Tips for Creating Animations with p5.js

Learn functions and techniques that will help you take your p5.js animations to the next level!

At this point in your p5.js learning journey, you know how to create a canvas, draw and color shapes, and animate groupings of those shapes. Here are some tips and functions that will help you take your p5.js animations to the next level.

Relative Position and Size

If you are creating full-size browser animations, you might want to use relative positions and sizes for your shapes to adjust the animation for varying screen sizes. You previously learned how to use the `width` and `height` variables to dynamically size shapes in the Drawing and Coloring Shapes lesson. Using a combination of functions with built-in variables such as `width`, `height`, `windowWidth`, and `windowHeight` will allow you to dynamically position and size elements in your visualizations.

To create a canvas that matches the dimensions of your browser, use the `createCanvas()` function in the `setup()` function, passing the `windowWidth` and `windowHeight` variables as the width and height values. The `windowWidth` and `windowHeight` variables store the browser windows' current width and height.

```
function setup(){
  createCanvas(windowWidth, windowHeight);
}
```

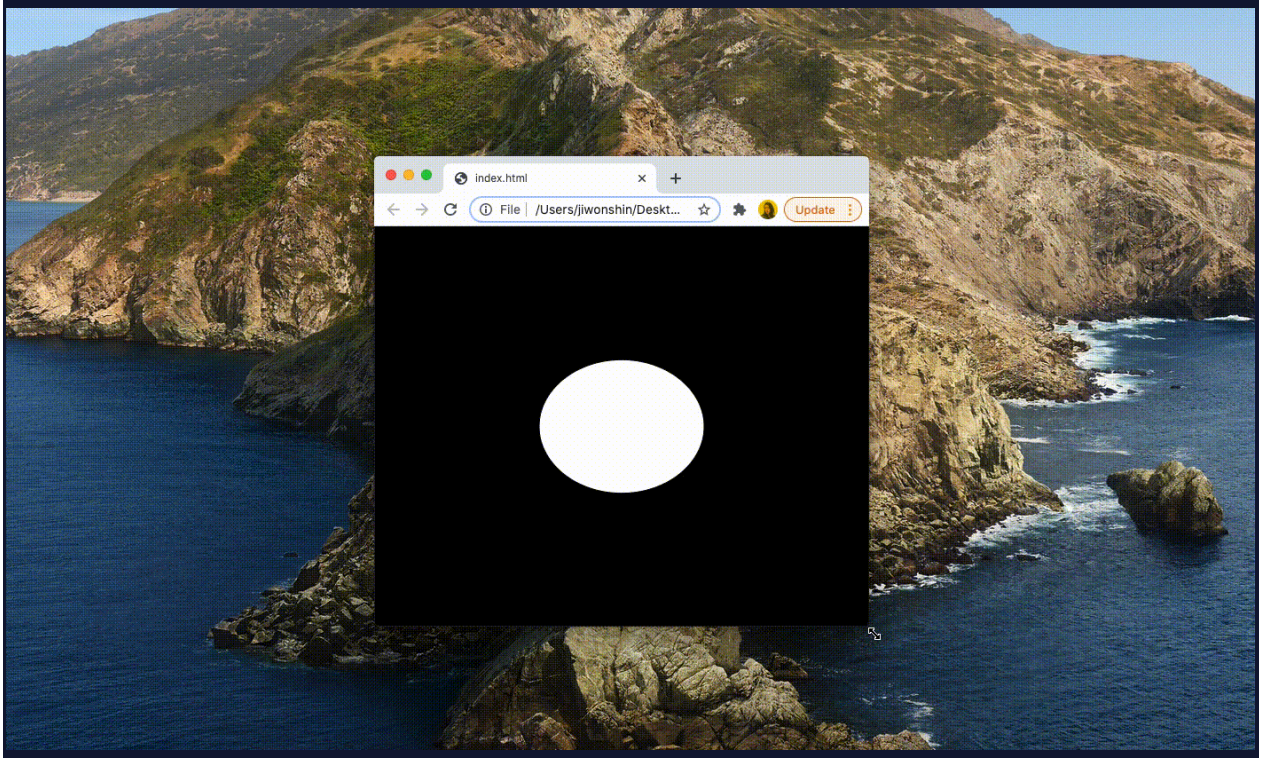
However, if you change the size of your browser window, you'll notice that the canvas size doesn't adjust to the new browser size! If you want your canvas to dynamically adjust to changes in the browser window's size, you can use the `resizeCanvas()` function in the `windowResized()` function:

```
function windowResized() {
  resizeCanvas(windowWidth, windowHeight);
}
```

Whenever p5.js detects a change in browser size, the code block inside the `windowResized()` function is triggered. This process is similar to how mouse and keyboard events are detected, which is covered in the Interaction lesson. The `resizeCanvas()` function takes the updated `windowWidth` and `windowHeight` variables as arguments to resize the canvas. The function also updates the values of the

canvas' `width` and `height` values, resulting in the resizing and repositioning of all elements that use the `width` and `height` variables in their arguments.

In this gif, you can see an example sketch where the sketch and its contents dynamically resize as the width and height of the browser changes.



Here is the code for the p5.js sketch running in the gif:

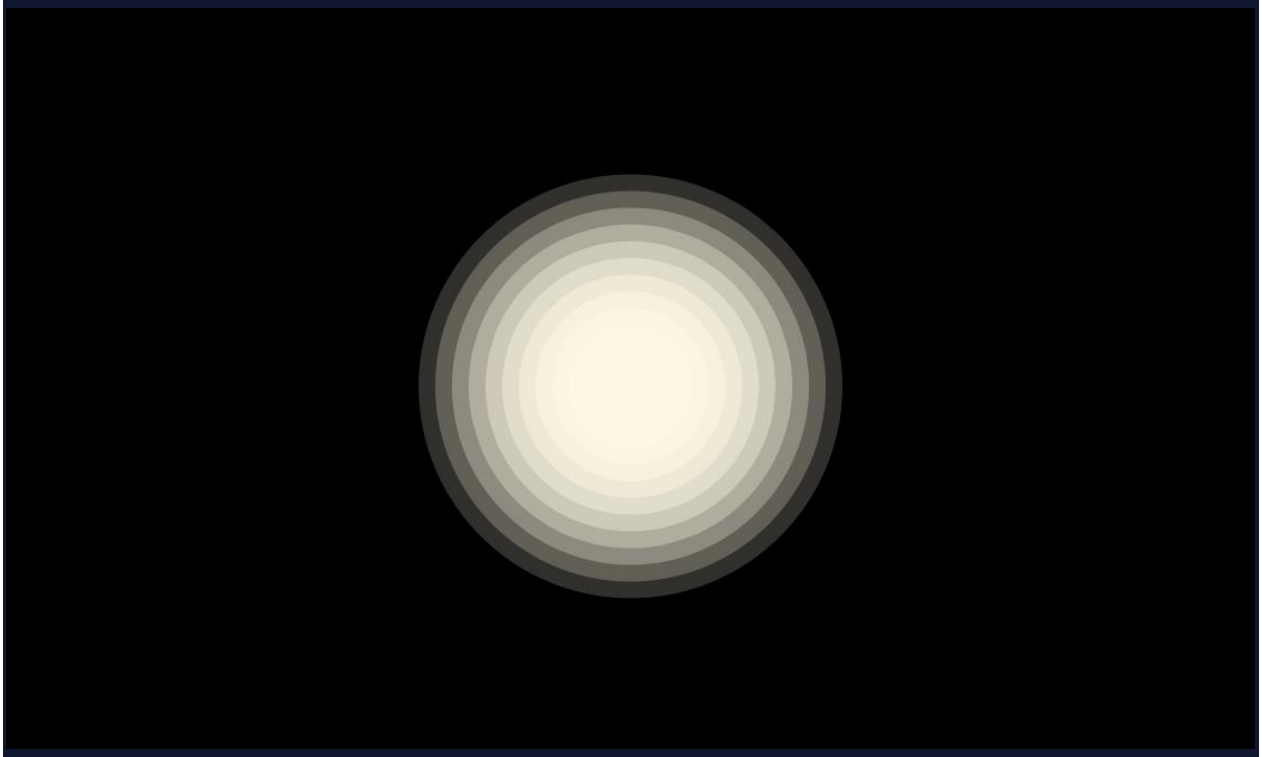
```
function setup() {
  // Canvas created to the width and height of browser window
  createCanvas(windowWidth, windowHeight);
}

function draw() {
  background(0);
  // Ellipse drawn in the middle of the canvas
  ellipse(width / 2, height / 2, width / 3, height / 3);
}

function windowResized() {
  // Whenever browser size changes, canvas is resized to browser's current width and height
  resizeCanvas(windowWidth, windowHeight);
}
```

In this code snippet, the `ellipse()` function uses the `width` and `height` variables to control its position and size. The `windowResized()` function controls the relative size of both the canvas and the elements in the sketch, such as the ellipse.

Opacity

Opacity is key to creating smooth and fluid visualizations. The Drawing and Coloring Shapes lesson covers how to use opacity to color shapes, enabling

them to be more or less transparent. Remember that you can specify the opacity (alpha) value with coloring functions such as `fill()` and `stroke()`.



The image above depicts a series of light yellow ellipses placed on top of each other. You can create this spotlight-like visualization using varying levels of opacity and layering. Here is the code for the p5.js sketch above:

```
function setup() {
  createCanvas(windowWidth, windowHeight);
}

function draw() {
  background(0);
  noStroke();

  for (let i = 0; i < 10; i++) {
    // alpha value increases by 10 with every iteration
    let alpha = 50 + i * 10;
    // size value decreases by 30 with every iteration
    let size = width / 3 - i * 30;

    push();
    fill(255, 250, 229, alpha);
    ellipse(width / 2, height / 2, size, size);
    pop();
  }
}
```

In the code above, the program uses a `for` loop to create ten ellipses that decrease in size and increase in opacity with each iteration. `for` loops are convenient when you want to create copies of the same shape, with the potential to modify the shape with each iteration. You can use the iterator variable, in this case, `i`, to create variances between each shape drawn. In this

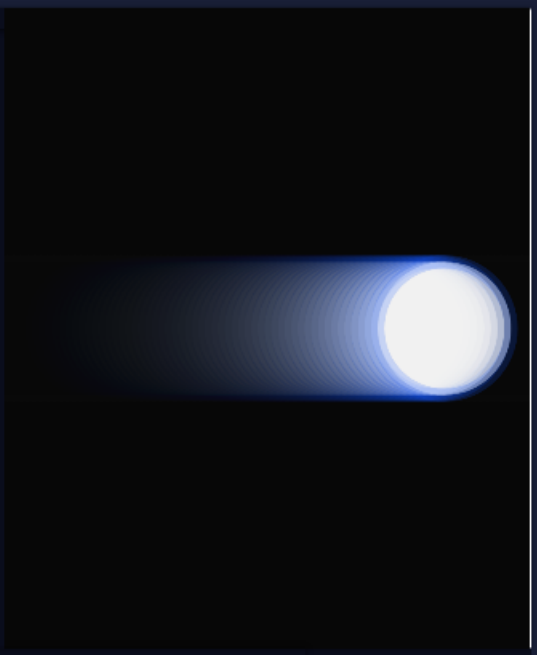
case, `i` is used to augment the alpha value used to control and change the ellipses' opacity value.

The background color can also have an opacity value—enabling you to create special effects! When the background color has opacity, instead of the new frame replacing the previous frame, the frames drawn to the canvas leave a faint trace, creating a trailing effect for any moving elements!

Coding question

In the sketch below, the x position of the ellipse is incremented by 5 pixels every frame and reset when the x position value becomes greater than the canvas's width. You can see that the background uses an opacity value and therefore causes the ellipse to leave traces. Play around with the opacity value of the background, as well as the fill and stroke color of the ellipse, and see what happens! You can also alter the increment of the `xpos` variable to see the changes in the trailing effect.

```
function draw() {
  // Background is black with opacity of 15
  // Opacity ranges between 0 and 255
  background(0, 15);

  // Fill color is white with opacity of 127
  fill(255, 127);
  strokeWeight(10);
  // Stroke color is blue with opacity of 50
  stroke(21, 87, 255, 50);
  ellipse(xpos, height / 2, 100, 100);

  // Try changing the increment value below
  xpos += 5;

```



Run

Check answer

Run your code to check your answer

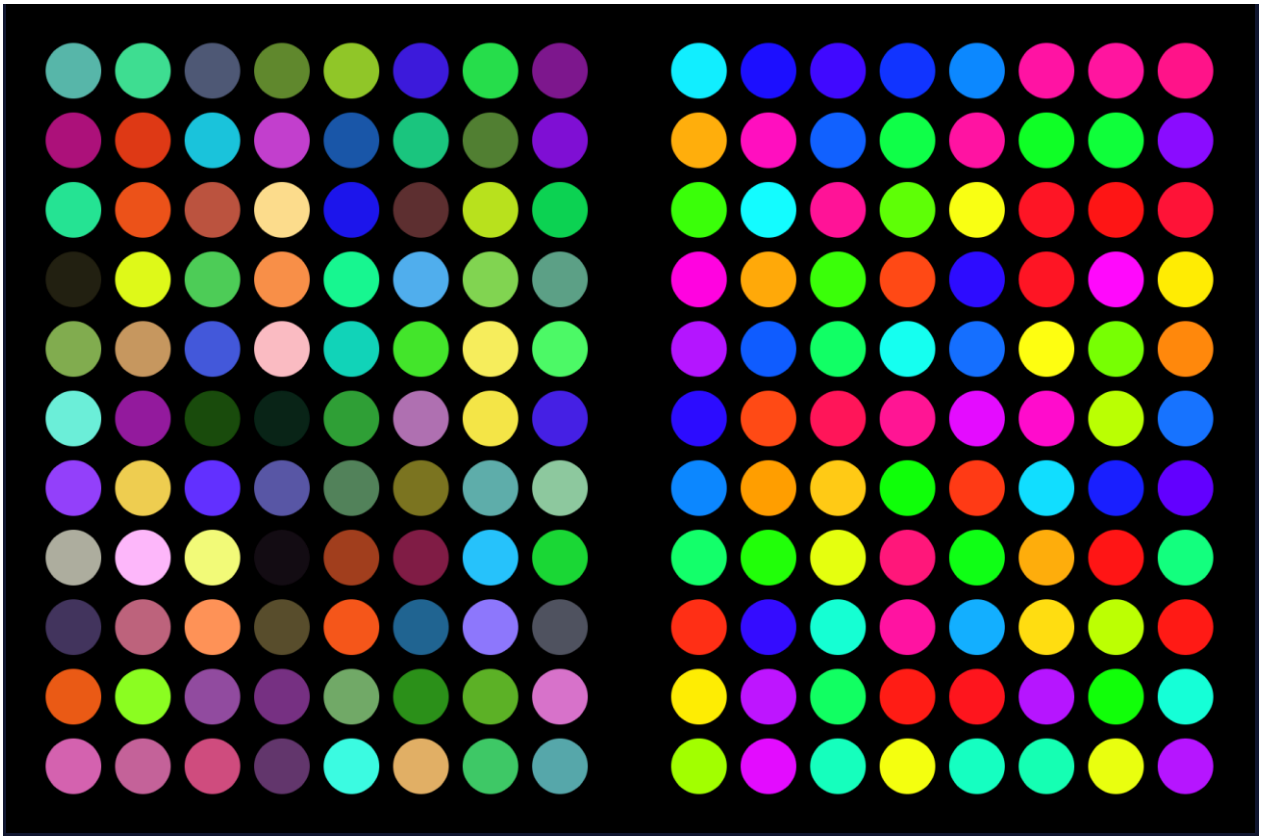
```
let xpos = 0;

function setup() {
  createCanvas(windowWidth, windowHeight);
}
```

```
function draw() {
  // Background is black with opacity of 15
  // Opacity ranges between 0 and 255
  background(0, 15);

  // Fill color is white with opacity of 127
  fill(255, 127);
  strokeWeight(10);
  // Stroke color is blue with opacity of 50
  stroke(21, 87, 255, 50);
  ellipse(xpos, height / 2, 100, 100);

  // Try changing the increment value below
  xpos += 5;

  if(xpos > width){
    xpos = 0;
  }
}

function windowResized() {
  resizeCanvas(windowWidth, windowHeight);
}
```

Color Mode

There are various ways to represent color with p5.js, such as RGB values, named colors values, and hex values. Another way to specify color is with HSB (Hue, Saturation, Brightness) values. There are times when representing the color in HSB values is more useful than using the RGB notation. Using HSB values allows you to randomize hue values while keeping the saturation and brightness values constant.

The `colorMode()` built-in function allows us to change the way color values are interpreted within our sketch. The function requires one argument to specify the color mode:`RGB` or `HSB`.



The image above shows two sets of ellipses, each filled with randomly generated colors. Ellipses on the left-side are filled with colors with random red, green, and blue values, and those on the right are filled with randomly generated hues. Notice that the ellipses on the right-side have a variety of hues while maintaining saturation and brightness levels.

```
// Color mode is set to RGB
colorMode(RGB);
// Randomly generate r, g, and b values between 0 - 255
r = random(255);
g = random(255);
b = random(255);
fill(r, g, b);

// Color mode is set to HSB
colorMode(HSB);
// Randomly generate hue balue between 0 - 360
hue = random(360);
// When color mode is set to HSB, maximum value of saturation and brightness is 100
fill(hue, 100, 100);
```

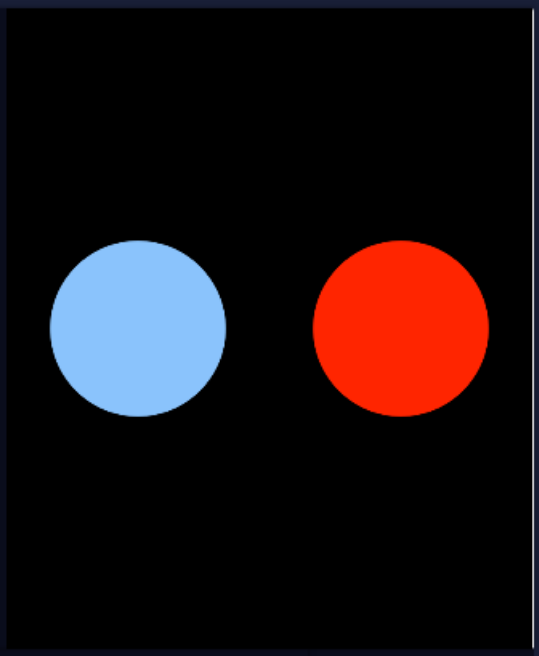
The code example above shows a segment of the sketch that generated the image above with the two sets of ellipses. See how the `colorMode()` function is used to set the color mode, first as `RGB` and then as `HSB`. When the color mode is set to HSB, you only need to randomize one value to get various shades of colors.

Coding question

Coding question

In the sketch below, the colors of the two ellipses are randomly generated once every second. The left ellipse's color mode is set to `RGB` and its color is generated by assigning the `redValue`, `blueValue`, and `greenValue` variables to random numbers between 0 and 255. The right ellipse's color mode is set to `HSB` and its color is generated by assigning the `hueValue` variable to a random number between 0 and 360. Notice the difference in the range of randomly generated colors. Can you alter the arguments of the `random()` functions to generate colors in a specific hue range? Is it easier to do this when the color mode is set to `RGB` or `HSB`?

```
let size;

function setup() {
  createCanvas(windowWidth, windowHeight);

  noStroke();
  // Set width and height of ellipses to
  be width / 3
  size = width / 3;

  // Set frame rate to 1 so colors are
  only generated once per second
  frameRate(1);
}

function draw() {
  background(0);

  // Color mode for the left ellipse is
```



Run

Check answer

```
let size;

function setup() {
  createCanvas(windowWidth, windowHeight);

  noStroke();
  // Set width and height of ellipses to be width / 3
  size = width / 3;

  // Set frame rate to 1 so colors are only generated once per second
  frameRate(1);
}
```

```
function draw() {
  background(0);

  // Color mode for the left ellipse is set to RGB
  colorMode(RGB);
  // Generate a random number between 0 and 255 for each red, green, and blue values
  let redValue = random(255);
  let blueValue = random(255);
  let greenValue = random(255);

  fill(redValue, blueValue, greenValue);
  ellipse(width / 4, height / 2, size, size);

  // Color mode for the right ellipse is set to HSB
  colorMode(HSB);
  // Generate a random number between 0 and 360 for the hue value
  let hueValue = random(360);

  fill(hueValue, 100, 100);
  ellipse(width / 4 * 3, height / 2, size, size);
}

function windowResized() {
  resizeCanvas(windowWidth, windowHeight);
}
```

map()

Using variables is a great way to create dynamic animations with p5.js. A single variable will often affect multiple visual properties in a sketch, such as the position and the color of a shape.

```
colorMode(HSB);
// x_position is used to animate hue value
fill(x_position, 100, 100);
// x_position is used to animate x position of the ellipse
ellipse(x_position, height / 2, 100, 100);

// Increment x_position by 1
x_position++;

if(x_position > width){
  // Reset x_position when it becomes greater than width
  x_position = 0;
}
```

In the code example above, we have an ellipse that moves from the left to the right of the canvas. When the value of the `x_position` variable is greater than the `width` of the canvas, the position of the ellipse resets to the left of the canvas. The `x_position` variable is also used as the hue value of the fill color, so the color of the ellipse will change as it moves across the canvas.

Remember that the maximum value of the hue value is 360. What happens if the width of our canvas is larger than 360, meaning that the value of the `x_position` variable that determines the hue will be greater than 360?

.

.

.

.

.

Answer: the ellipse will remain red until the `x_position` value becomes smaller than 360 again.

This example is a situation where the `map()` function comes in handy. The `map()` function translates a number from one range to another range.

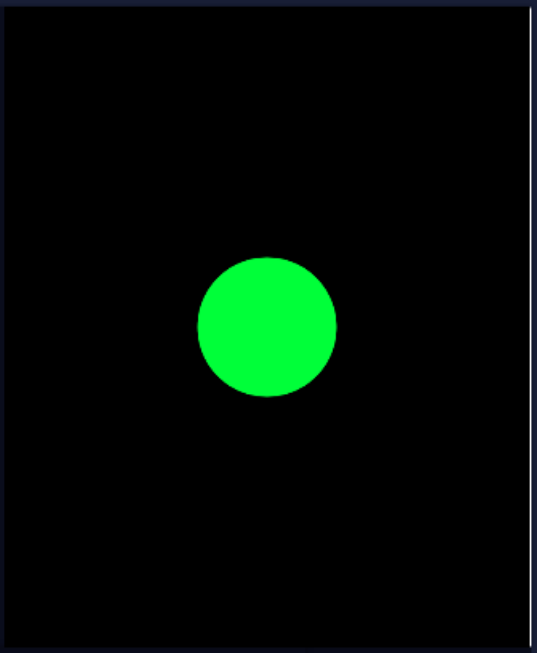
```
let hue_value = map(x_position, 0, width, 0, 360);
```

Take a look at how you can use the `map()` function to solve the previous problem. The function takes in the variable to be mapped as its first argument. The original range of the `x_position` value is given as the second and third arguments. The final two arguments are the minimum and maximum values of the new range. You can now use the `hue_value` variable to animate the full range of hues as the ellipse travels from the left to the right of the canvas.

Coding question

In the code example below, there is an ellipse that changes in color as the horizontal position of the mouse changes. You can learn more about using mouse events in the Interaction lesson. Play around with the different arguments with the `map()` function. Can you limit the range of the hue value even further? For example, can you try to map the `mouseX` position to shades of blue?

```
function setup() {
  createCanvas(windowWidth, windowHeight);
  // Set color mode to HSB
  // Hue ranges between 0 - 360
  // Saturation ranges between 0 - 100
  // Brightness ranges between 0 - 100
  colorMode(HSB, 360, 100, 100);
}

function draw() {
  background(0, 15);

  // Map mouseX with initial range of 0 and width to a new range of 0 and 360
  let hue = map(mouseX, 0, width, 0, 360);
  // Map mouseX with initial range of 0 and width to a new range of 50 and width / 2
  let size = map(mouseX, 0, width, 50,
```



Run

Check answer

Run your code to check your answer

```
function setup() {
  createCanvas(windowWidth, windowHeight);
  // Set color mode to HSB
  // Hue ranges between 0 - 360
  // Saturation ranges between 0 - 100
  // Brightness ranges between 0 - 100
  colorMode(HSB, 360, 100, 100);
}

function draw() {
  background(0, 15);

  // Map mouseX with initial range of 0 and width to a new range of 0 and 360
```

```
  let hue = map(mouseX, 0, width, 0, 360);
  // Map mouseX with initial range of 0 and width to a new range of 50 and width / 2
  let size = map(mouseX, 0, width, 50, width / 2);

  noStroke();
  fill(hue, 100, 100);
  ellipse(width / 2, height / 2, size, size);
}

function windowResized() {
  resizeCanvas(windowWidth, windowHeight);
}
```

Lerp

When updating the positions of shapes by incrementing and decrementing values, you get a very linear motion. The `lerp()` function is what you can use to create a more smooth and natural animation! *lerp* is a programming term that is short for linear interpolation, a way of figuring out points that lie between two endpoints of a line. In p5.js, the `lerp()` function calculates a number between two values at a specific percentage increment.

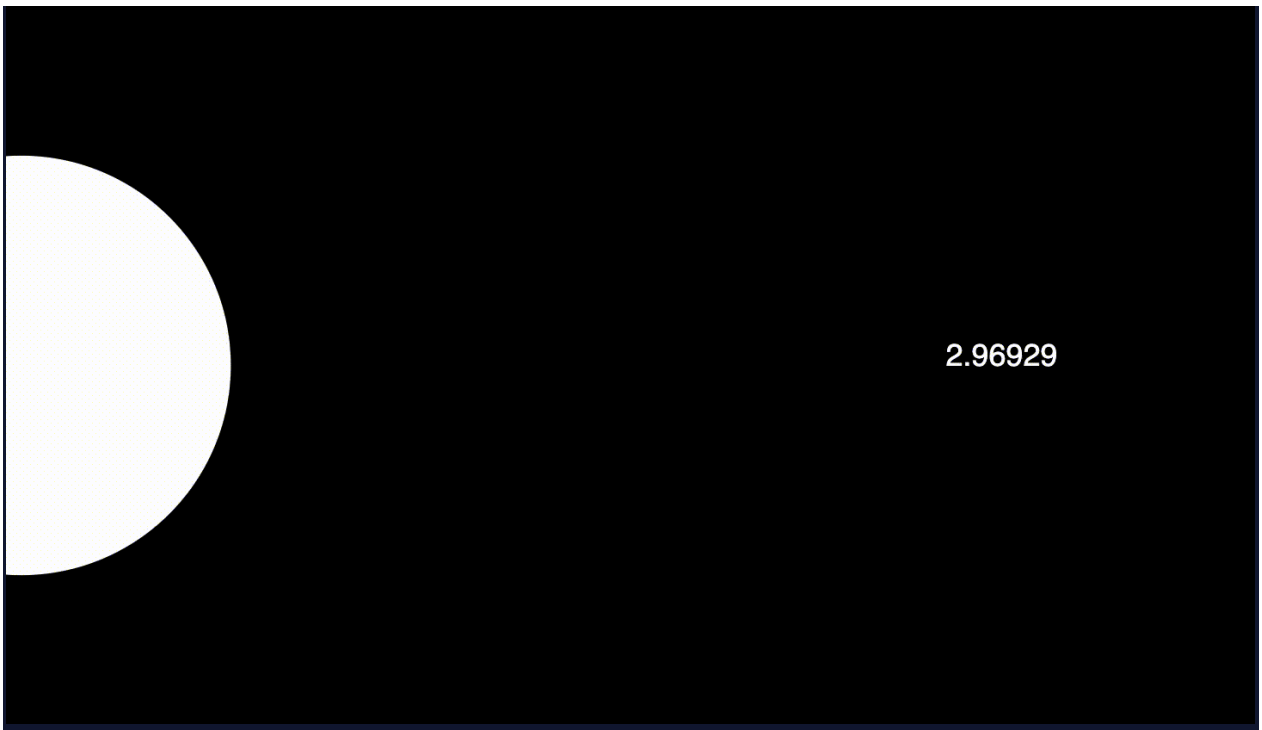
```
let value = lerp(20, 250, 0.2); // value is 66
```

In the code example above, the `lerp()` function is used to calculate a number that is 20 percent (0.2) from 20 towards 250. The value of the variable `value` is equal to 66.

The `lerp()` function shines when it is used to update the start value of the function.

```
let x_position = 0;
let end_position = width;

// Calculate value that is 2 percent (0.02) from x_position towards end_position
x_position = lerp(x_position, end_position, 0.02);
```

Here, the `x_position` variable is updated with a 0.02 decimal percentage increment (2%) towards the `end_position` value. When this code runs in the `draw()` loop, we see an ease-in and ease-out motion because the 2% increment value gradually becomes smaller as the distance between `x_position` and `end_position` is reduced. Take a look at the GIF below. The number next to the moving ellipse shows the lerp value between 0 and 100. Notice how the number changes faster between 10 and 90 and slows down when approaching 0 and 100.



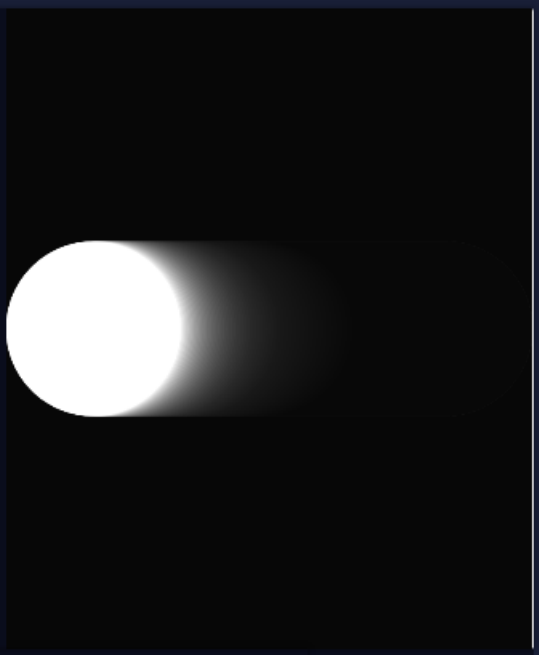
Coding question

Take a look at the sketch below. Play around with the increment value (the third argument of the `lerp()` function). What happens when the increment value is smaller or bigger? What values do you have to change to set the overall range of horizontal movement of the ellipse?

Coding question

Take a look at the sketch below. Play around with the increment value (the third argument of the `lerp()` function). What happens when the increment value is smaller or bigger? What values do you have to change to set the overall range of horizontal movement of the ellipse?

```
let pos, target;

function setup() {
  createCanvas(windowWidth, windowHeight);

  // Initialize pos and target variables
  // pos holds the current position of
  the ellipse
  pos = width / 6;
  // target holds the end position of the
  ellipse
  target = width;
}

function draw() {
  background(0, 15);
  noStroke();
  fill(255);
  // Calculate lerp value from pos to
```



Run

Check answer

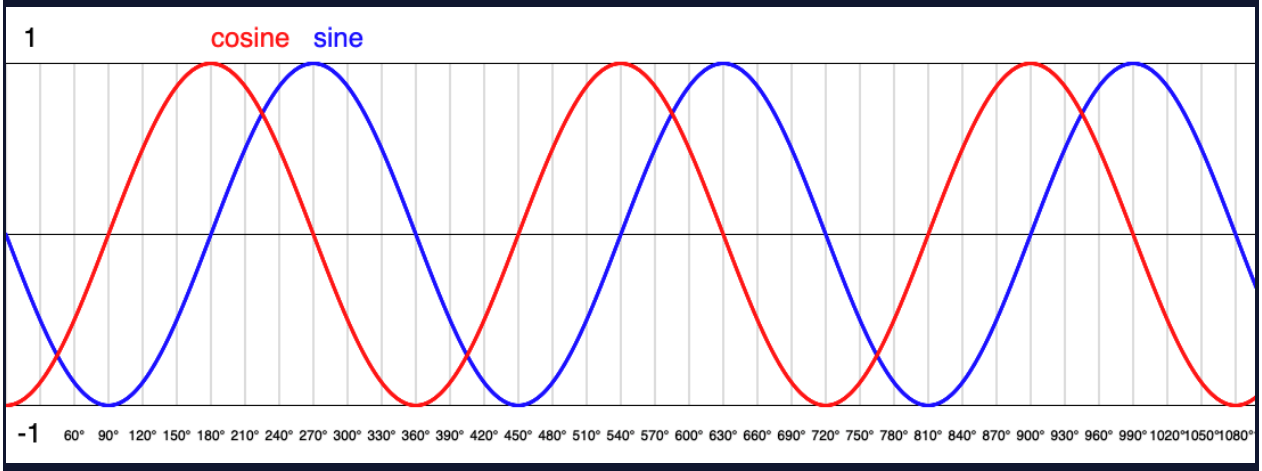
Run your code to check your answer

```
let pos, target;

function setup() {
  createCanvas(windowWidth, windowHeight);

  // Initialize pos and target variables
  // pos holds the current position of the ellipse
  pos = width / 6;
  // target holds the end position of the ellipse
  target = width;
}

function draw() {
  background(0, 15);
  noStroke();
```

```
  fill(255);
  // Calculate lerp value from pos to target
  // Increases 2 percent every loop
  pos = lerp(pos, target, 0.02);
  ellipse(pos, height / 2, width / 3, width / 3);

  // Set target back to 0 when the ellipse reaches the right side of the canvas
  if(pos > width - width / 6){
    target = 0;
  }

  // Set target back to width when the ellipse reaches the left side of the canvas
  if(pos < width / 6){
    target = width;
  }
}

function windowResized() {
  resizeCanvas(windowWidth, windowHeight);
}
```

Oscillation with Sine and Cosine

Remember the sine and cosine curves you learned in school? Think back about the sine and cosine graphs—they look like waves fluctuating between -1 and 1.



When creating animations in p5.js, using the `sin()` and `cos()` functions will help you create naturally oscillating movements like the movement of a spring or a swing. The `sin()` and `cos()` functions calculate the sine and cosine value of an angle, given as the argument of the functions. There is not much difference between

the `sin()` and `cos()` values other than the starting point. At `sin(0)`, the value is 0 and at `cos(0)` the value is 1.

```
let size = sin(frameCount * 0.01) * width / 2 + width / 2;
```

In the code example above, the `sin()` function has the argument of `frameCount * 0.01`. The `frameCount` variable is used to increment the angle constantly, and the incrementation is slowed down by multiplying the frame count by the value 0.01. (Remember that in p5.js, angle values are given in radians!) Because sine values range between 1 and -1, when you multiply the value by `width / 2` you get numbers ranging between `-width / 2` and `width / 2`. And by adding on `width / 2`, the final range of the `size` variable becomes 0 and width!

Coding question

It takes a little practice to get used to using the `sin()` and `cos()` functions. Take a look at the sketch below. Try changing the number the `frameCount` variable is multiplied by. Can you make the oscillation slower or faster? Play around with the range of motion by changing the `width / 5` value to something else! One piece of advice—change one value at a time to see what the visual difference is.

```
function setup() {
  createCanvas(windowWidth, windowHeight);
}

function draw() {
  // Background color is black with
  opacity of 15
  background(0, 15);
  noStroke();
  fill(255, 127);
  // Oscillate size using sin()
  // sizeOne ranges between 0 and width /
  2
  let sizeOne = sin(frameCount * 0.01) *
  width / 4 + width / 4;
  ellipse(width / 4, height / 2, sizeOne,
  sizeOne);

  // Oscillate size using cos()
```

Run

Check answer

Run your code to check your answer

```
function setup() {
  createCanvas(windowWidth, windowHeight);
}
```

```
function draw() {
  // Background color is black with opacity of 15
  background(0, 15);
  noStroke();
  fill(255, 127);
  // Oscillate size using sin()
  // sizeOne ranges between 0 and width / 2
  let sizeOne = sin(frameCount * 0.01) * width / 4 + width / 4;
  ellipse(width / 4, height / 2, sizeOne, sizeOne);

  // Oscillate size using cos()
  // sizeTwo ranges between 0 and width / 2
  let sizeTwo = cos(frameCount * 0.01) * width / 4 + width / 4;
  ellipse(width / 4 * 3, height / 2, sizeTwo, sizeTwo);
}

function windowResized() {
  resizeCanvas(windowWidth, windowHeight);
}
```

Practice, Experiment, Play!

The functions and techniques introduced in this article showed you new ways you can create animations with p5.js. You now know how to use different built-in functions such as `resizeCanvas()`, `colorMode()`, `map()`, `lerp()`, `sin()` and `cos()`. Using these techniques, on their own or in isolation, will level up your creations. But the most important thing to know is that it takes a little practice, a bit of experimentation, and a playful spirit to have fun and generate interesting and exciting visualizations!