Week 9a: Translation, Scaling, Rotation

Translation, Scaling, Rotation

Week9a The World Revolves Around You

- Week 7a,b: Mathematics/Algorithms
- Week 7a: Translation and Rotation (in 3D!)
  - The Z axis
  - P3D versus OpenGL
  - Vertex Shapes
  - Custom 3D Shapes
  - Simple Rotation
  - Rotation around different Axes
  - Scale
  - Saving the transformation state in the stack: pushMatrix() and popMatrix()
  - A Processing Solar System

The Z Axis

- In three-dimensional space (such as the actual, real-world space), a third axis (commonly referred to as the Z-axis) refers to the depth of any given point.
- In a Processing sketch's window, a coordinate along this Z -axis indicates how far in front or behind the window a pixel lives.
- Wait… my screen is only 2D…
  - The Z-axis will create the illusion of three-dimensional space in your Processing window.

Class Activity 1: A growing rectangle

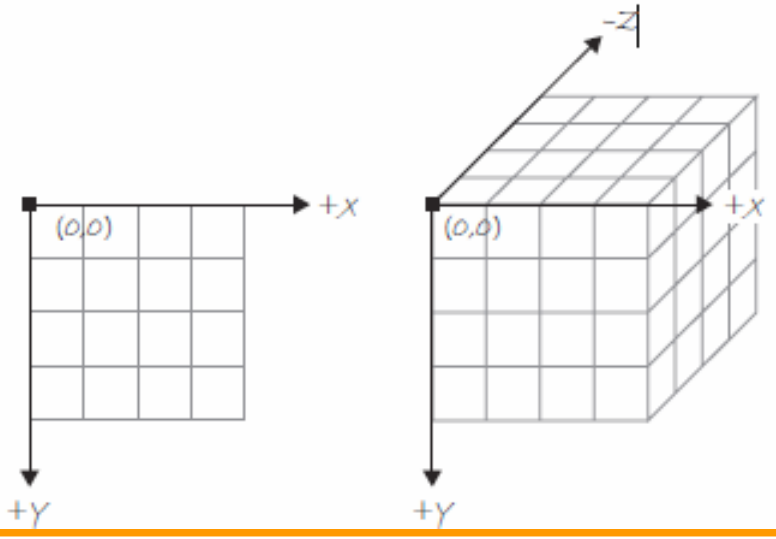
- You can create the illusion of 3D with tools you already know!

```
float r = 8;
void setup() {
  size(200,200);
}
void draw() {
  background(255);
  // Display a rectangle in the middle of the screen
  stroke(0);
  fill(175);
  rectMode(CENTER);
  rect(width/2,height/2,r,r);
  // Increase the rectangle size
  r++ ;
}
```

So the rectangle grows.. Toward you?

Now for the 3D Part…

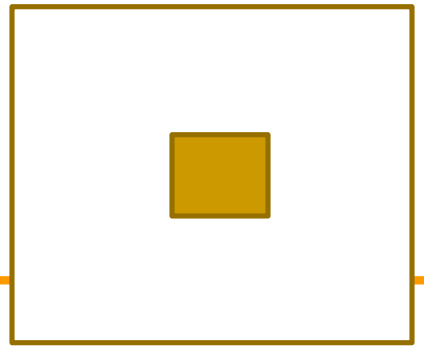
- If we choose to use 3D coordinates, Processing will create the illusion for us.
- In order to specify points in three dimensions, the coordinates are specified in the order you would expect: x , y , z .
- Cartesian 3D systems can be described as “left-handed” or “right-handed.”. Processing is “left-handed”.



The `translate()` function

- There is no rect(x,y,z…) in Processing
- In order to use the z axis, we need to learn `translate()`
- The function translate( ) moves the **origin point** (0,0 relative to its previous state.
  - The "origin" in a Processing sketch is the point (0,0) in two dimensions or (0,0,0) in three dimensions.
  - It is always at the top left corner of the window unless you move it using translate( ).
  - The origin always resets itself back to the top left corner at the beginning of draw( ).
  - Any calls to translate( ) only apply to the current cycle through the draw( ) loop.
- So if you translate to 100,100 and

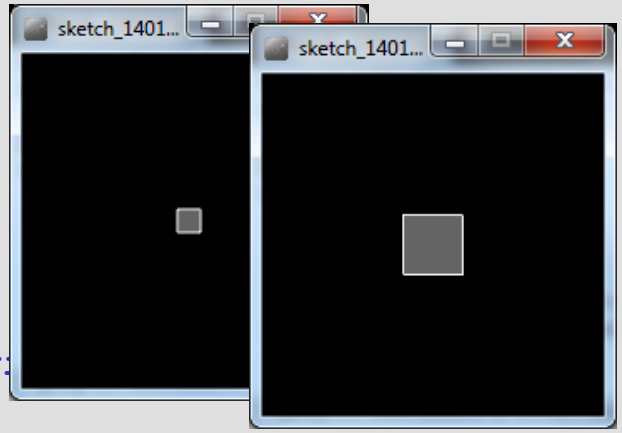
draw a rectangle at 0,0, it is drawn here:



Re-write Class Activity 1:

- Use the Z-axis
- The size() method in setup has ‘P3D’ as a third param
- translate() has a z axis parameter!
- So as the z value increases, the rectangle gets ‘closer’

```
float z = 0; // a variable for the Z (depth) coordinate
void setup() {
  size(200,200,P3D);
}
void draw() {
  background(0);
  stroke(255);
  fill(100);
  // Translate to a point before display
  translate(width/2,height/2,z);
  rectMode(CENTER);
  rect(0,0,8,8);
  z++; // Increment Z (i.e. move the shape toward the viewer)
}
```



P3D mode versus OpenGL

```
void setup() {
  size(200,200,P3D);
}
```

- size() can take a third parameter for 'rendering mode'
- The default mode (when none is specified) is "JAVA2D"
  - It uses Java 2D libraries to draw shapes, set colors, and so on.
  - JAVA2D mode doesn't know how to do 3D (surprised?)
  - Trying to use translate(x,y,z) in this mode will fail
- There are two 3D rendering modes in Processing:
  - P3D: Developed by the creators of Processing . It should also be noted that anti-aliasing (enabled with the smooth() function) is not available with P3D.

```
size(200,200,P3D);
```

  - OpenGL: Employs hardware acceleration. If you have an OpenGL compatible graphics card installed on your computer (which is pretty much every computer), you can use this mode. It may improve rendering speed.
    - You must also import the openGL library

```
import processing.opengl.*;
size(200,200,OPENGL);
```

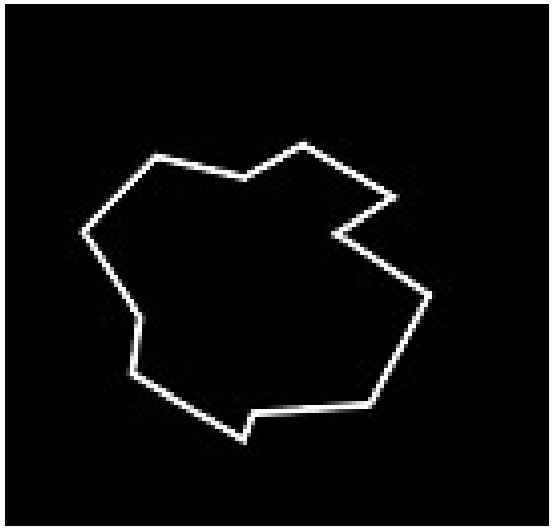
Vertex shapes

- You may have been wondering how to create more interesting shapes (like asteroids?)
  - We have been limited to lines, triangles, ellipses…
- Three new functions
  - beginShape()
  - vertex()
  - endShape(CLOSE)
    - Declares that the shape should be closed, that is, that the last vertex point should connect to the first.
- Current fill(), stroke(), etc. still apply

```
beginShape();
vertex(50,50);
vertex(150,50);
vertex(150,150);
vertex(50,150);
endShape(CLOSE);
```

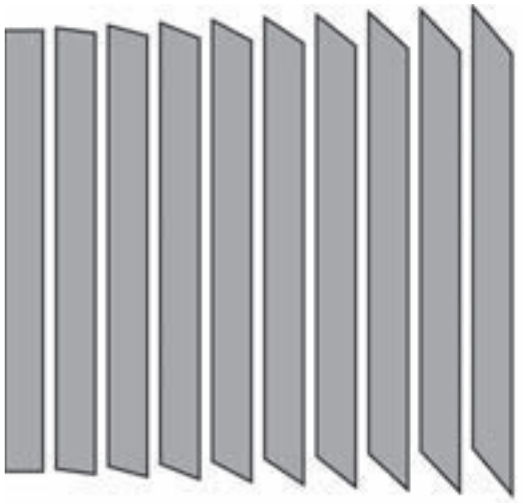
Make me an asteroid

- Get out your graphpaper
- How many vertexes? _____
- You don't need to go back to the first
- How could you make it smaller or larger?



```
beginShape();
// one for each 'point'
endShape(CLOSE);
```

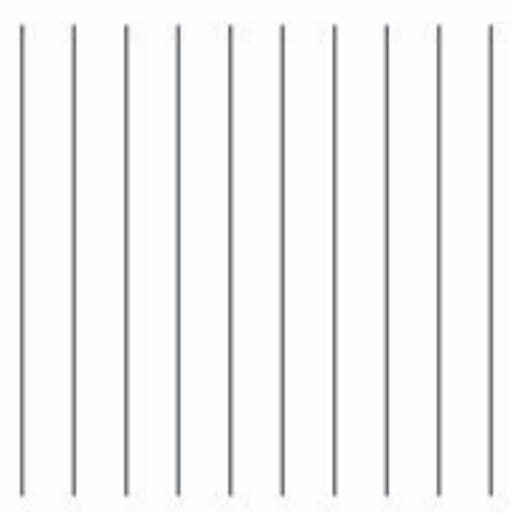
Vertex Shapes in loops



- How could you create a set of shapes line these?
  - With a loop of course, and vary some of the vertexes
  - You CAN draw shapes with unequal sides!

```
stroke(0);
for (int i = 0; i < 10; i++) {
  beginShape();
  fill(175);
  vertex(i*20,10-i);
  vertex(i*20 + 15,10 + i);
  vertex(i*20 + 15,180 + i);
  vertex(i*20,180-i);
  endShape(CLOSE);
}
```

beginShape Agruments

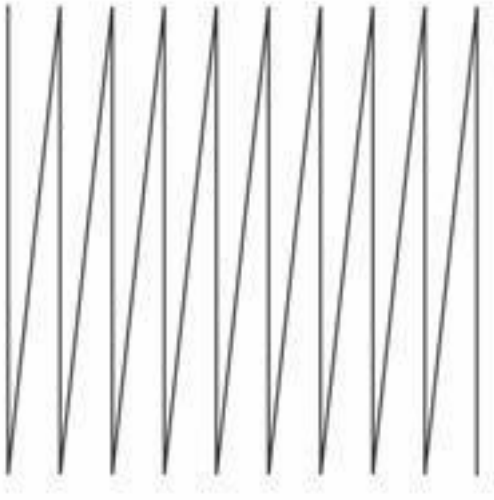


- You can use a set of shape keywords to tell processing how many vertexes are per shape
- Put beginShape outside the loop
  - POINTS: 1
  - LINES: 2
  - TRIANGLES: 3
  - QUADS: 4
  - Advanced Shapes:
    - TRIANGLE_FAN
    - TRIANGLE_STRIP
    - QUAD_STRIP
    - See processing reference for details

```
stroke(0);
beginShape(LINES);  // argument!
for (int i = 10; i < width; i+= 20) {
  vertex(i,10);
  vertex(i,height-10);
}
endShape();  // No CLOSE
```

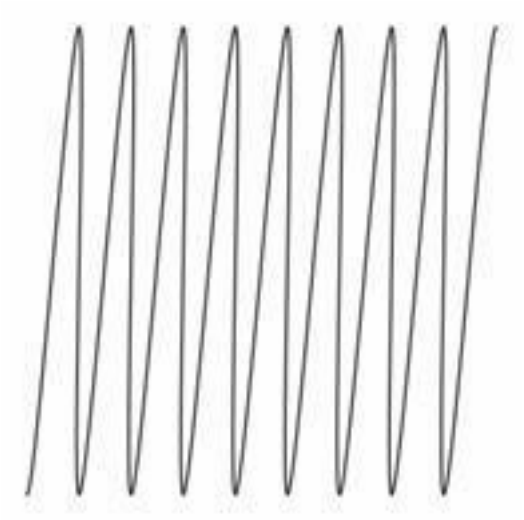
Continuous Shape

- For a continuous loop:
  - use noFill()
  - beginShape before loop (no argument)
  - Specify all the vertex points in loop
    - Two vertices per loop (vertical line)
    - Diagonal connects last to first of next
    - Total of 20 vertices, 19 lines
  - endShape after loop (no CLOSE)

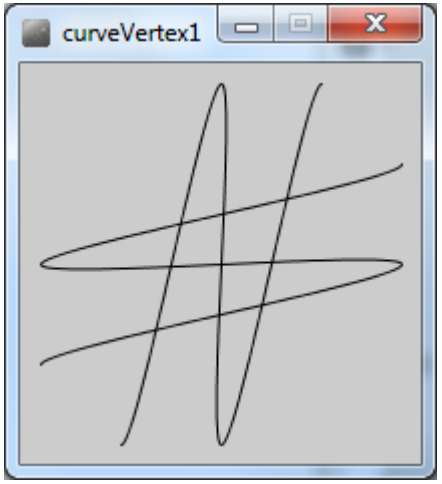


```
noFill();
stroke(0);
beginShape();  // No argument
for (int i = 10; i < width; i+= 20) {
  vertex(i,10);
  vertex(i,height-10);
}
endShape();    // No CLOSE
```

curveVertex



- To join the points with curves instead of straight lines.
- Note how the first and last points are not displayed.
  - They are required to define the curvature of the line as it begins at the second point and ends at the second

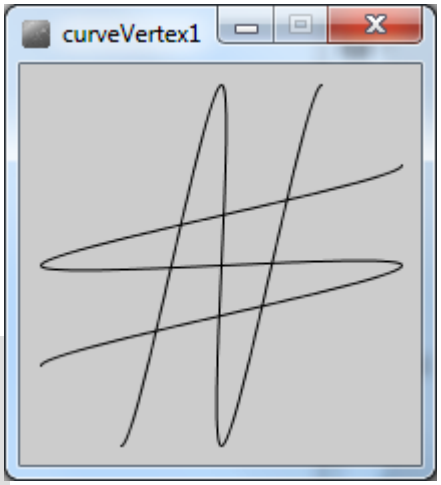



```
noFill();
stroke(0);
beginShape();
for (int i = 10; i < width; i += 20) {
  curveVertex(i,10);
  curveVertex(i,height-10);
}
endShape();
```

curveVertex

- See if you can draw some script letters (your initials?)

```
void draw() {   // assumes size(200,200);
  noFill();
  stroke(0);
  beginShape(); // Draw the N
  for (int i = 50; i < width; i += 50){
    curveVertex(i, 10);
    curveVertex(i, height-10);
  }
  endShape();
  beginShape(); // Now draw an S
  for (int i = 50; i < width; i += 50) {
    curveVertex(10, i);
    curveVertex(width-10, i);
  }
  endShape();
}
```

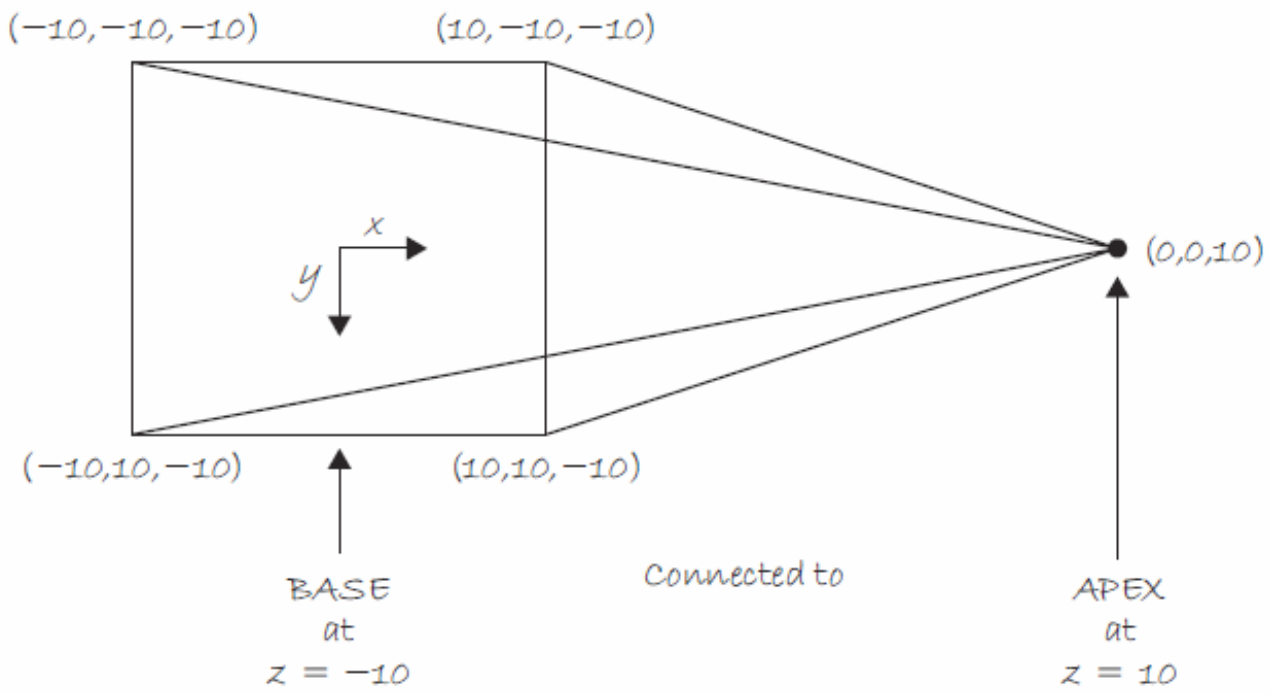


x	y	show?
50	10	N
50	190	Y
100	10	Y
100	190	Y
150	10	Y
150	190	N

x	y	show?
10	50	N
190	50	Y
10	100	Y
190	100	Y
10	150	Y
190	150	N

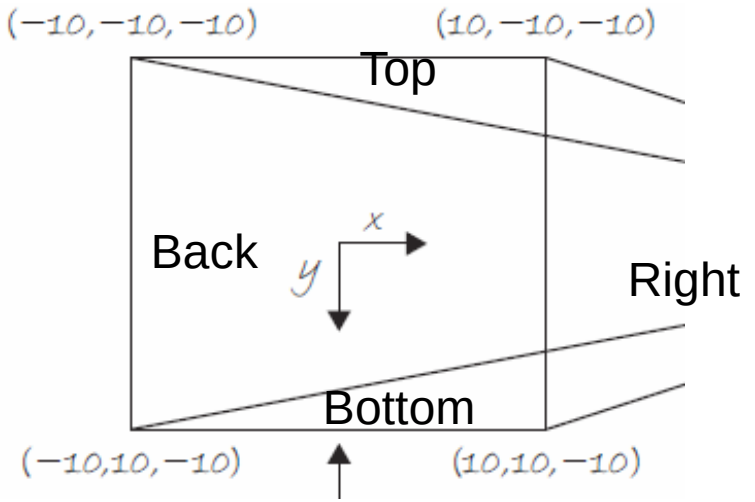
Custom 3D Vector Shapes

- Three-dimensional shapes can be created using beginShape( ), endShape( ), and vertex( ) by placing multiple polygons side by side in the proper configuration.
- Example: A four-sided pyramid made up of four triangles, all connected to one point (the apex) and a flat plane (the base).



Pyramid Vectors

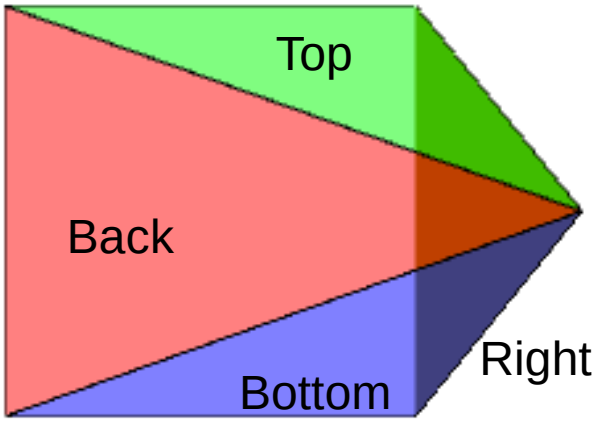
- Four Triangles
- “Back” x y z
  - vertex(–10,–10,–10);
  - vertex(–10, 10,–10);
  - vertex(0, 0, 10); // apex
- “Top”
  - vertex(–10,–10,–10);
  - vertex(10,–10,–10);
  - vertex(0, 0, 10); // apex
- “Bottom”
  - vertex(–10, 10,–10);
  - vertex(10, 10,–10);
  - vertex(0, 0, 10); // apex
- “Right”
  - vertex(10,–10,–10);
  - vertex(10, 10,–10);
  - vertex(0, 0, 10); // apex



- If your shape is simple enough, you might be able to get by with just writing out the code.
- In most cases, however, it is best to start sketching it out with pencil and paper

Pyramid Function (with Color)

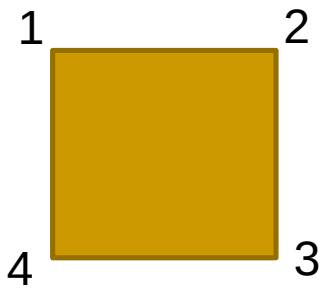
- translate() sets the 'center' point
- beginShape(TRIANGLES)
  - 3 vertices
- Draws opaque right first to see it




```
void drawPyramid(int t) {
  translate(width/2,height/2,0);
  stroke(0);
  beginShape(TRIANGLES);
  fill(127,127,127);  // Right: Opaque gray
  vertex( t,-t,-t); vertex( t, t,-t); vertex( t, 0, t);
  fill(255,0,0,127);  // Back: Transparent Red
  vertex(-t,-t,-t); vertex(-t, t,-t); vertex( t,0,t);
  fill(0,255,0, 127); // Top: Transparent Green
  vertex(-t,-t,-t); vertex( t,-t,-t); vertex( t, 0, t);
  fill(0,0, 255,127); // Bottom: Transparent Blue
  vertex(-t, t,-t); vertex( t, t,-t); vertex( t, 0, t);
  endShape();
}
```

Drawing Quads

- beginShape(QUAD) requires four vertices
  - Start anywhere (I started UL)
  - Go around one way
  - Back is green, front is White

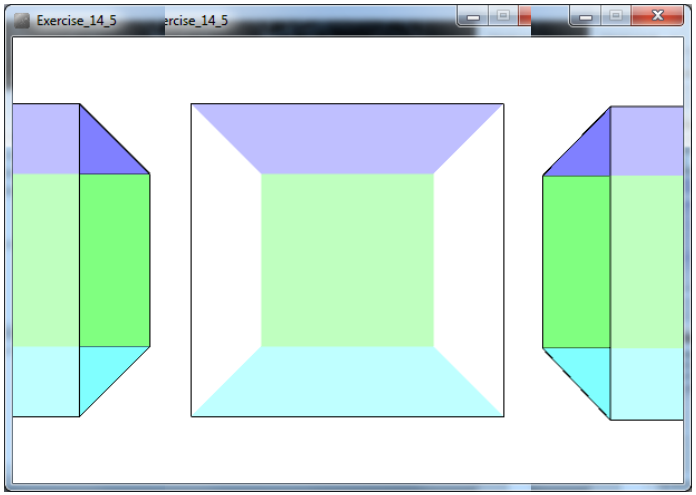


Front	x	y	z
UL (1)	-t	-t	t
UR(2)	+t	-t	t
LR(3)	+t	+t	t
LL(4)	-t	+t	t

Top	x	y	z	Blue
	-t	-t	t	UL Front
	+t	-t	t	UR Front
	+t	-t	-t	UR back
	-t	-t	-t	LR back

Back	x	y	z
UL (1)	-t	-t	-t
UR(2)	+t	-t	-t
LR(3)	+t	+t	-t
LL(4)	-t	+t	-t

Bot	x	y	z	Cyan
	-t	+t	t	LL Front
	+t	+t	t	LR Front
	+t	+t	-t	LR back
	-t	+t	-t	LL back



- Top and bottom appear as trapezoids
- Draw back shapes first

Left: Translate x to 0, Right x to width

Quads

- Make the other four sides
  - Top, Bottom, Left and Right
  - Suggested order:
    - Top
    - Right
    - Bottom
    - Left
    - Back
    - Front
    - Make them different colors

When you're done, lookup Processing's 3D 'Box' shape

```
void drawQuad(int t) {
  // translate before calling (mouse?)
  stroke(0);
  beginShape(QUADS);
  fill(0,255,0,127);    // Green Back
  vertex( -t,-t, -t);   // UL corner
  vertex(  t,-t, -t);   // UR
  vertex(  t, t, -t);   // LR
  vertex( -t, t, -t);   // LL
  fill(255,255,255,127);// White Front
  vertex( -t,-t, t);    // UL corner
  vertex(  t,-t, t);    // UR
  vertex(  t, t, t);    // LL
  vertex( -t, t, t);    // LR
  endShape();
}
```

Simple Rotation (in 2D)

- One way to really see a 3D object would be to rotate the it. So let's learn about rotation.
- We already know how to change the 'center point' using translate().
- Programming rotation, unfortunately, is not so simple. All sorts of questions come up.
  - Around what axis should you rotate?
  - At what angle?
  - Around what origin point?
- Processing offers several functions related to rotation, which we will explore slowly, step by step.
- Our goal will be to program a solar system simulation with multiple planets rotating around a star at different rates.

Three Principles of Rotation

- We should be able to get rotation going with the following three principles:

1. Shapes are rotated in Processing with the rotate() function.

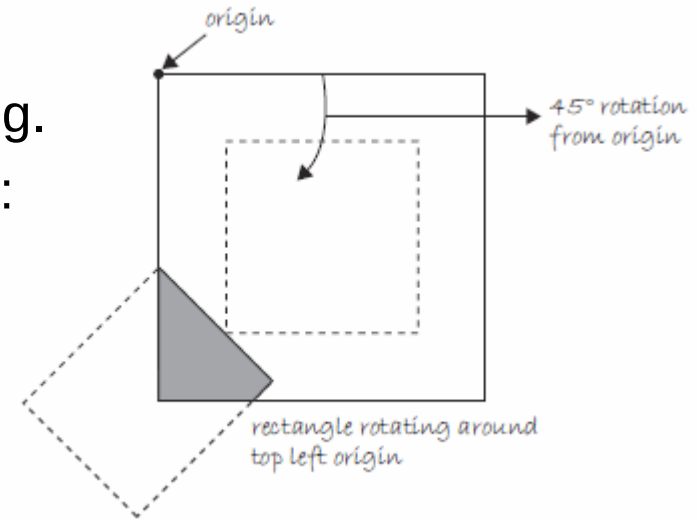
2. The rotate( ) function takes one argument, an angle measured in radians.

3. rotate( ) will rotate the shape in the clockwise direction (to the right).

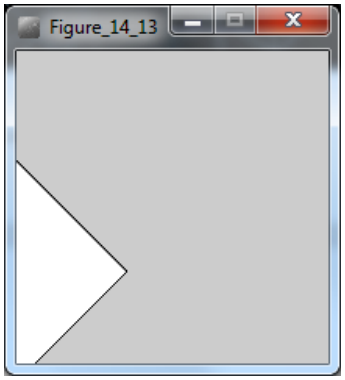
Rotating a rectangle in 3D

- We should be able to just call the rotate( ) function and pass in an angle.
  - Say, 45° (or PI/4 radians) in Processing.
  - Here is our first (albeit flawed) attempt:

```
rotate(radians(45));
rectMode(CENTER);
rect(width/2,height/2,100,100);
```



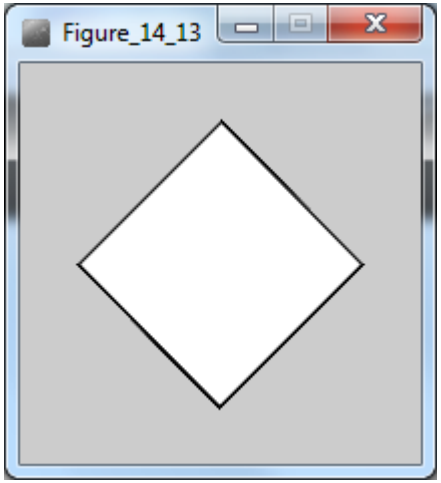
- What's wrong?
  - The rectangle looks rotated, but it is in the wrong place!
  - Where is the point of origin in this example? The top left corner!
  - The origin has not been translated.



The single most important fact to remember about rotation in Processing is that shapes always rotate around the point of origin.

Rotating after translating

```
translate(width/2,height/2);
rotate(radians(45));
rectMode(CENTER);
rect(0,0,100,100);
```



- That worked… Here's why
  - Translate put the origin in the center of the screen
  - Rotate is set to 45 degrees
  - You drew a rectangle at the origin (0,0), not (width/2, height/2)

Class Activity: Mouse controlled rotation

- Use the mouseX location to calculate an angle of rotation and thus animate the rectangle, allowing it to spin

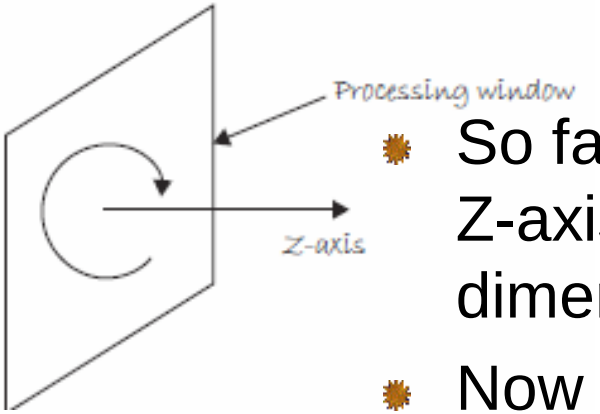
```
void setup() {
  size(200,200);
}
void draw() {
  background(255);
  stroke(0);
  fill(175);
  translate(width/2,height/2); // Translate origin to center

  // theta is a common name of a variable to store an angle
  float theta = PI*mouseX / width;

  rotate(theta);    // Rotate by the angle theta
  rectMode(CENTER); // Display rectangle with CENTER mode
  rect(0,0,100,100);
}
```

The angle ranges from 0 to PI, based on the ratio of mouseX location to the sketch's width.

Exercise 1: Twirl a Baton

- Create a line that spins around its center (like twirling a baton).
- Draw a circle at both endpoints.
- Can you make it twirl in either direction?

Rotation Around Different Axes

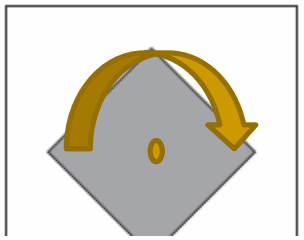


- So far we have rotated shapes around the Z-axis. This is the default axis for two-dimensional rotation.
- Now that we have basic rotation out of the way, we can begin to ask the next important rotation question:
  - Around what axis do we want to rotate?

- Processing can rotate shapes around the x or y-axis with the functions rotateX( ) and rotateY( ),which each require P3D or OPENGL mode. The function rotateZ( ) also exists and is the equivalent of rotate( ).

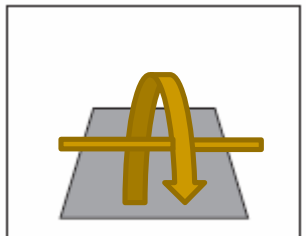
Rotation Axes Examples

Z



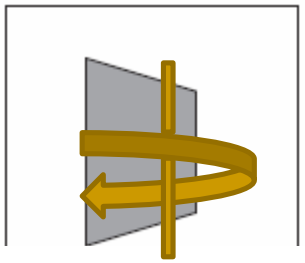
Around center

X



Flip (Rotisserie)

Y



Spin (like a top)

```
float theta = 0.0;
void setup() {
  size(200,200,P3D);
}
void draw() {
  background(255);
  stroke(0);
  fill(175);
  translate(width/2,
            height/2);
  rotateZ(theta);
  rectMode(CENTER);
  rect(0,0,100,100);
  theta + = 0.02;
}
```

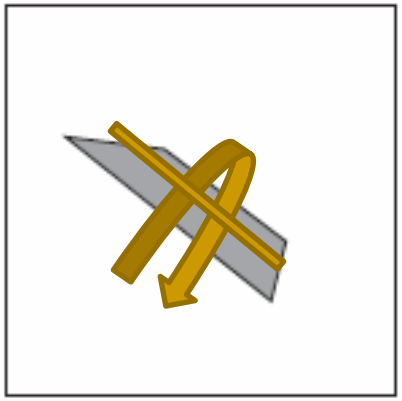
```
float theta = 0.0;
void setup() {
  size(200,200,P3D);
}
void draw() {
  background(255);
  stroke(0);
  fill(175);
  translate(width/2,
            height/2);
  rotateX(theta);
  rectMode(CENTER);
  rect(0,0,100,100);
  theta + = 0.02;
}
```

```
float theta = 0.0;
void setup() {
  size(200,200,P3D);
}
void draw() {
  background(255);
  stroke(0);
  fill(175);
  translate(width/2,
            height/2);
  rotateY(theta);
  rectMode(CENTER);
  rect(0,0,100,100);
  theta + = 0.02;
}
```

Rotation Around Multiple Axes

- Processing can also use rotate functions in combination
  - Rotate in both X (rotisserie) and Y (spinning top)

```
void setup() {
  size(200,200,P3D);
}
void draw() {
  background(255);
  stroke(0);
  fill(175);
  translate(width/2,height/2);
  rotateX(PI*mouseY/height);
  rotateY(PI*mouseX/width);
  rectMode(CENTER);
  rect(0,0,100,100);
}
```



Scale

- In addition to translate( ) and rotate( ), there is one more function, scale( ), that affects the way shapes are oriented and drawn onscreen.
- scale( ) increases or decreases the size of objects onscreen. Just as with rotate( ), the scaling effect is performed relative to the origin's location.
- scale() increases the dimensions of an object relative to the origin by a percentage (1.0 = 100%).
  - scale( ) can also take two arguments (for scaling along the x and y-axes with different values) or three arguments (for the x -, y -, and z -axes).
- Notice how in this example the scaling effect causes the outline of the shape to become thicker.

Class Activity: Scale

- A growing rectangle

```
float r = 0.0;
void setup() {
  size(200,200);
}
void draw() {
  background(0);
  // Translate to center of window
  translate(width/2,height/2);
  // Scale any shapes according to value of r
  scale(r);
  // Display a rectangle in the middle of the screen
  stroke(255);
  fill(100);
  rectMode(CENTER);
  rect(0,0,10,10);
  // Increase the scale variable
  r += 0.02;
}
```

The Matrix (pushMatrix and popMatrix)

- What is the matrix?
- In order to keep track of rotations and translations and how to display the shapes according to different transformations, Processing (and just about any computer graphics software) uses a matrix.
- How matrix transformations work is beyond the scope of this book; however, it is useful to simply know that the information related to the coordinate system is stored in what is known as a transformation matrix.
- When a translation or rotation is applied, the transformation matrix changes. From time to time, it is useful to save the current state of the matrix to be restored later.
  - This will ultimately allow us to move and rotate individual shapes without them affecting others.

The Matrix Revealed

- What is inside the matrix?
- A matrix is a table of numbers with rows and columns. In Processing, a transformation matrix is used to describe the window orientation — is it translated or rotated?
- You can view the current matrix at any time by calling the function printMatrix( ).
- This is what the matrix looks like in its “normal” state, with no calls to translate( ) or rotate( ) :

```
1.0000 0.0000 0.0000
0.0000 1.0000 0.0000
```

: Rotate Two Squares

- But in different ways!

One Square:

```
float theta2 = 0;
void setup() {
  size(200,200,P3D);
}
void draw() {
  background (255);
  stroke(0);
  fill(175);
  rectMode(CENTER);
  translate(150,150);
  rotateY(theta2);
  rect(0,0,60,60);
  theta2 += 0.02;
}
```

```
float theta1 = 0;
float theta2 = 0;
void setup() {
  size(200,200,P3D);
}
void draw() {
  background(255);
  stroke(0);
  fill(175);
  rectMode(CENTER);
  translate(50,50);
  rotateZ(theta1);
  rect(0,0,60,60);
  theta1 += 0.02;
  translate(100,100);
  rotateY(theta2);
  rect(0,0,60,60);
  theta2 += 0.02;
}
```

- Two Squares
  - But there's a problem…

This first call to rotateZ() affects all shapes drawn afterward. Both squares rotate around the center of the first square.

Why didn't work right?

- Running this example will quickly reveal a problem.
  - The first (top left) square rotates around its center .
  - However, while the second square does rotate around its center, it also rotates around the first square!
- Remember, all calls to translate and rotate are relative to the coordinate system's previous state.
- We need a way to restore the matrix to its original state so that individual shapes can act independently.
  - Saving and restoring the rotation/translation state is accomplished with the functions pushMatrix( ) and popMatrix( ) .

Algorithm to Rotate Two shapes

For each square to rotate on its own, we can write the following algorithm (with the new parts bolded).

1. **Save the current transformation matrix.**

   This is where we started, with (0,0) in the top left corner of the window and no rotation.
2. Translate and rotate the first rectangle.
3. Display the first rectangle.
4. **Restore matrix from Step 1 so that it isn't affected by Steps 2 and 3!**
5. Translate and rotate the second rectangle.
6. Display the second rectangle.

Example : Rotate Two Squares

Always put the matrix back the way it was before you translate and rotate.

```
float theta1 = 0;
float theta2 = 0;
void setup() {
  size(200,200,P3D);
}
```

- Push = Save
- Pop = Restore

Original Matrix

```
void draw() {
  background(255);
  stroke(0);
  fill(175);
  rectMode(CENTER);
  pushMatrix();
  translate(50,50);
  rotateZ(theta1);
  rect(0,0,60,60);
  popMatrix();
  pushMatrix();
  translate(150,150);
  rotateY(theta2);
  rect(0,0,60,60);
  popMatrix();
  theta1 += 0.02;
  theta2 += 0.02;
}
```

Algorithm Steps

1. Save matrix
2. Translate, Rotate
3. Display
4. Restore original and save it again
5. Translate and Rotate (2)
6. Display (2)

Put things back the way it was

Best Practices with pushMatrix, popMatrix

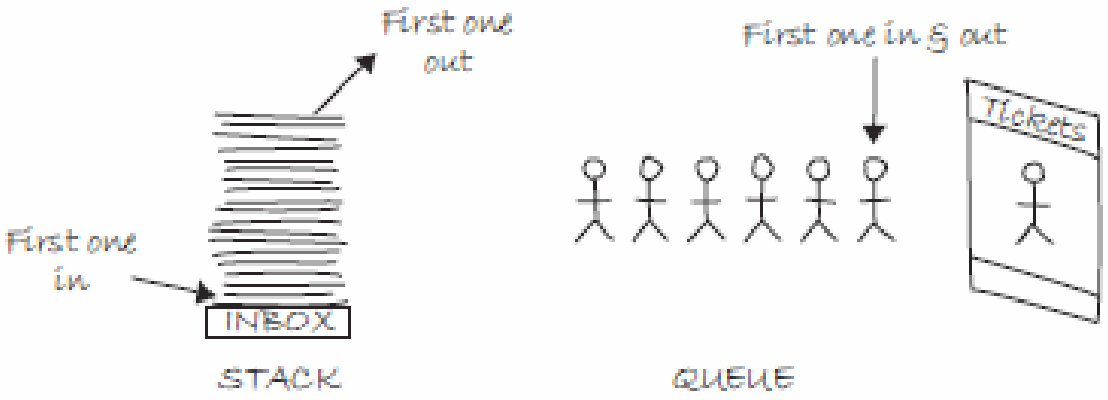
- Although technically not required, it is a good habit to place pushMatrix( ) and popMatrix( ) around the second rectangle as well (in case we were to add more to this code).
- A nice rule of thumb when starting is to use pushMatrix( ) and popMatrix( ) before and after translation and rotation for all shapes so that they can be treated as individual entities.
- In fact, previous example you should really be object oriented, with every object making its own calls to pushMatrix( ), translate( ), rotate( ), and popMatrix( ).
- There must always be an equal amount of calls to both pushMatrix( ) and popMatrix( ), but they do not always have to come one right after the other.

Example : Rotating Objects

```
class Rotater {
  float x,y, theta, speed, w;
  Rotater(float tX, float tY, float tSpeed, float tW) {
    x = tX;      y = tY;
    theta = 0; speed = tSpeed;  w = tW;
  }
  void spin() {
    theta += speed;
  }
  void display() {
    rectMode(CENTER);
    stroke(0);  fill(0,100);
    pushMatrix();
    translate(x,y);
    rotate(theta);
    rect(0,0,w,w);
    popMatrix();
 }
}
```

A short Computer Science Revision

- What is all this about pushing and popping. Sounds pretty violent if you ask me.
- push and pop are computer science terms for using a 'stack' data structure.
  - You 'push' something onto the top of the stack
  - You 'pop' something off of the top of the stack
  - You can keep stacking (pushing) things, but you can only access (pop) what is currently on top of the stack
  - And you can't pop if the stack is empty!
  - A stack is "LIFO"
    - Last In, First Out
  - A queue is "FIFO"
    - First In, First Out



: Solar System

```
void draw() {
  background(255);
  stroke(0);
  // Center the sun
  translate(width/2,height/2)
  fill(255,200,50);
  ellipse(0,0,20,20);
  // Earth rotates around the
  pushMatrix();  // Save Sun
  rotate(theta);
  translate(50,0);
  fill(50,200,255);
  ellipse(0,0,10,10);
```

- Nested pushMatrix and popMatrix
  - Earth rotates around the sun
  - Moons rotate around the Earth

```
  // Moon #1 rotates around Earth
  pushMatrix();  // Save Earth
  rotate(-theta*4);
  translate(15,0);
  fill(50,255,200);
  ellipse(0,0,6,6);
  popMatrix();  // Restore Earth
  // Moon #2 rotates around Earth
  pushMatrix(); // Save Earth
  rotate(theta*2);
  translate(25,0);
  fill(50,255,200);
  ellipse(0,0,6,6);
  popMatrix();  // Restore Earth
  popMatrix();  // Restore Sun
  theta + = 0.01;
}
```

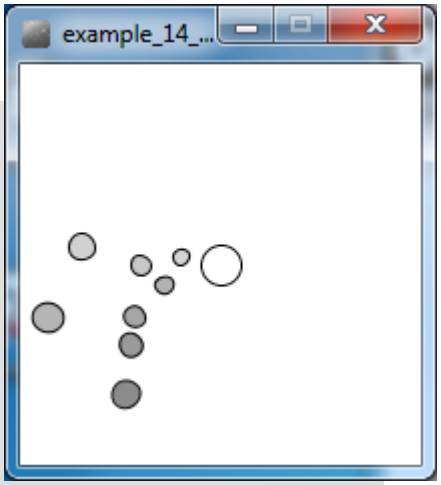
- Why save the Sun matrix?
  - It works the same without saving
  - In case we add another planet!

: Rotating Planet Object

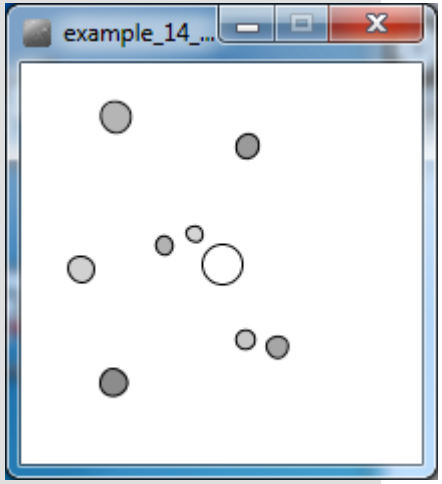
```
class Planet {
  float theta;       // Rotation around sun
  float diameter;    // Size of planet
  float distance;    // Distance from sun
  float orbitspeed; // Orbit speed
  Planet(float distance_, float diameter_) {
    distance = distance_;   diameter = diameter_;
    theta = 0;  orbitspeed = random(0.01,0.03);
  }
  void update() {
    theta += orbitspeed; // Increment the angle to rotate
  }
  void display() {
    pushMatrix();  // Save Sun matrix
    rotate(theta); // rotate orbit
    translate(distance,0); // translate out distance
    stroke(0);
    fill(orbitspeed * 8000); //dws
    ellipse(0,0,diameter,diameter);
    popMatrix(); // Afer we are done, restore Sun matrix!
}
```

Pop the matrix back the way it was after you translate and rotate.

: Pushing Planets




```
Planet[] planets = new Planet[8];  // No Pluto
void setup() {
  size(200,200);  smooth();
  for (int i = 0; i < planets.length; i++) {
    planets[i] = new Planet(20 + i*10,i + 8);
  }
}
void draw() {
  background(255);
  // Drawing the Sun: no push/pop if Planets put it back
  translate(width/2,height/2);
  stroke(0);
  fill(255);
  ellipse(0,0,20,20);
  // Drawing all Planets
  for (int i = 0; i < planets.length; i++) {
    planets[i].update();
    planets[i].display();
  }
}
```



Lab Work: in Groups

- You need to complete this by today.
  - How would you add moons to the planets? Hint: Write a Moon class that is virtually identical to the Planet.
  - Then, incorporate a Moon variable into the Planet class.
- //Extended Version
  - Extend the solar system example into three dimensions. Try using sphere() or box() instead of ellipse().
  - Note sphere() takes one argument, the sphere's radius.
  - box() can take one argument (size, in the case of a cube) or three arguments (width, height, and depth.)