100, 0, 0) and now something is rendered 100 units from the origin along the X axis.

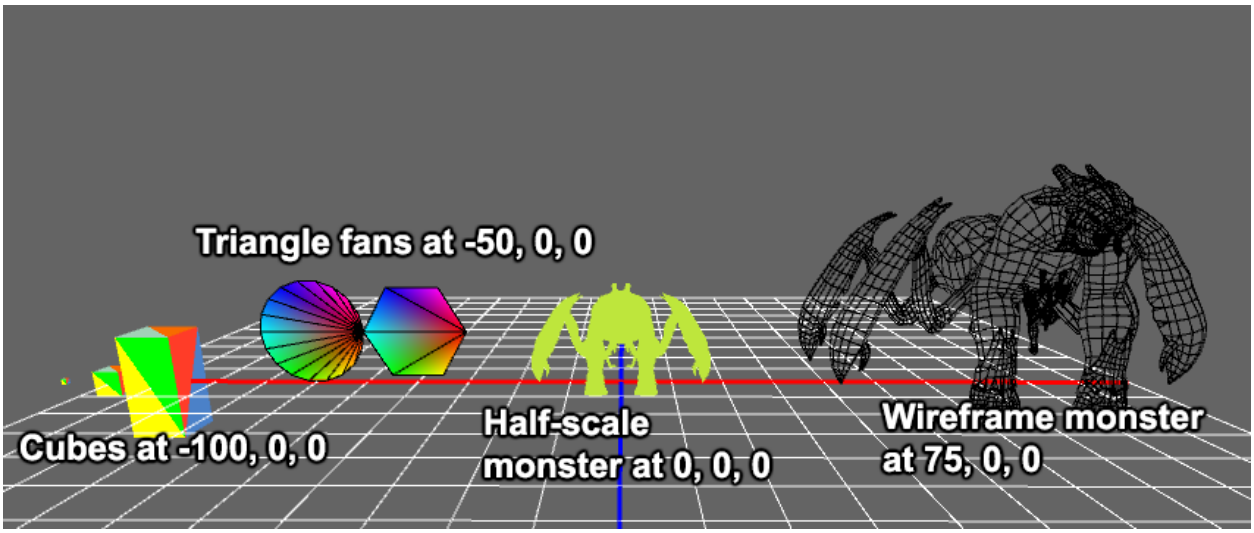
To render something 100 units away from THAT location, however, you might want to change the origin temporarily. In OpenGL (and in Processing) that would be with the pushMatrix() function. This takes whatever previous calls to translate/rotate/scale and uses that as a new origin. Any future calls will be relative to that. When you want to “go back” to the previous transformation, you would use the popMatrix() function. For example, to draw the boxes the left side of the first image, you might do something like this:



(You don’t have to use push/popMatrix, but it is often quite helpful)

All that translating, rotating, or scaling just sets a matrix to be used to transform whatever you happen to draw after. So you can draw the same shape 100 times using different translations or rotations to populate an entire scene. Virtually every application out there uses a similar process—set a transformation, draw something, set a transformation, draw something, for each object.

Where does everything go cheat sheet



Submissions

Create a .zip file with any code files you created for this project (in Processing they are files with the extension .pde), and name the file ***LastName.FirstName.Project3.zip***. Submit the .zip file on the Canvas page for Project 3.

Tips

- Get the grid and the camera implemented first. Navigating 3D space can be unintuitive at first, having a frame of reference makes all the difference.

Grading

Item	Description	Maximum Points
Grid	#frameofreferenceftw	10
Camera	Orbit functionality, required functions implemented	40
Load and draw shapes from .OBJ file	Wireframe monster and half-scale monster	20
Cubes as a PShape	Cube created and rendered 3 times with 3 transformations	20
Triangle fans	Circle and hex with HSB color and stroke	10
	Total	100