Three D Space . . . muhaaaaaaaa

Thus far the first line of code in our programs has been a function call of the function **size( )** similar to this :

  size(400, 400);

This tells Processing how big the graphics window must be for our program. There is another form (or **signature**) for calling the function **size( )**:

  size(400, 400. P3D);

This second form or signature tells Processing that we want to work in 3-d Space (**Processing 3 Dimesions**). This will cause Processing to use a different set of arithmetic , geometric, and trigonometric functions to figure out how to render (draw) the shapes we tell Processing to draw. So there is a slightly different first line required in our code.

There are two functions in the API specifically for use in 3-d:
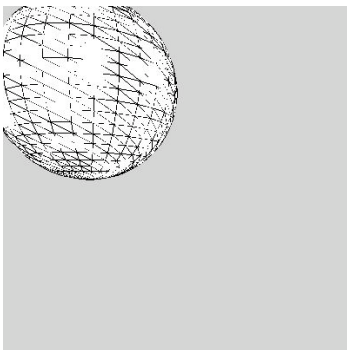
  box( ) and **sphere( )**.

The **box( )** function has two signatures:

  box(length of the edges);
  box(width, height, depth);

The **sphere( )** function has only one signature:

  sphere(radius);

Note : You can see at the left that the sphere is drawn with lots of tiny triangles. It takes a lot of computation to determine where to draw and then draw all of them. Even if you turn off the stroke with **noStroke( )** - the triangles are still drawn but with out the stroke outlining them.. If you compose your initials with a lot of spheres, it will take a long time for Procecessing to compute the location of the triangles and draw them. A future homework will use this code as the basis for "flying" around your initial in 3-D space. An initial with a lot of spheres could execute so slowly that the animation becomes jumpy and not very pleasing. Use them if you wish - it is up to you. . . da' moose
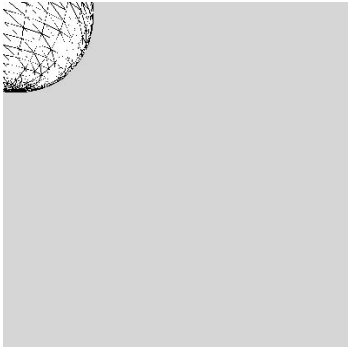
3. The box and sphere are drawn centered at the **(0, 0, 0)** location in the window. It does not appear that you can alter this .

4. *The function translate*(**x, y, z**) *is used to move or shift the*
(**0, 0, 0**) *point around.*

If we execute this code:

```
size( 400, 400, P3D );
sphere( 100 );
```
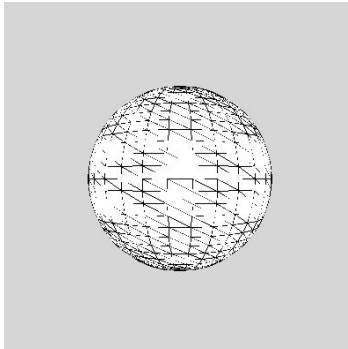*we would see this on the screen:*

Since the origin is at the upper left corner, we see ¼ of the sphere.
If we execute this code:

```
size( 400, 400, P3D );
translate( 200, 200, 0 );
sphere( 100 );
```
we see this:

The center of this sphere is 200 pixels to there right and 200 pixels
down and zero pixels in depth and we see the entire sphere.

If we execute this code:

```
size( 400, 400, P3D );
strokeWeight( 5 );
translate( width/2, height/2, 0 );
stroke( 0, 255, 0 );        // green
```

```
line( 0, -200, 0, 200 );  // horizontal line
stroke( 0, 0, 255 );      // blue
line( -200, 0, 200, 0 );  // vertical line
noStroke( );
fill( 0 );
ellipse( 0, 0, 10, 10 );   // circle at the origin
fill( 255, 0, 0 );
textSize( 18 );
textAlign( CENTER );
text( "(0, 0 )", 0, -10 );
text( "- x space\n- y space", -100, -100 );
text( "+ x space\n- y space",  100, -100 );
text( "+ x space\n+ y space",  100,  100 );
text( "- x space\n+ y space", -100,  100 );
saveFrame( "bn0114.2.jpg" );
```

we see this output: (Check the API for the functions this code uses
that you are not familiar with. Bring your questions to class.):
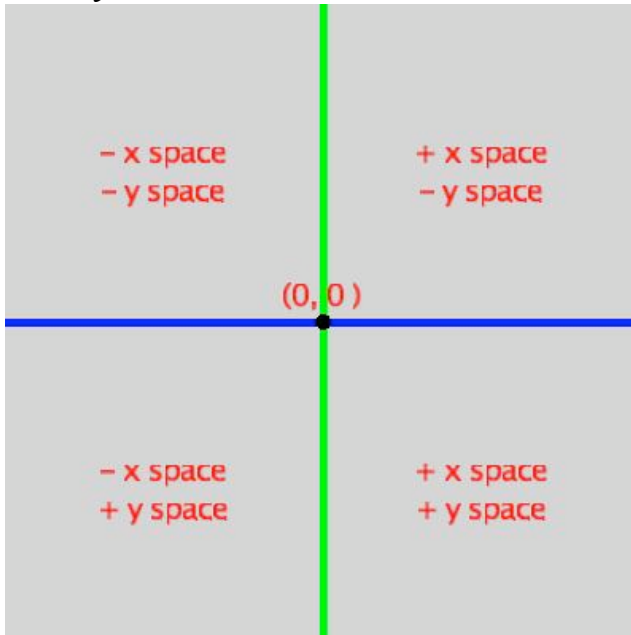
The z coordinate space works as follows:
positive Z translation moves the(0,0,0) point closer to you
negative Z translation moves the(0,0,0) point away from you

5. One translation is ok but if we want to locate a bunch of spheres or boxes on the screen, keeping track of where we are can be difficult.

One solution to this problem is to always translate from the upper left corner of the window. We could do this

```
size( 400, 400, P3D );
translate( 100, 100, 0 );
sphere( 10 );
translate( -100, -100, 0 );
translate( 200, 50, 0 );
box( 12 );
translate( -200, -50, 0 );
```
etc…

but for a lot of translations, it can get very confusing and if we are using variables and expressions, it can get worse.

IMPORTANT NOTE: Once your code draws something, subsequent translations DO NOT alter the location of previously drawn objects.

There is another way. We can do "temporary" translations from the origin and then jump back to the origin when we are done. Here is how this works. In order to do the arithmetic (geometry) needed to draw the shapes, Processing uses a **matrix** to keep track of the translations and rotations. When we translate and rotate, Processing alters this matrix for us (thank goodness…)

Processing will give us a "temporary copy of the **matrix**" if we ask for it. Processing will use the **temporary matrix** for any translations and drawings and not change the original **matrix** . Using this **temporary matrix**, we can shift to a new point, draw stuff, and then throw it away. When we throw it away, the origin is back where it was before we asked for the **temporary matrix** and the **matrix** is unchanged.

To get the **temporary matrix**, we use the function:
```
pushMatrix( );
```
To throw it away we use the function
```
popMatrix( );
```

*The above code would look like t*his:

```
size( 400, 400, P3D );
    // ( 0, 0, 0) is the upper left corner of the window
    pushMatrix( );
        translate( 100, 100, 0 );
        sphere( 10 );
    popMatrix( );
    // ( 0, 0, 0) is the back at the upper left corner of the window
    pushMatrix( );
        translate( 200, 50, 0 );
        box( 12, 5, 7 );
     popmatrix( );
    // ( 0, 0, 0) is the back at the upper left corner of the window
```

The indenting Jim used is just to show visually what is done inside the temporary matrix. It is not required but it makes the code much easier for others to read.

6. For those of you who really want to spend some time – you can explore three **rotate** *functions:* **rotateX( ), rotateY( ), and rotateZ( )***.*
The argument must be the amount of rotation in *radians**. We strongly recommend you use the***radians( )** *fu*nction for the argument.*
  rotateX(radians(15));
This will rotate the x-axis 15 degrees in the positive direction.
You have to rotate each axis separately.
Translation should be done first, then rotation, and finally draw:

```
    size( 400, 400, P3D );
    noStroke( );
    background( 0 );
    lights( );
    // ( 0, 0, 0) is the upper left corner – rotations are 0 degrees
    pushMatrix( );
        translate( 100, 100, 0 );
        rotateX( radians( 25 ) );
        box( 50 );
    popMatrix( );
    // ( 0, 0, 0) is back at the upper left corner – rotations are 0 degrees
    pushMatrix( );
```
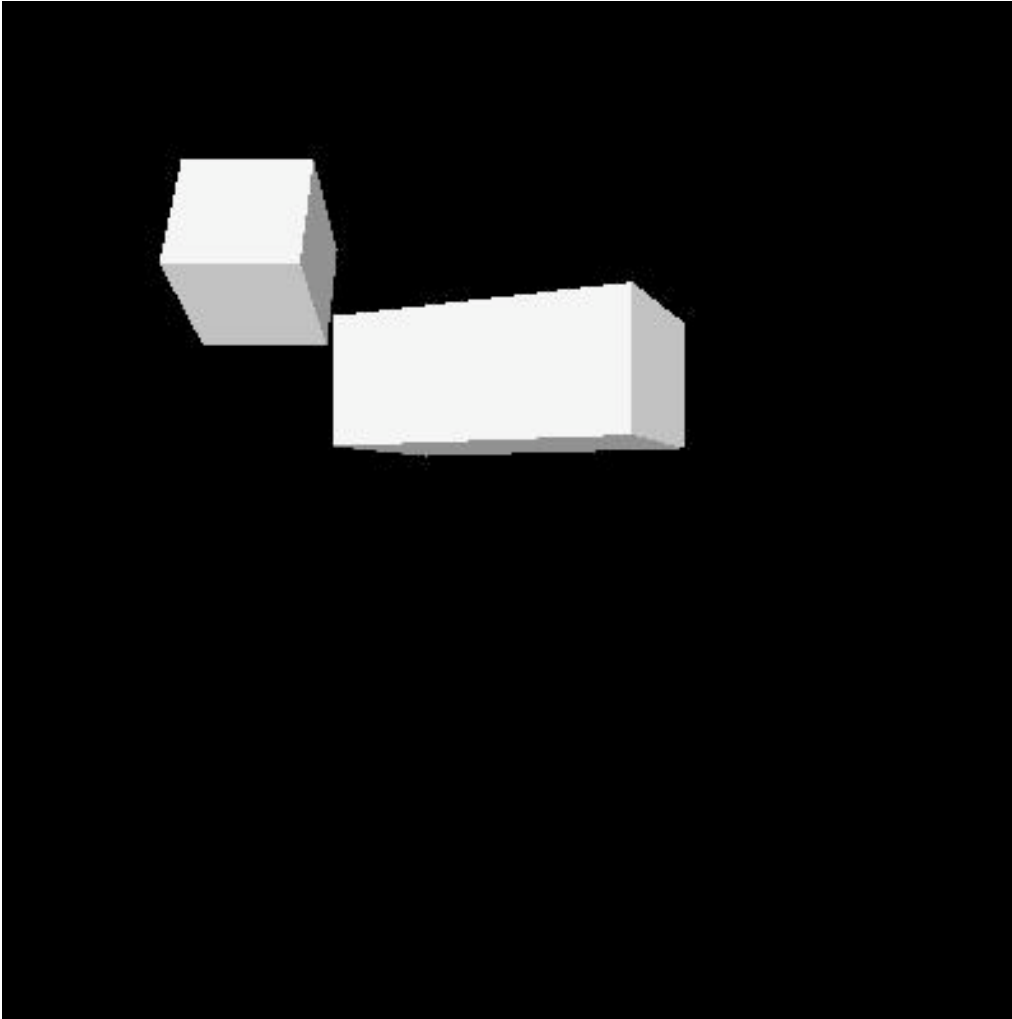
```
     translate( 200, 150, 0 );
     rotateY( radians( -25 ) );
      box( 120, 50, 70 );
   popMatrix( );
  // ( 0, 0, 0) is back at the upper left corner – rotations are 0 degrees
```

The code on the previous page produces this output in the graphics window:

Copy the code into a Processing program and alter the values of the rotations. Comment out the call of the **lights()** *function. This is one way to "get a feel" for working with these functions*