# Shapes + Interpolation

# Transformations

Calls in Processing that allow us to move the coordinate system around---changes the interpretation of positions.

Useful transformations:

- `translate`
- `rotate`
- `shearX`
- `shearY`
- `scale`

# Recalling Transformations

Sometimes we might want to save the current transformation: can do with with `pushMatrix()`

The most recent transformation saved with `pushMatrix()` can be restored by calling `popMatrix()`

We should generally try to make sure every `pushMatrix()` is matched with a `popMatrix()`

# Questions

## Is there a way to iterate over the transformations stored by `pushMatrix()`?

Not without accessing the Necronomicon.

(The transformation stack is a Processing internal, meaning you have to write some custom Java code to get to it).

## Can we create our own custom transformations?

See `applyMatrix()` and `resetMatrix()`.

## How do we know which transformations to apply first?

Try it and see.

There are mathematically backed ways of doing this, but in practice, you do it enough times that you get an intuition for how it works.

## Do you have to use linear algebra to do these transforms?

In Processing, no. The transformation functions we studied basically do all the transforms you need to worry about.

# How are successive transformations represented?

$$\begin{bmatrix} a_1 & b_1 \\ c_1 & d_1 \end{bmatrix} \left( \begin{bmatrix} a_2 & b_2 \\ c_2 & d_2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \right) = \left( \begin{bmatrix} a_1 & b_1 \\ c_1 & d_1 \end{bmatrix} \begin{bmatrix} a_2 & b_2 \\ c_2 & d_2 \end{bmatrix} \right) \begin{bmatrix} x \\ y \end{bmatrix}$$

$$= \begin{bmatrix} a_3 & b_3 \\ c_3 & d_3 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

**Matrix Multiplication!**

# Are there any other uses for matrices in graphics programming aside from linear transforms?

Meshes can be represented as matrices.

Lights can be represented as matrices.

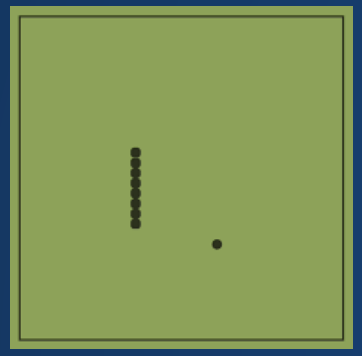
Physics operations can be represented as matrices.

## The world of graphics floats on matrices.

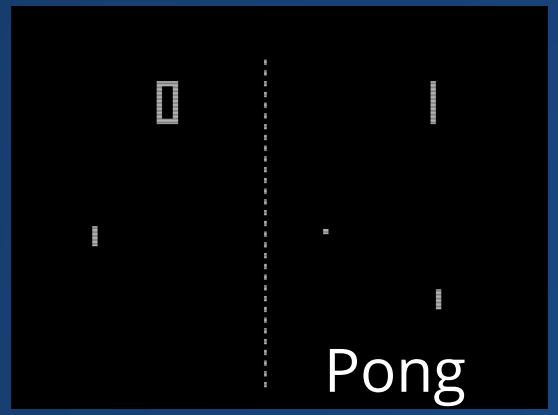
In this class, we will mostly be able to dodge them, since Processing handles a lot of stuff for us.

# With all the stuff we've learned, could we make a simple game?

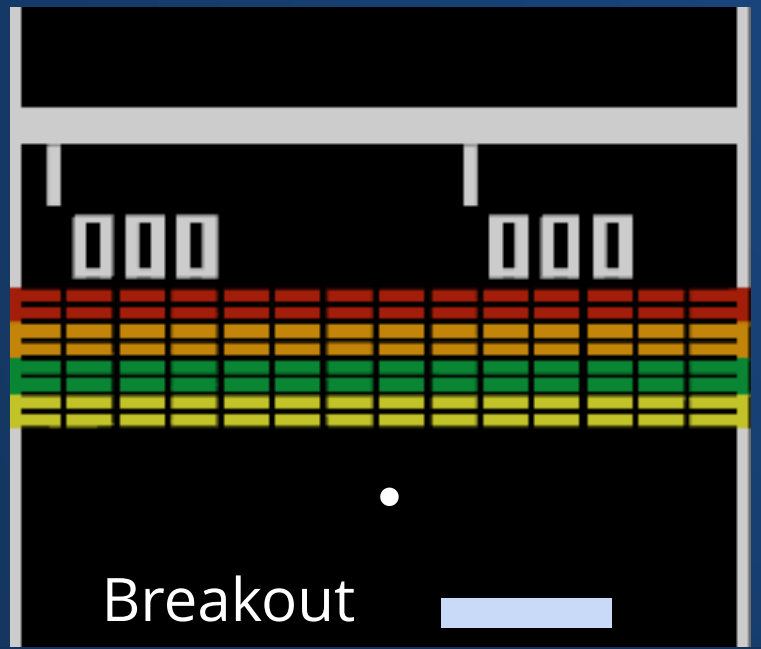
After today's class, you could probably feasibly write the following retro games:



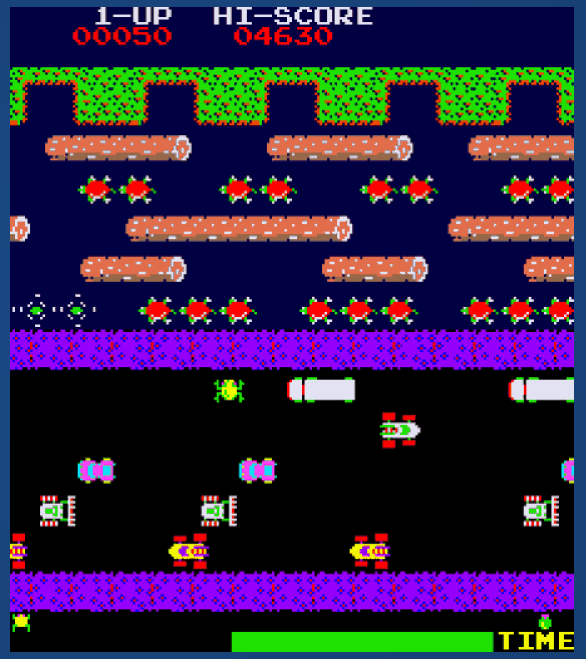
Snake



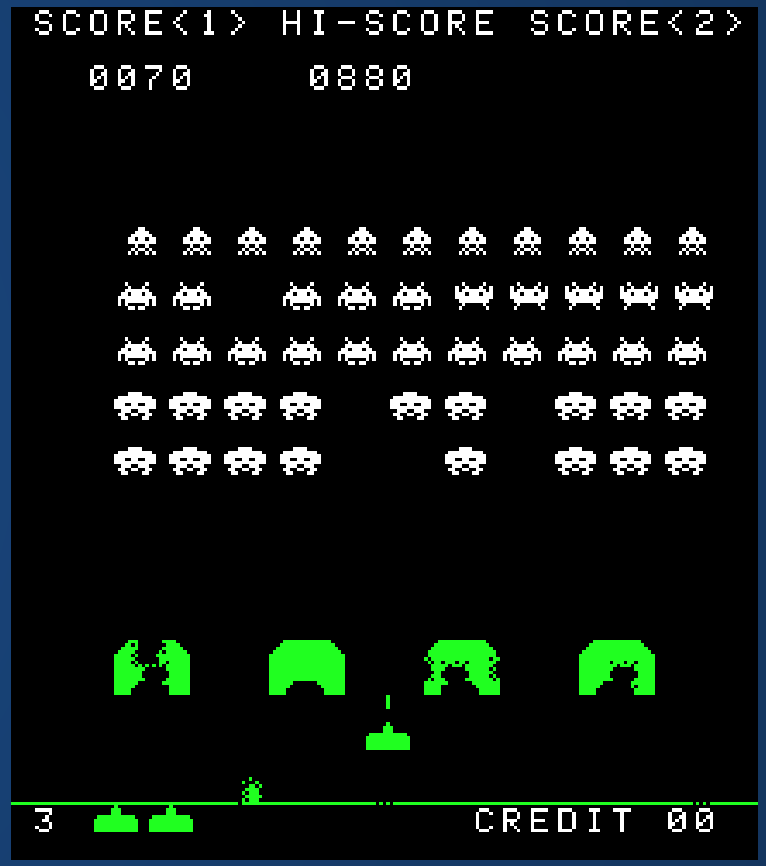
Pong



Breakout




Frogger




Space Invaders

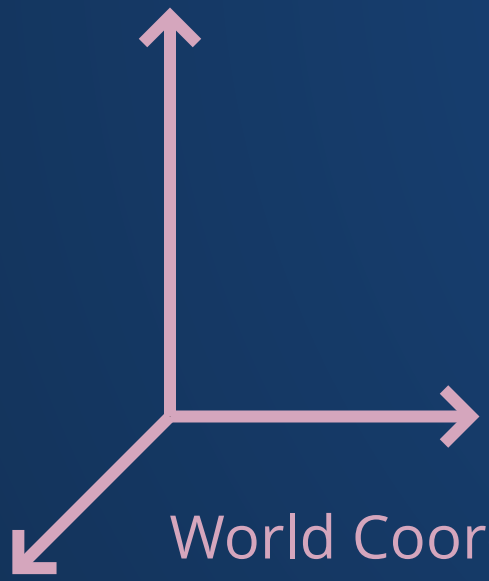
...and possibly Pac-Man or Tetris?

# Thinking About Transforms

# Another Way of thinking of Transforms

Camera Coordinate System

Object Coordinate System

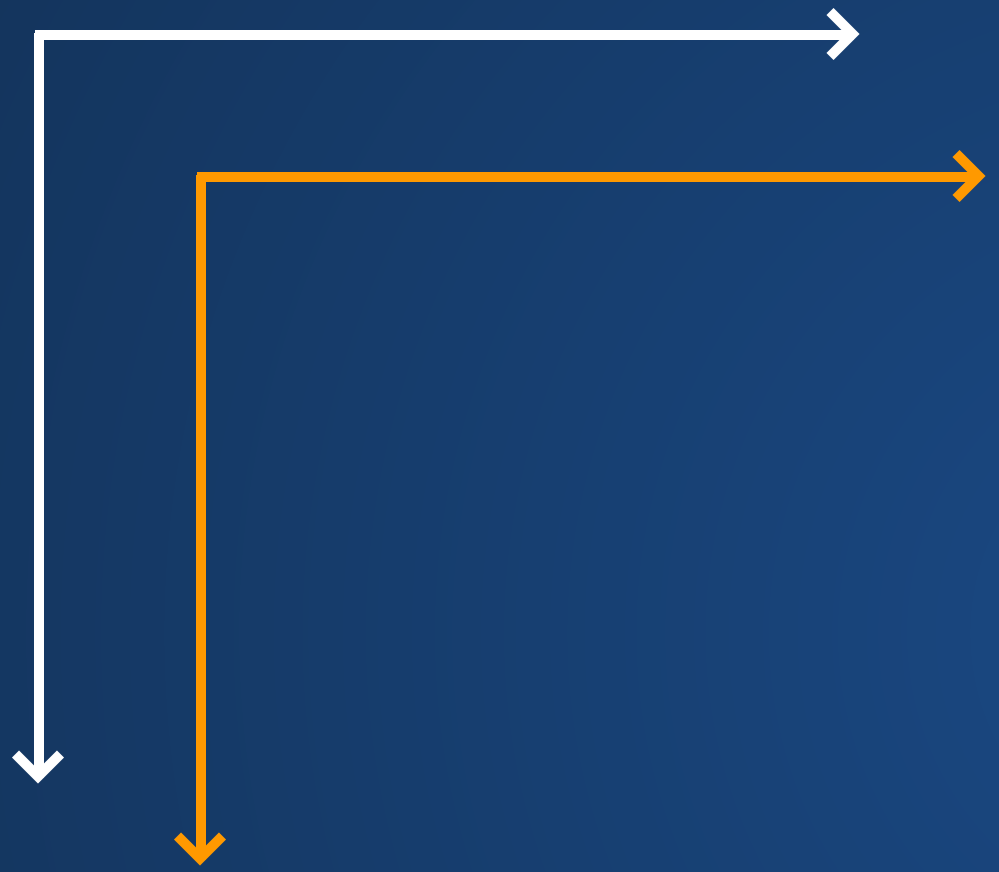


World Coordinate System

Transforms **modify the current coordinate system**!

Screen Coordinates "Object Coordinates"

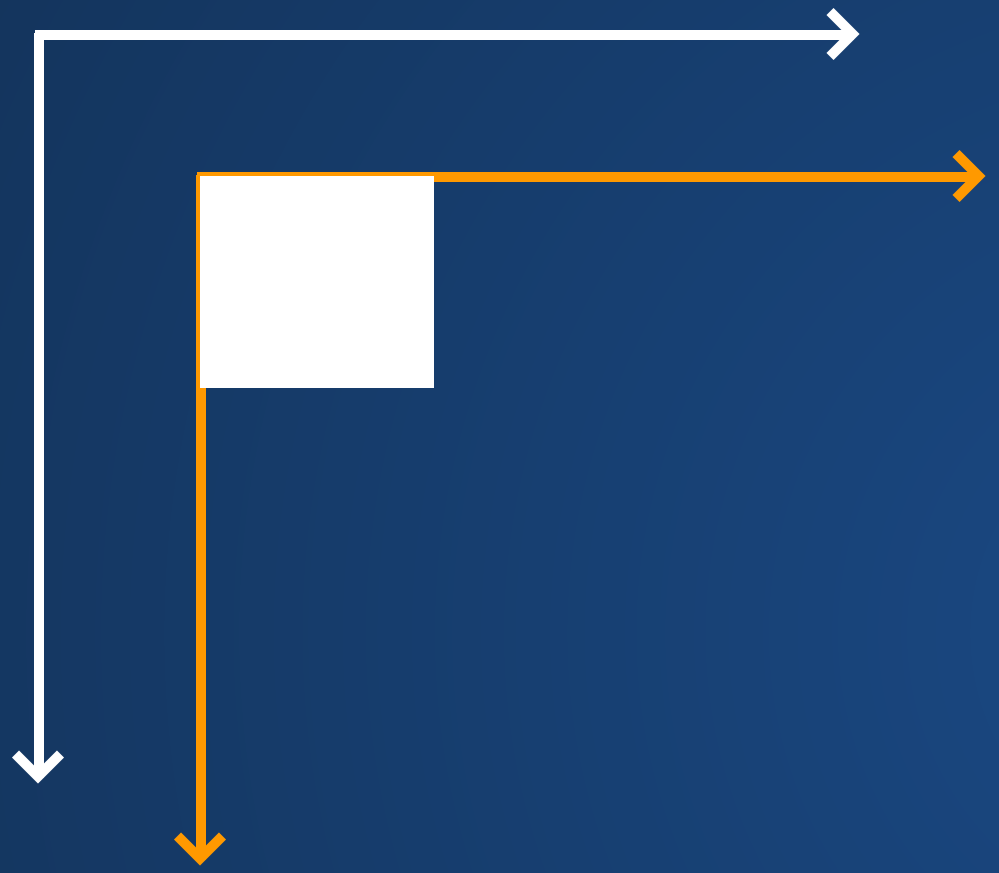
translate(100, 100);



Screen Coordinates "Object Coordinates"

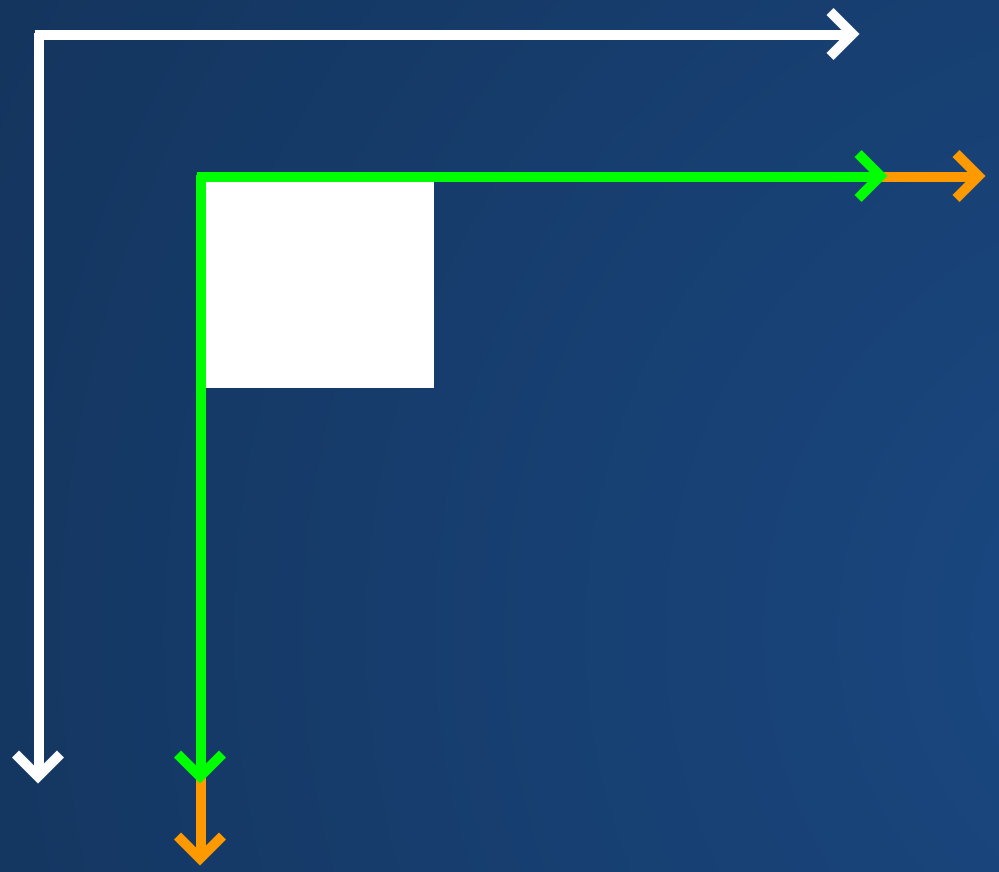
translate(100, 100);

rect(0, 0, 50, 50);



Screen Coordinates "Object Coordinates"

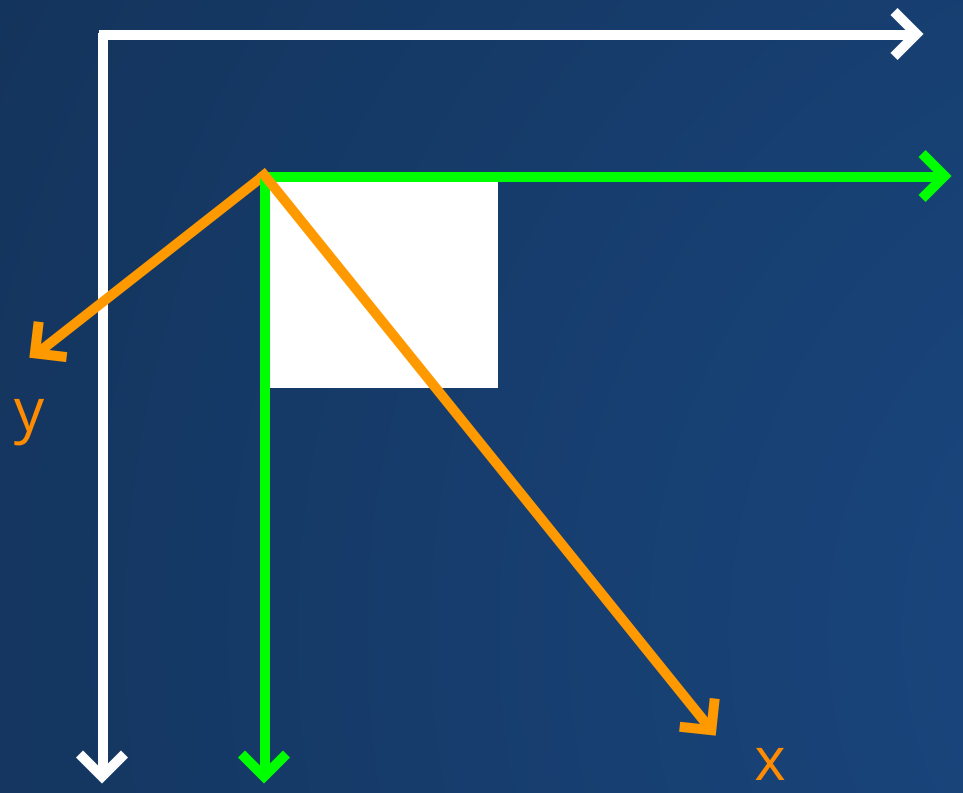
```
translate(100, 100);
rect(0, 0, 50, 50);
pushMatrix();
```



Screen Coordinates "Object Coordinates"

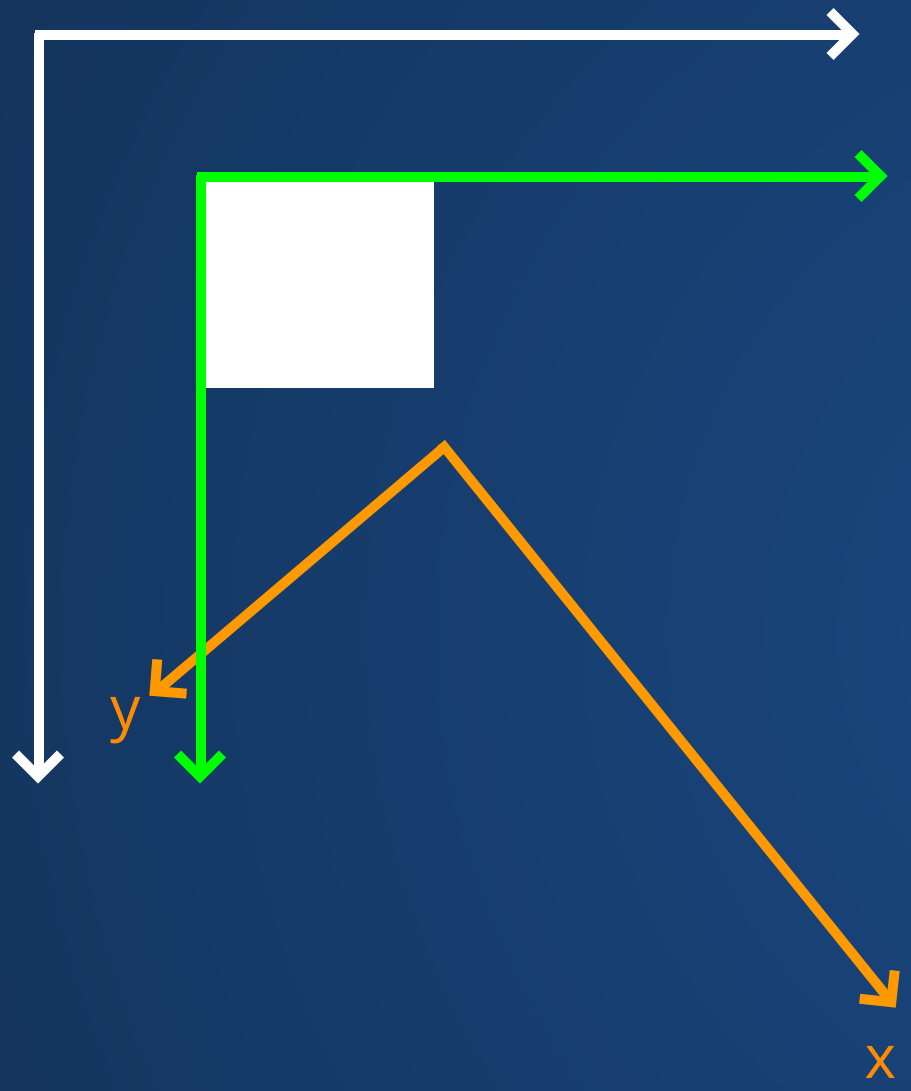
```
translate(100, 100);
rect(0, 0, 50, 50);
pushMatrix();
rotate(PI/4);
```

Screen Coordinates "Object Coordinates"



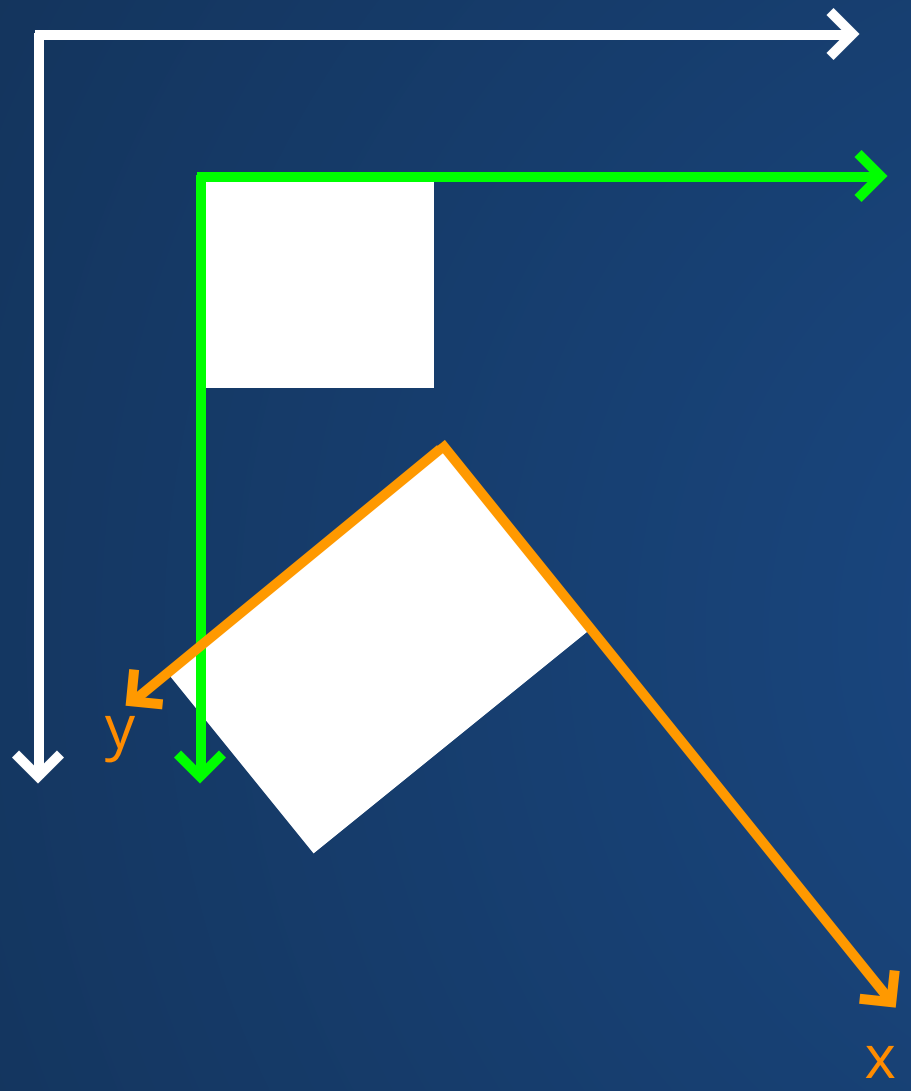

```
translate(100, 100);
rect(0, 0, 50, 50);
pushMatrix();
rotate(PI/4);
translate(100, 0);
```

Screen Coordinates "Object Coordinates"

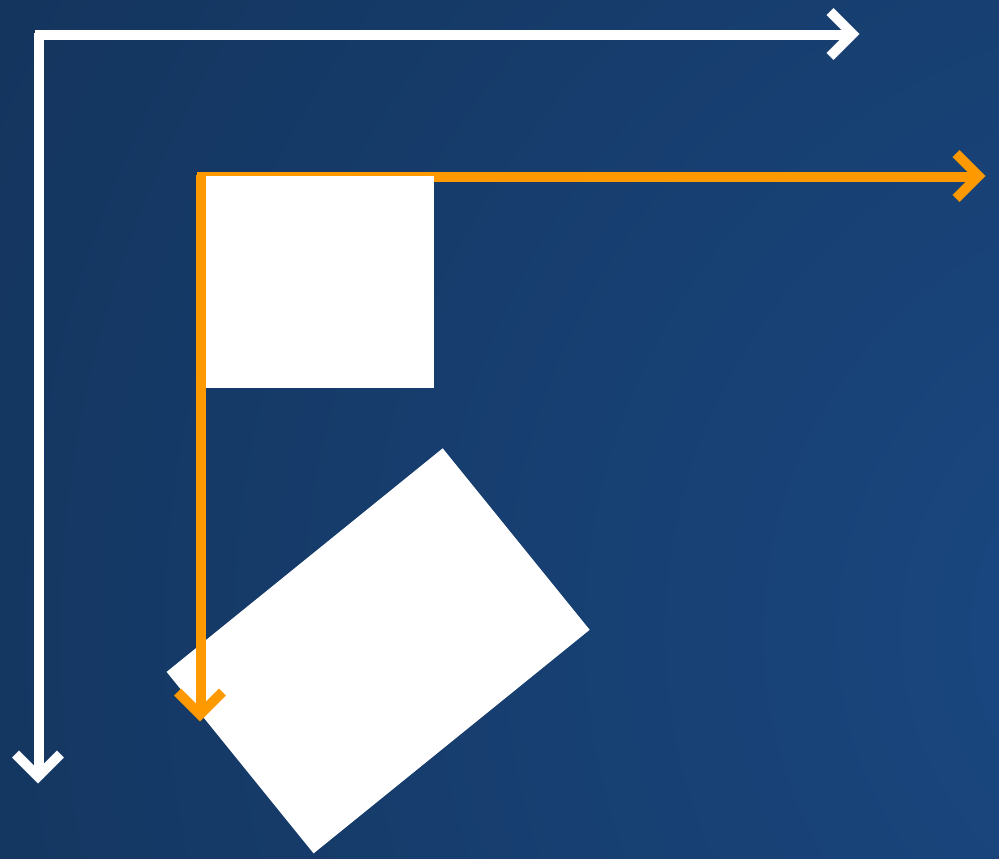



```
translate(100, 100);
rect(0, 0, 50, 50);
pushMatrix();
rotate(PI/4);
translate(100, 0);
rect(0, 0, 50, 100);
```

Screen Coordinates "Object Coordinates"



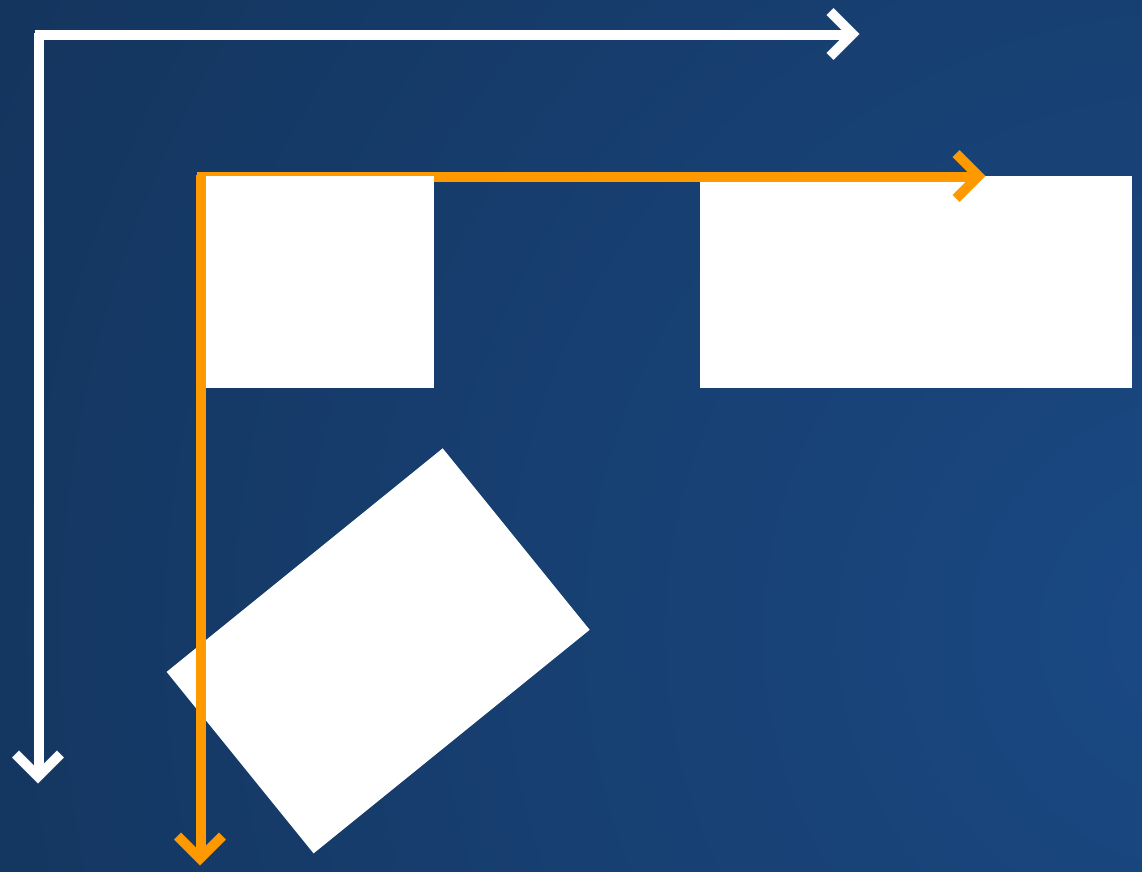

```
translate(100, 100);
rect(0, 0, 50, 50);
pushMatrix();
rotate(PI/4);
translate(100, 0);
rect(0, 0, 50, 100);
popMatrix();
```



Screen Coordinates "Object Coordinates"

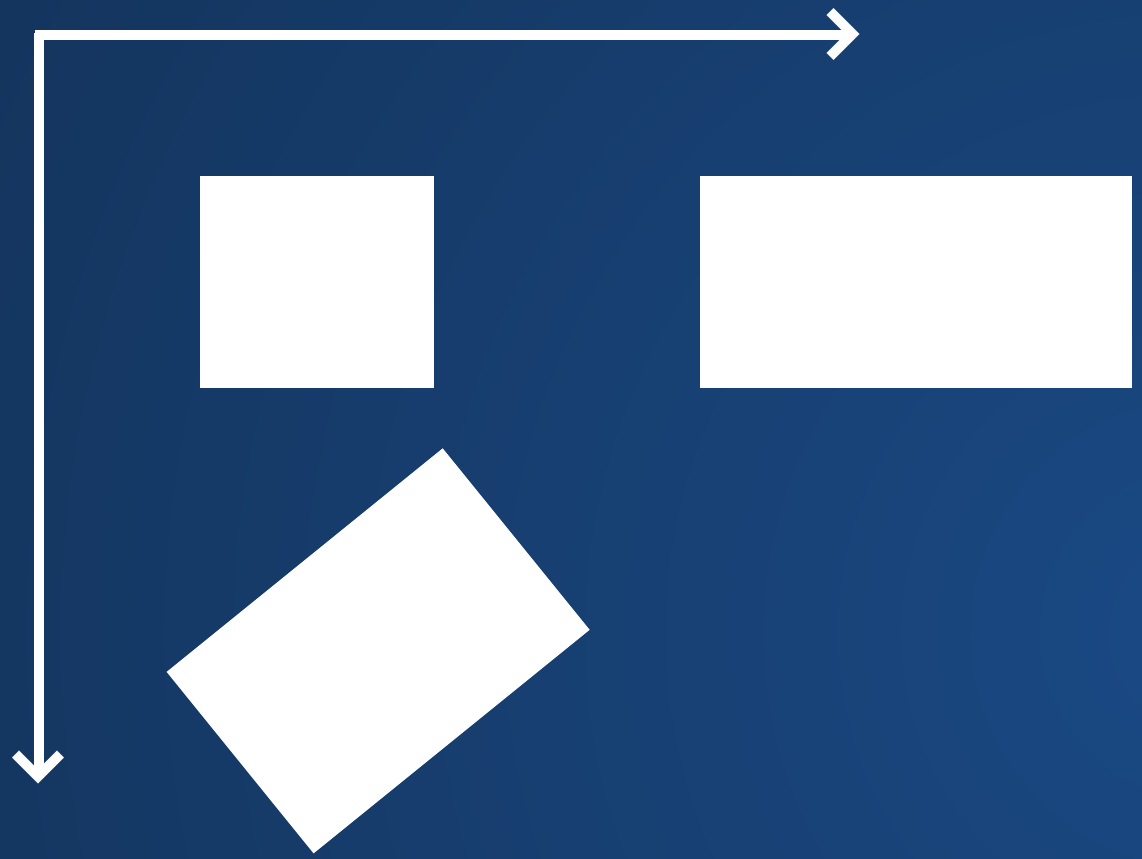
```
translate(100, 100);
rect(0, 0, 50, 50);
pushMatrix();
rotate(PI/4);
translate(100, 0);
rect(0, 0, 50, 100);
popMatrix();
rect(100, 0, 100, 50);
```

Screen Coordinates "Object Coordinates"



```
translate(100, 100);
rect(0, 0, 50, 50);
pushMatrix();
rotate(PI/4);
translate(100, 0);
rect(0, 0, 50, 100);
popMatrix();
rect(100, 0, 100, 50);
```

Screen Coordinates "Object Coordinates"

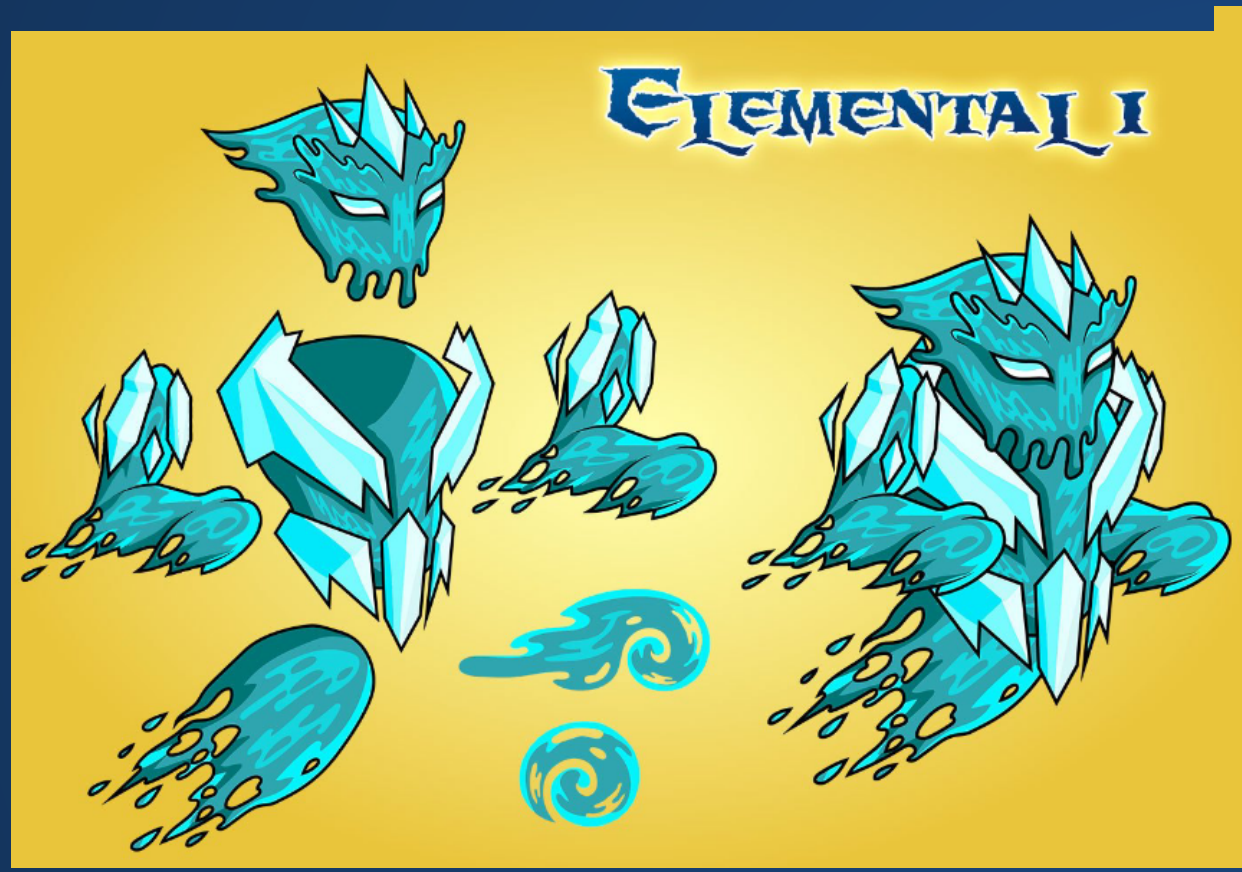


```
translate(100, 100);
rect(0, 0, 50, 50);
pushMatrix();
rotate(PI/4);
translate(100, 0);
rect(0, 0, 50, 100);
popMatrix();
rect(100, 0, 100, 50);
```

# Demo: Bike and Car


Animations Included:
x idle
x attack
x hurt
x jump
x run
x walk
x die
.PNG

# How to draw the individual pieces?




# Creating Shapes from Vertices

1. Use `beginShape()` to start the shape
2. Specify points defining the shape with `vertex()`
3. Complete the shape with `endShape()`

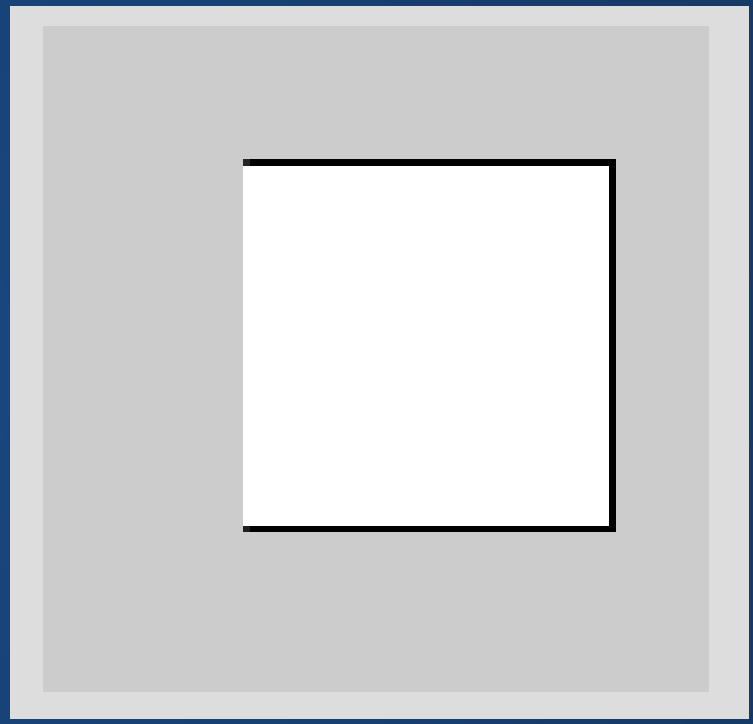
`fill()`, `stroke()`, `noFill()`, `noStroke()` and `strokeWeight()` control the shape attributes

`endShape()` ends the shape

`endShape(CLOSE)` ends the shape **and** closes it by drawing a line back to the starting point

# Example

```
1 beginShape();
2 vertex(30, 20);
3 vertex(85, 20);
4 vertex(85, 75);
5 vertex(30, 75);
6 endShape();
```

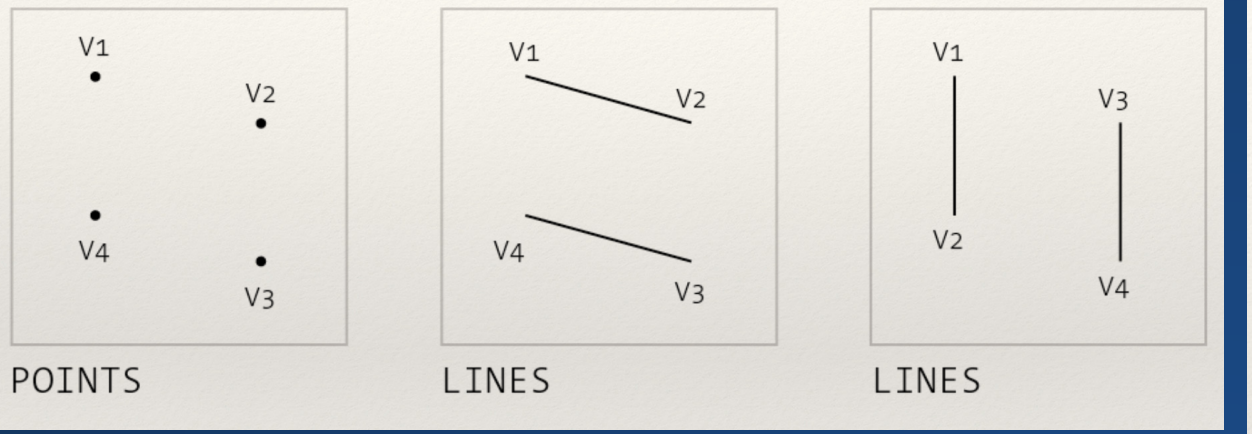


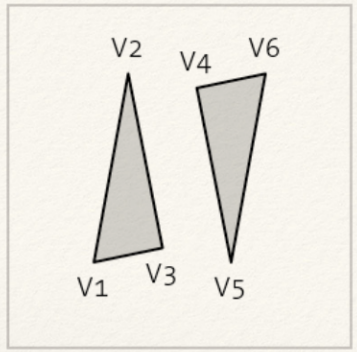
`endShape(CLOSE)` would add a line connecting back to the first vertex

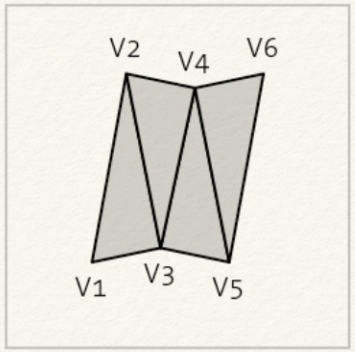
# Geometry

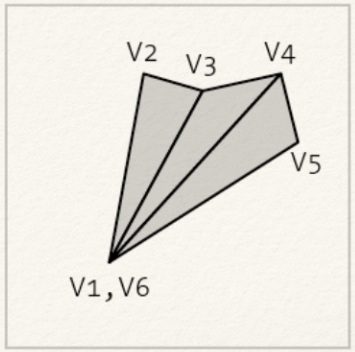
`beginShape()` accepts different parameters to define the drawing of vertex data

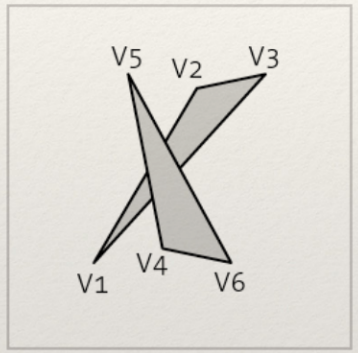
- POINTS
- LINES
- TRIANGLES
- TRIANGLE_STRIP
- TRIANGLE_FAN
- QUADS
- QUAD_STRIP

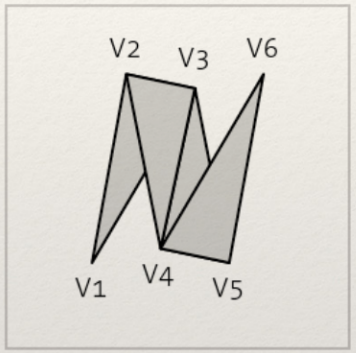

V1
V2
V4
V3
POINTS
V1
V2
V4
V3
LINES
V1
V2
V3
V4
LINES

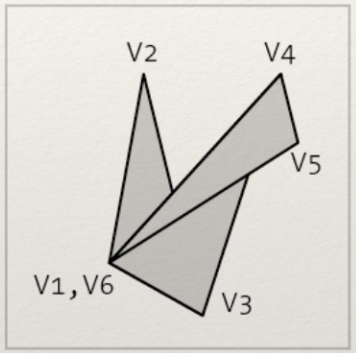

V2
V4
V6
V1
V3
V5
TRIANGLES

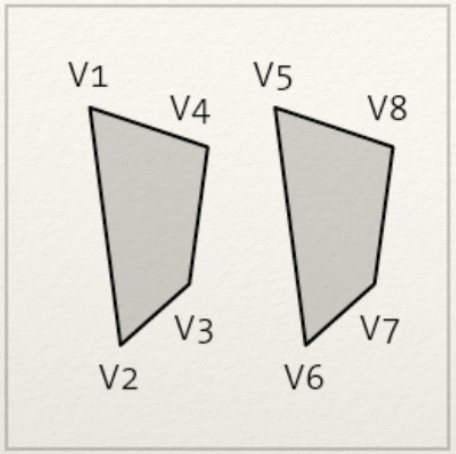

V2
V4
V6
V1
V3
V5
TRIANGLE_STRIP

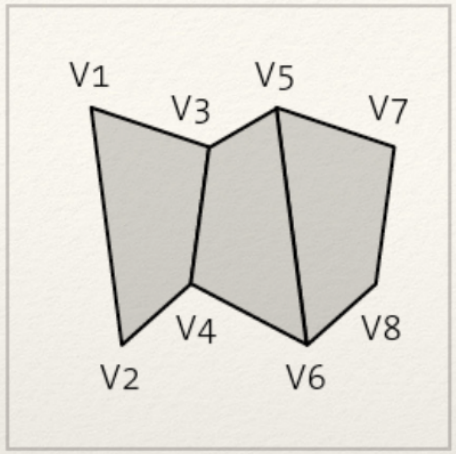

V2
V3
V4
V5
V1,V6
TRIANGLE_FAN

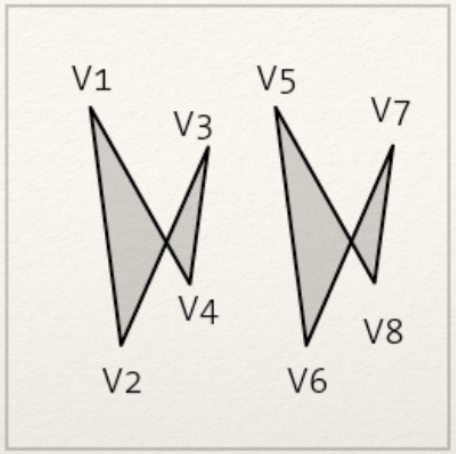

V5
V2
V3
V1
V4
V6
TRIANGLES

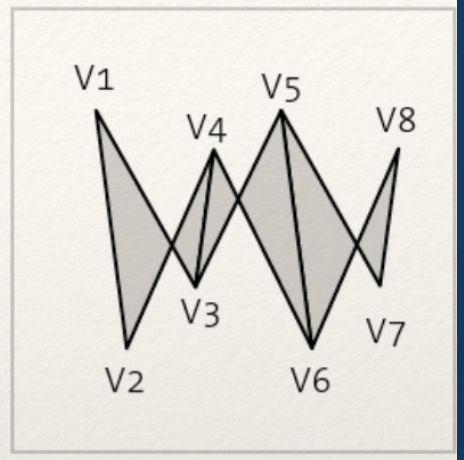

V2
V3
V6
V1
V4
V5
TRIANGLE_STRIP


V2
V4
V5
V1,V6
V3
TRIANGLE_FAN


V1
V4
V5
V8
V3
V7
V2
V6
QUADS


V1
V3
V5
V7
V4
V8
V2
V6
QUAD_STRIP


V1
V3
V5
V7
V4
V8
V2
V6
QUADS

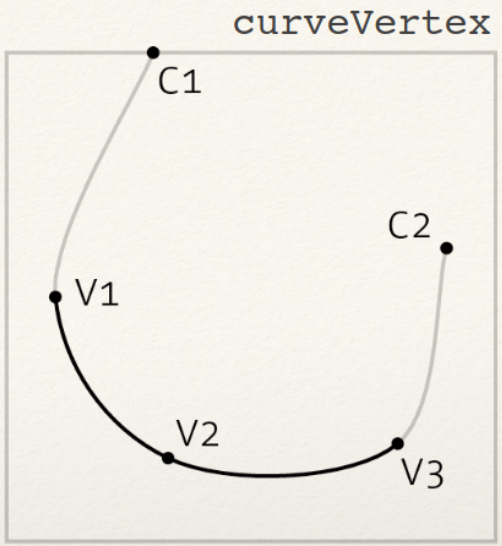

V1
V4
V5
V8
V3
V7
V2
V6
QUAD_STRIP

# Curves

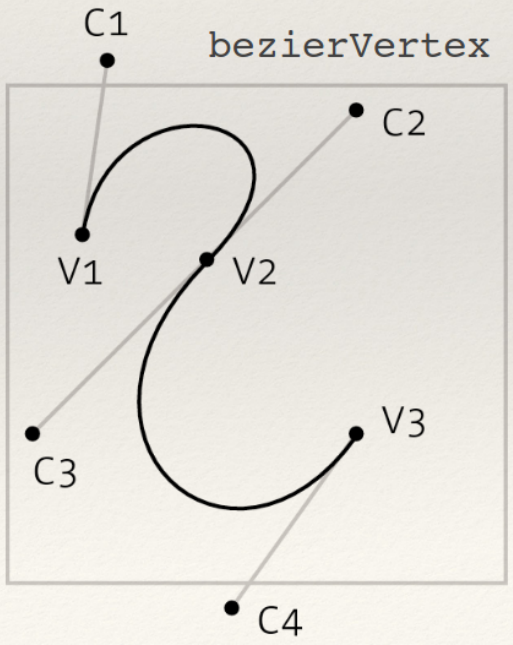
If there is no parameter passed to `beginShape()`, we can also draw curves using special vertex functions.

- `curveVertex()` draws curves which are defined by a first control point, intermediate points, and a second control point.
- `bezierVertex()` is defined by an initial anchor point, then triples of (control, control, anchor) points.

```
beginShape();
curveVertex(C1);
curveVertex(V1);
curveVertex(V2);
curveVertex(V3);
curveVertex(C2);
endShape();
```




```
beginShape();
vertex(V1);
bezierVertex(C1, C2, V2);
bezierVertex(C3, C4, V3);
endShape();
```



# Contours

We can also cut holes into shapes via contours.

Call `beginContour()` and `endContour()` inside of a `beginShape()/endShape()` block

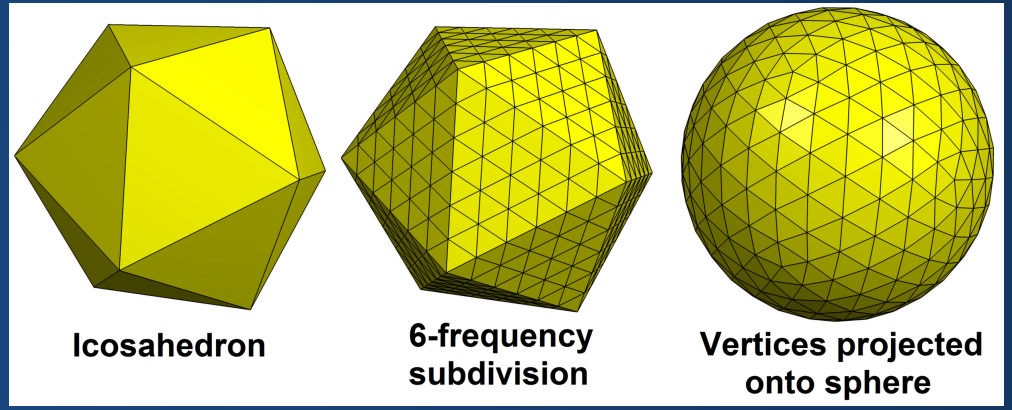
Shape placed over Shape

Contour removing from Shape

# Defining Shapes with Vertices

...is **really** hard.

Even harder in 3D. Quick, what are the points needed for an approximate sphere?




**Solution: don't. Design shapes using graphical interfaces (like Illustrator or Maya) and then import them!**

# PShape

A class for storing Shapes

- Allows us to load and display SVGs and OBJs
  - SVG is an open standard for storing 2D vector graphics
  - OBJ is a standard for storing 3D vector geometry

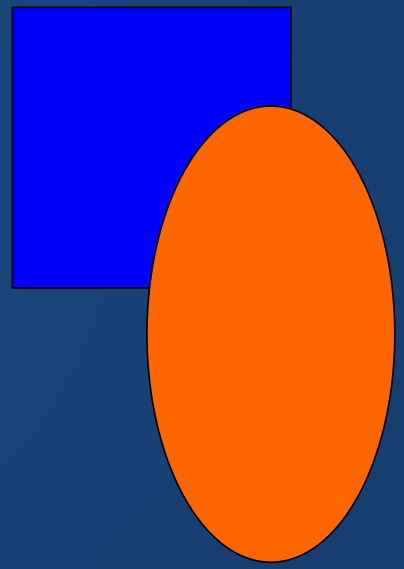
- Call `loadShape(filename)` to load a file into a PShape

- `shape(PShape, x, y)` or `shape(PShape, x, y, width, height)` to display it

## Pretty similar to `PImages`!

# Example SVG

```
<g
   inkscape:label="Layer 1"
   inkscape:groupmode="layer"
   id="layer1">
  <rect
     style="fill:#0000ff;stroke:#00000
     id="rect234"
     width="76.99118"
     height="77.594948"
     x="17.603966"
     y="36.327324" />
  <ellipse
     style="fill:#ff6600;stroke:#00000
     id="path498"
     cx="89.078629"
     cy="126.6972"
     rx="34.270332"
     ry="63.077023" />
</g>
```

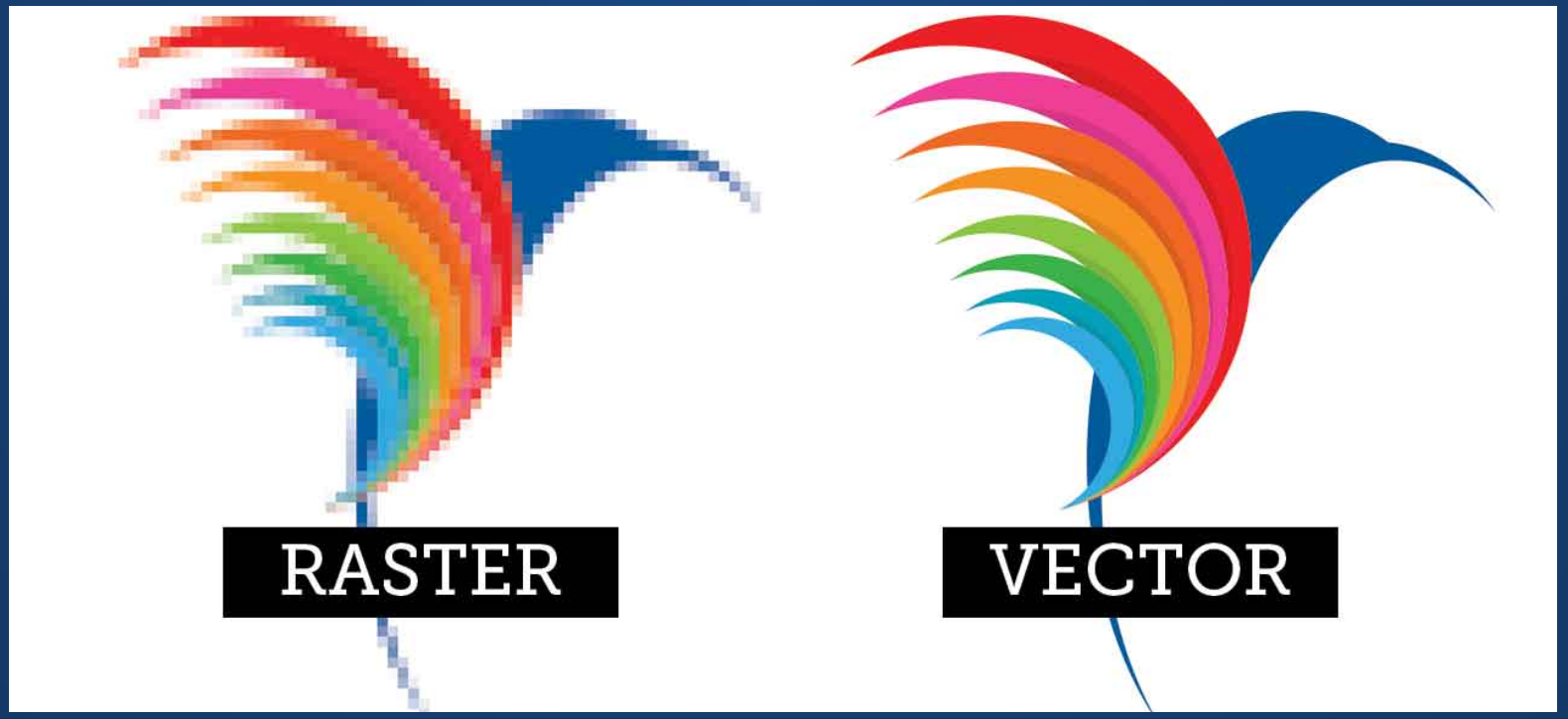


# Aside: Vector vs Raster Images

We saw that images are grids of pixels, where each pixel has a color.

These are more specifically known as *raster* images.

Formats like SVG are known as *vector* images (or vector graphics).



# Grouping PShapes

We can also group multiple `PShapes` to make a more complex shape.

```
1 PShape person = createShape(GROUP);
2 PShape head = createShape(ELLIPSE, 25, 25,50, 50);
3 PShape body = createShape(RECT, 0, 50, 50,100);
4 person.addChild(head);
5 person.addChild(body);
6 shape(person);
```

# Hands-On: Using PShapes

1. Create a free-hand shape using `vertex()` points
2. Create a Shape using `curveVertex()`
3. Load an SVG from https://www.svgrepo.com/ into a PShape and display it to the screen.

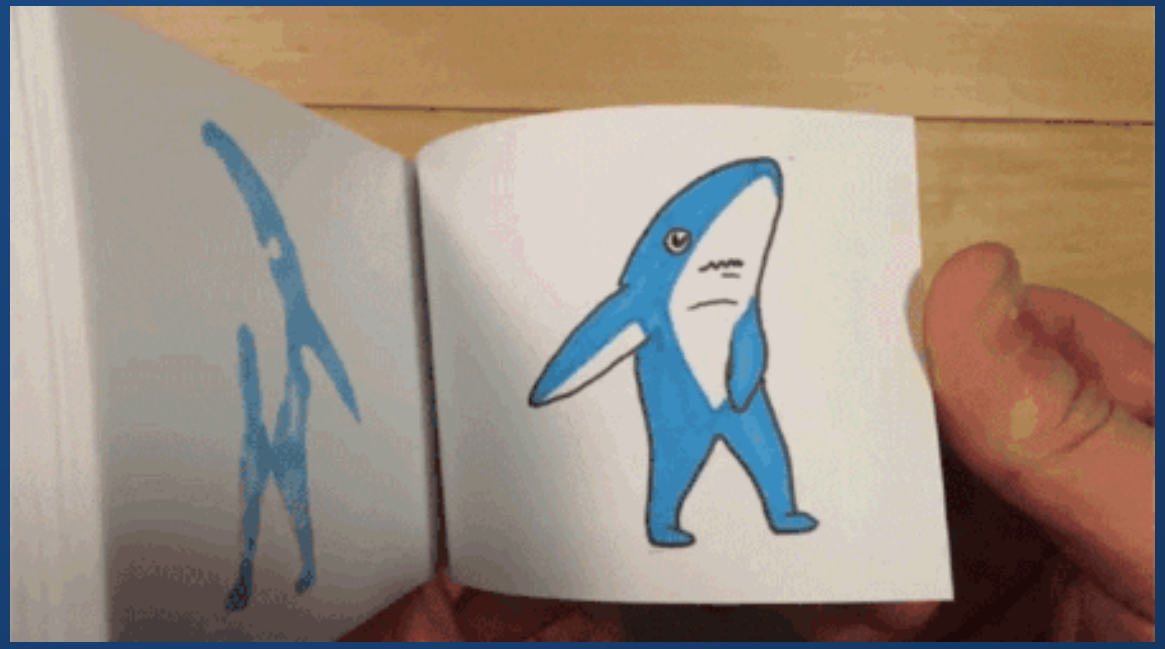
Remember, most of these will need to be sandwiched inside `beginCurve()` and `endCurve()`

# Animating a Scene

So far, we've focused on how to construct static images. Static images are so 15,000 B.C.E. Let's make moving pictures!

Static images constructed with 15,000 B.C.E. technology.

One way to make an animation would be to define a sequence of static images that form the final animation.



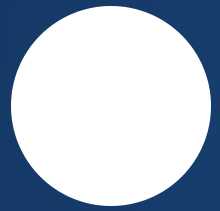
# Tweening

- "In-betweening"
- Used in both traditional and digital animation
- Define distinct keyframes and automatically create intermediates derived from them or *interpolate* between keyframes

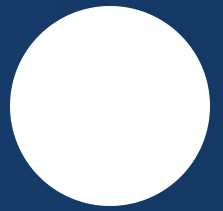
Chu and Lee, 2009

# Example

Key configurations: $t = 0$ and $t = 100$



$t = 0$



$t = 100$

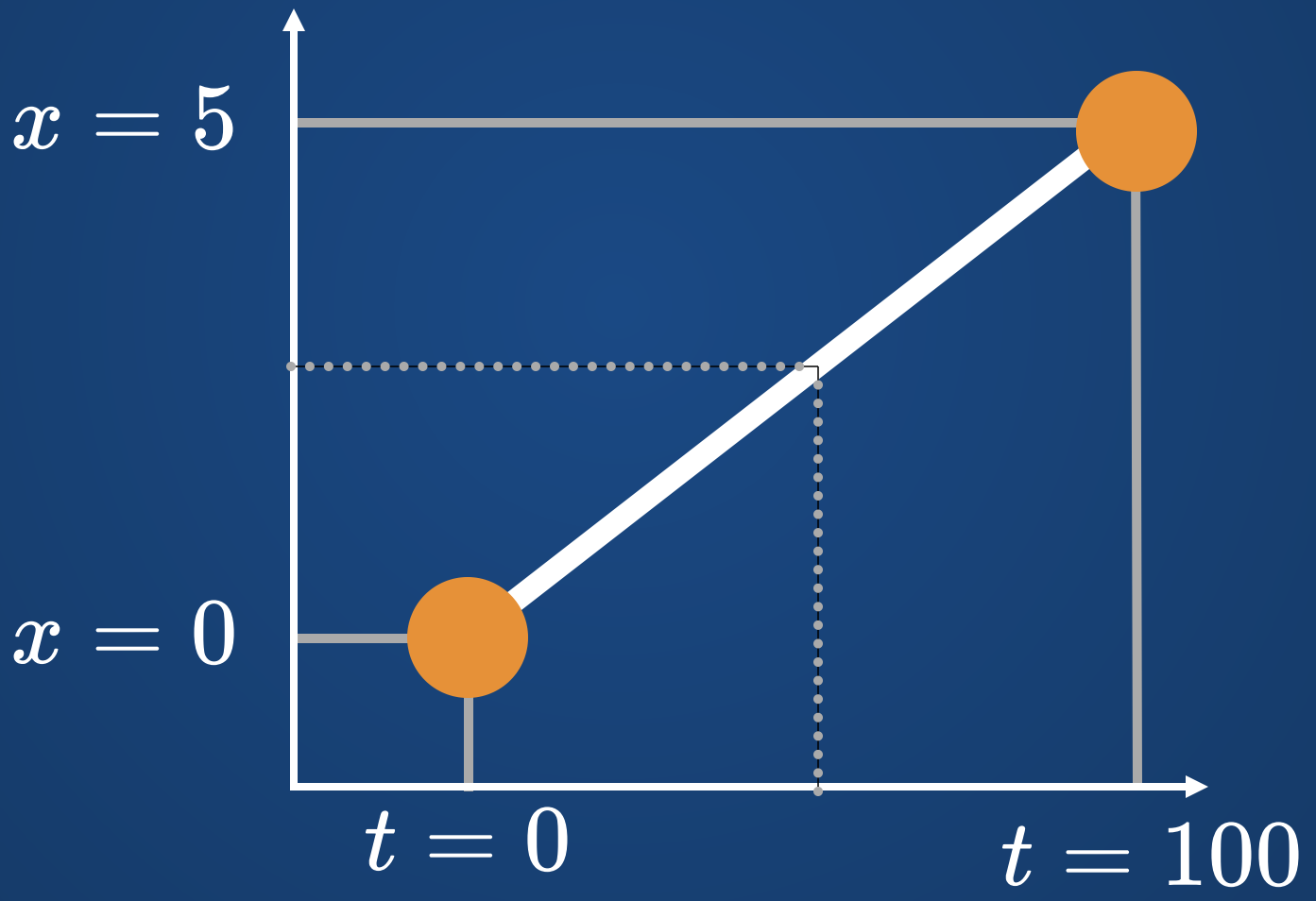
Where should the ball be at $t = 50$? What about $t = 10$?

# Linear Interpolation

Given a starting and ending target, we can change a value by a fixed amount each timestep. This way, the change happens at a linear rate (i.e. if we wait twice as long, the change is twice as large).

This is *linear interpolation*, sometimes nicknamed "lerp".

# Linear Interpolation



# Linear Interpolation in Processing

Processing has a function called `lerp()`

Usage: `lerp(v1, v2, t)` where t is between 0 and 1

$$\text{lerp}(v_1, v_2, t) = v_1(1-t) + v2(t)$$

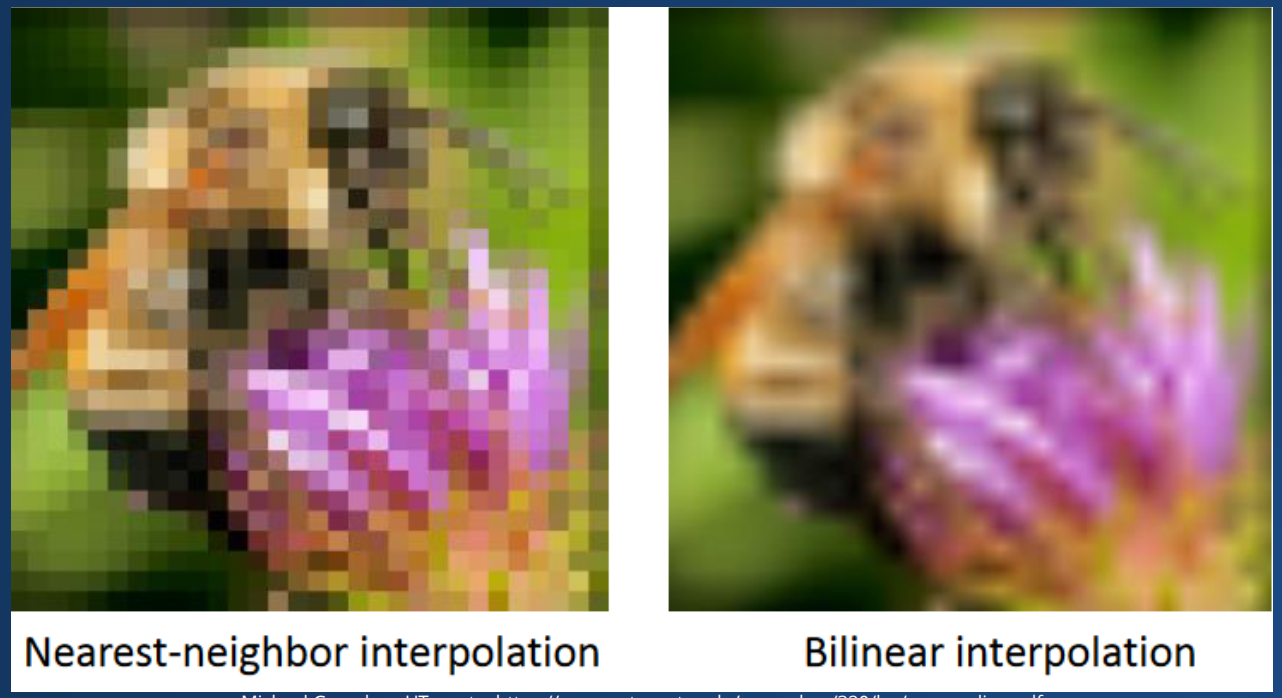
Does not actually *apply* the lerp over time---you need to do that yourself! This just tells you what the result should be.

# An Example in Processing

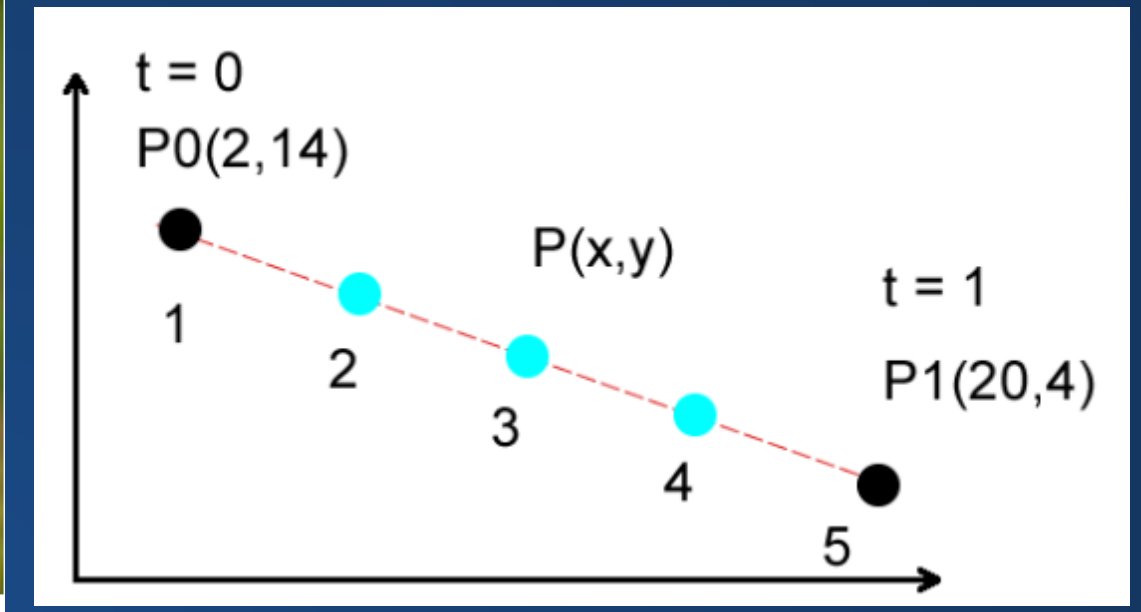
```
1  float xBegin = 0.0;
2  float yBegin = 0.0;
3  float xEnd = 500.0;
4  float yEnd = 300.0;
5
6  float time = 0.0;
7  float timeEnd = 100.0;
8
9  float x = xBegin;
10 float y = yBegin;
11
12 void setup(){
13   size(500, 500);
14 }
15
16 void draw(){
17   time++;
18   if(time > 100){
19     noLoop();
20     return;
21   }
22   ellipse(x, y, 50, 50);
23   // How do we get the new x and y?
24
25 }
```

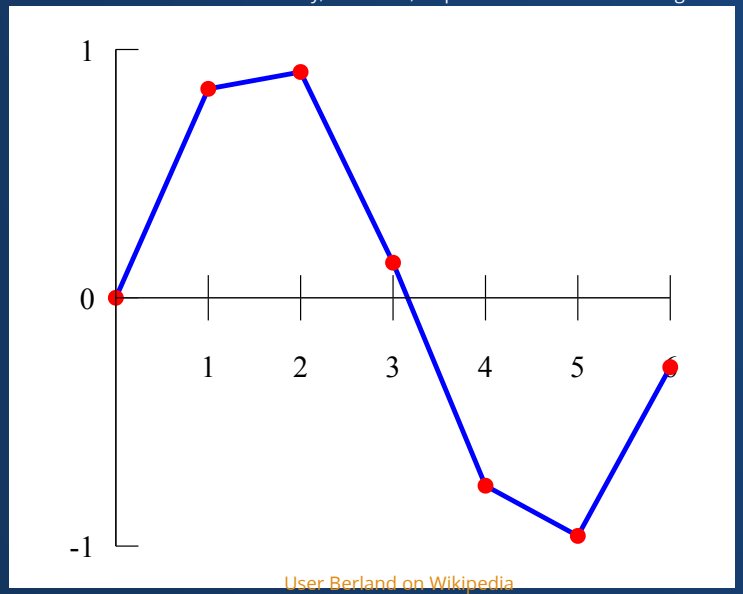
# Uses for Lerp

# Any time you want values "in-between" in a simple manner.




Michael Guerzhoy, UToronto, https://www.cs.toronto.edu/~guerzhoy/320/lec/upsampling.pdf



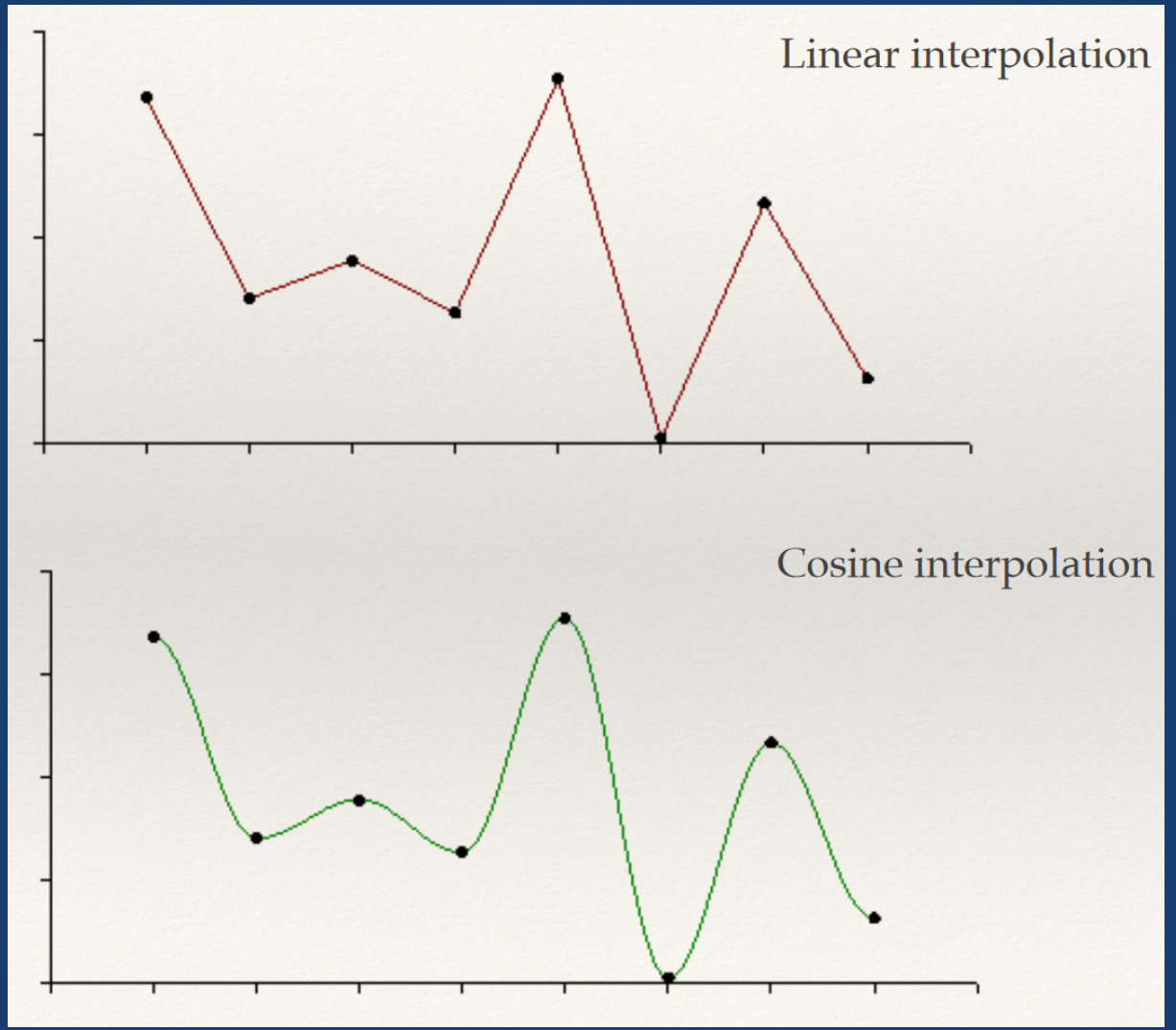

User Berland on Wikipedia

# Cosine Interpolation

Linear Interpolation can have sharp discontinuities at each point. Cosine interpolation can smooth these out without requiring more data points.

$$t_2 = \frac{1-\cos(\pi t)}{2}$$

$$v(t) = (1 - t_2)v_1 + t_2 v_2$$




paulbourke.net

# Hands-On: Interpolation

1. Create a global counter `time` which runs from 0.0 to 1.0 in increments of 0.01. Increment time at the end of your `draw()` loop (and roll it back over to 0.0 if it exceeds 1.0).

2. Use `lerp()` to move a shape between two points of your choosing in your `draw()` loop. Use the global time you set up in step 2 to do the interpolation.

3. Do the same thing as in step 3, but use `lerpColor()` to make the shape change color smoothly as well.

4. OPTIONAL: Instead of making the counter roll over from 1.0 to 0.0, make it smoothly oscillate between 0.0 an 1.0, back and forth. There are several ways to do this, but one is to use `0.5 * (sin(time)+ 1)` as your lerp argument (since it is always in the range [0.0, 1.0])

# Index Cards!

1. Your name and EID.

2. One thing that you learned from class today. You are allowed to say "nothing" if you didn't learn anything.

3. One question you have about something covered in class today. You *may not* respond "nothing".

4. (Optional) Any other comments/questions/thoughts about today's class.