**Advanced Coding**
**Tuesday Topics**

## Coding Example – Estimating the value of Pi!

What is Pi? Where does it come from? We can simulate a physics-style experiment to estimate the value of pi!

Imagine a circle or radius r inscribed inside a rectangle.



From our knowledge of geometry we know that the area of the circle is Πr2. The area of the square is 2r\*2r or 4r2. If we take the ratio of the areas we get:



The r2’s cancel out leaving us with the ratio equaling



This means if we can figure out the ratio of the areas, and multiply it by 4, then we would get the value of pi!

How can we get the ratio of the areas? One way is to imagine the circle and square is on a picture hung up on the wall and we start throwing a bunch of darts at the picture. If we count up how many darts are in the circle and divide it by how many darts are in the square then we get an approximation of the ratio!

We can simulate this in a program where we use a loop that picks random coordinates for the darts. Starting with a new program, we can delete draw because we’ll just do everything from setup.

A good starting point is to just draw a square and a circle in the square. I made a radius variable set to 250 that is global because I’ll access later in a separate function.

let radius = 250;

function setup() {

 createCanvas(windowWidth, windowHeight);

 background(100);

 // Black stroke and white inside

 fill(255,255,255);

 // Square and circle

 rect(0,0,radius\*2,radius\*2);

 circle(radius,radius,radius\*2);

}



The center of the circle is at coordinate radius,radius. This makes the square’s side radius\*2. The upper left corner is at 0,0.

Next to pick a random coordinate for our dart we can use the random function. This code sets x and y to random numbers between 0 and 2\*radius (exclusive).

 let x = random(2\*radius);

 let y = random(2\*radius);

Here is a function that will return true if a point x,y is inside the circle. It uses the dist function:

function isInCircle(x,y)

{

 let distance = dist(x,y,radius,radius);

 if (distance < radius)

 {

 return true;

 }

 else

 {

 return false;

 }

}

We can tie these together to draw a red dot for the “dart” inside the circle and a green dot for the “dart” outside the circle!

 // Pick a random x,y point between 0 and 2\*r

 let x = random(2\*radius);

 let y = random(2\*radius);

 if (isInCircle(x,y))

 {

 fill(255,0,0); // Draw a red dot

 }

 else

 {

 fill(0,255,0); // Draw a green dot

 }

 circle(x,y,4);

This code is on the Open Processing classroom site as Pi Estimator Starter Sketch.

## Coding Challenge!

Starting with the given code, add a loop to simulate many darts (hundreds? Thousands?), count how many land in the square, how many in the circle, and output the estimate of Pi. How close to the actual value of Pi can you get?

Share your sketch in the “Tuesday submissions” folder.

## Transformations

Let’s do a short introduction to transformations which allow us to do some cool and fancy operations with regard to drawing. Transformations are really important in 3D graphics, which we won’t get into, but this will give you an idea of what you are in for if you decide to get into 3D graphics later!

Transformations basically allow you to change the origin and orientation of items drawn on the canvas. Here we’ll look at these three functions: **translate**, **rotate**, and **scale**.

**Translate** allows us to change the origin from whence items are drawn.

Here is code that draws a square whose upper left corner is at 100,100:

function setup() {

 createCanvas(windowWidth, windowHeight);

}

Function draw() {

 background(100);

 fill(255,255,255);

 rect(100,100,200,200);

}



We could do the same thing by translating the origin to 100,100. Then if we draw the rectangle with an upper left corner at 0,0 we get the same thing!

 translate(100,100);

 rect(0,0,200,200);

Why might you want to do this instead of the version using absolute coordinates? One reason is that translate accumulates. If we do another one it will be relative to the previous origin:

 translate(100,100);

 rect(0,0,200,200);

 translate(10,10);

 fill(255,0,0);

 rect(0,0,10,10);



This can sometimes be useful whenever we want to change the origin. Try changing the translate to mouseX and mouseY! Note that transformations do reset back to 0,0 each time we call draw, so you’ll have to restart your transformations for each function call.

**Rotate** is really useful if you want something to rotate. Rotation is not as simple as just saying to rotate something. If you want something to rotate then you have to tell the computer what it needs to rotate around. What is the center of rotation? It will be the center of origin, as specified by translate!

We can experiment by rotating our square.

First, I added this line to setup so we can rotate in degrees instead of radians:

 angleMode(DEGREES);

I made a global variable to hold the angle of rotation. It is global so it will keep its value for each time draw is run. Here is the code:

let angle = 0;

function setup() {

 createCanvas(windowWidth, windowHeight);

 angleMode(DEGREES);

}

function draw() {

 background(100);

 fill(255,255,255);

 rotate(angle);

 rect(0,0,200,200);

 angle = angle + 1;

}

This rotates around the upper left corner, because that is the default origin!

If we translate so the origin is in the center of the square then it will rotate about its axis!

function draw() {

 translate(100,100);

 background(100);

 fill(255,255,255);

 rotate(angle);

 rect(0,0,200,200);

 angle = angle + 1;

}



This rotates around 100,100 but it draws the upper left corner of the square at 100,100 because we translated it there! If we want the square to be drawn at global 0,0 then we can draw it at x=-100,y=-100 to compensate for the translation:

function draw() {

 translate(100,100);

 background(100);

 fill(255,255,255);

 rotate(angle);

 rect(-100,-100,200,200);

 angle = angle + 1;

}



Another way to do the same thing is to use rectMode and tell it to reference around the center:

function draw() {

 translate(100,100);

 rectMode(CENTER);

 background(100);

 fill(255,255,255);

 rotate(angle);

 rect(0,0,200,200);

 angle = angle + 1;

}

Here is code that makes a rotating square in the center, and a small circle that rotates around the square in the opposite direction. This is in the sketch Square Circle Rotation in the classroom website.

let squareAngle = 0;

let circleAngle = 0;

function setup() {

 createCanvas(windowWidth, windowHeight);

 angleMode(DEGREES);

}

function draw() {

 background(0);

 // Draw and rotate square

 translate(400,400);

 rectMode(CENTER);

 fill(255,0,0);

 rotate(squareAngle);

 rect(0,0,200,200);

 squareAngle = squareAngle + 1;

 // Draw and rotate circle

 fill(0,0,255);

 rotate(circleAngle);

 ellipse(250,0,10,10);

 circleAngle = circleAngle - 5;

}

The **scale** function changes the size of an object. It is a multiplicative effect so scale(2) will make everything twice as big. Try adding the following right in from of the ellipse line:

scale(squareAngle / 200);

Finally, if you want to save the origin/scale/rotation/color then you can use **push()** and **pop()** will reset the original/scale/rotation/color to the values when they were pushed. For example, consider this in draw:

function draw() {

 background(255);

 fill(255,0,0);

 translate(200,200); // Draw red circle centered at 200,200

 ellipse(0,0,100,100);

 fill(255,255,0);

 translate(200,200); // Move origin 200 over and 200 down

 scale(0.5);

 ellipse(0,0,100,100); // Yellow circle half the size

}

The output is:



Now consider if we push right after setting the color to red, then pop right before drawing the second ellipse. This will reset the origin back to 0,0 and set the color to red:

function draw() {

 background(255);

 fill(255,0,0);

 **push(); // Save color, rotation, origin, scale**

 translate(200,200); // Draw red circle centered at 200,200

 ellipse(0,0,100,100);

 fill(255,255,0);

 translate(200,200); // Move origin 200 over and 200 down

 scale(0.5);

 **pop(); // Restore origin to 0,0, color = red, scale = 1**

 ellipse(0,0,100,100); // Not circle half the size

}



## Coding Challenge!

Make a miniature solar system simulation with a sun and two planets revolving around the sun. Then make a moon revolve around one of the planets. In our solar system simulation everything revolves in a circle, not in ellipses 😊

Share your sketch in the “Tuesday submissions” folder.

## Bonus: Kanizsa Illusion

This illusion was first discovered in 1955 by Gaetano Kanizsa , an Italian artist and psychologist. It is formed using pac man shapes! Here is a Kanizsa triangle:



You should see a white triangle, even lighter than the background, pointing upward with corners that correspond with the pac man mouths. This triangle is on top of a downward pointing triangle. Of course, there is no lighter triangle there in the middle, it is an optical illusion!

Here is some code that draws a Kinizsa Square where the rotation is controlled by the mouse X coordinate:

function pacMan(x,y,angle)

{

 push();

 fill(0,0,0);

 translate(x,y);

 rectMode(CENTER);

 arc(x,y,100,100,45+angle,-45+angle);

 pop();

}

function setup() {

 createCanvas(windowWidth, windowHeight);

 angleMode(DEGREES);

 noStroke();

}

function draw() {

 background(255);

 pacMan(200,200,mouseX);

 pacMan(275,200,mouseX+90);

 pacMan(200,275,mouseX-90);

 pacMan(275,275,mouseX-180);

}



## Coding Challenge!

Make an animated Kinizsa Triangle. You will have to change the angle of the mouth so it’s 60 degrees instead of 90 degrees.

Alternately, make a sequence of 8 pac men that animate as in this video (about 1:34 in):

<https://www.youtube.com/watch?time_continue=117&v=2ODdj9fhCMI>

Optional: Have the circles rotate automatically instead of controlled by the mouse.

Share your sketch in the “Tuesday submissions” folder.

## Using the p5.play Library

There is a library called p5.play that provides a way to more easily handle sprites (images) and animation if we’re going to write a 2D game or game-like application.

To enable the library in an Open Processing sketch, click on the three dots in the upper right and under Settings, click on “Show All” for Libraries. Toggle on the p5play library and p5.collide2d.



To see a reference for the p5.play library, go to <https://p5play.org/>. There are a lot of cool demos, documentation, and sample code on the website along with tutorials! Also check out the creator’s open processing sketches: <https://openprocessing.org/user/350295?view=sketches&o=36>

**Sprites**

To start with, we need some images to use to animate. A **sprite** is just a picture of a game element, like a character, monster, bullet, etc. We can animate a sprite by switching back and forth between different images, just like with a flip-book style animation. To start, we can create a sprite that is a simple shape.

p5play works best inside its own canvas, as opposed to using the standard createCanvas we have been using so far. In general, p5play has its own ways to access the mouse and keyboard too. We initialize our sprites in the setup() function. The default code for a p5play project looks like this:

function setup() {

 //createCanvas(windowWidth, windowHeight);

 **new Canvas(windowWidth, windowHeight);**

 // Change to specific values if desired or leave empty

 // to make it the size of the entire browser window

 background(100);

}

function draw() {

 **clear();**

}

Note that we just have a clear() in the draw function! The p5play library will draw the sprites for us so we don’t have to draw them ourselves. We just specify where they are and what they look like on the screen.

Here we create a sprite using the keyword “new”. If we don’t specify where it goes then it will be drawn in the middle of the screen as a box:

let sprite;

function setup() {

 new Canvas();

 background(100);

 sprite = new Sprite();

 sprite.width = 20;

 sprite.height = 30;

}

function draw() {

 clear();

}

Here is the result:



Here we set the width and height of the box. Notice that we didn’t specify anything about drawing the box, it draws itself! Some other properties we can set:

|  |  |
| --- | --- |
| .diameter | Diameter of a circle sprite |
| .radius | Radius of a circle sprite |
| .x | X position of center |
| .y | Y position of center |
| .position  | Both X and Y as {x: val, y: val} |
| .color | Color of sprite fill; can use text, e.g. “blue” |
| .stroke | Color of sprite border |
| .text | Text in the sprite, use “message” |
| .textColor | Color of text |
| .textSize | Font size |
| .visible | true or false to make visible or not |
| .rotation | Degrees to rotate |

We can also assign a velocity to sprites!

|  |  |
| --- | --- |
| .vel.x | X velocity, positive is to the right |
| .vel.y | Y velocity, positive is to the bottom |

This code will make the sprite move slowly down toward the right:

 sprite.vel.x = 0.2;

 sprite.vel.y = 0.2;

Here is an example that makes a new sprite that moves in a random direction at the location of the mouse. Pressing the mouse button creates a circle and pressing the “up” key creates a box sprite. Note the new, p5play way, to check if the mouse is pressed or if a key is pressed.

function setup() {

 new Canvas();

 background(255);

}

function draw() {

 clear();

 if (mouse.presses()) {

 let circle = new Sprite();

 circle.diameter = 30;

 circle.x = mouseX;

 circle.y = mouseY;

 circle.vel.x = random(-3,3);

 circle.vel.y = random(-3,3);

 }

 if (kb.presses('up')) {

 let box = new Sprite();

 box.width = 30;

 box.x = mouseX;

 box.y = mouseY;

 box.vel.x = random(-3,3);

 box.vel.y = random(-3,3);

 box.mass = 100;

 }

}

Note: There is physics if items collide!

Try changing “presses” to “pressing”. What is the difference?

For more on motion, see <https://p5play.org/learn/sprite.html?page=2>

We can also specify how sprites behave with respect to physics. The default is sprites are “dynamic” and move according to physics and gravity. There is a global variable called world and we can set the gravity in the x and y direction. This line sets the gravity to move toward the bottom at an acceleration of 10 pixels/time unit:

 world.gravity.y = 10;

Try adding this to setup and see what happens!

If you don’t want gravity to affect a sprite then set its collider property be “static”. This creates a floor that doesn’t move:

let floor;

function setup() {

 new Canvas();

 background(255);

 world.gravity.y = 10;

 floor = new Sprite();

 floor.width = windowWidth - 100;

 floor.height = 10;

 floor.y = windowHeight - 75;

 floor.collider = "static";

}

Here is an example that creates a column of sprites that you can shoot with a “gun” controlled by the mouse. The gun is just a rectangular sprite that rotates to face the mouse location.

let gun;

function setup() {

 new Canvas();

 background(255);

 gun = new Sprite();

 gun.width = 50;

 gun.height = 10;

 gun.x = 30;

 gun.y = windowHeight / 2;

 gun.collider = "none";

 // Create column of targets

 for (let i = 1; i < 20; i++) {

 let target = new Sprite();

 target.width = 20;

 target.height = 20;

 target.x = windowWidth - 50;

 target.y = (i \* windowHeight / 20)

 }

}

function draw() {

 clear();

 gun.rotateTowards(mouse, 0.1, 0);

 if (mouse.pressing()) {

 let circle = new Sprite();

 circle.diameter = 10;

 circle.x = gun.x;

 circle.y = gun.y;

 circle.mass = 100;

 circle.moveTowards(mouse, 0.03); // Impulse

 }

}

See <https://p5play.org/learn/sprite.html> for more information!

## Coding Challenge!

Make a game, gravity pong, where the ball starts in the upper left corner with an x velocity of 1. There should be static walls on the left and right but the top and bottom can be open. Place a paddle sprite with width 50 near the bottom of the screen. The paddle should also be static and move with the mouse. The world should have a downward gravity – you can try setting it to 10. Make the ball’s “bounciness” set to 1 (see <https://p5play.org/learn/sprite.html?page=9>).



For an extra challenge, add a way to inject some random velocity to the ball, and add some items for the ball to hit in the field of play.

Share your sketch in the “Tuesday submissions” folder.

**Groups**

Normally when we create a set of similar sprites we put them into a **group**. The group allows us to manipulate the sprites all as one or to take actions when something happens to one of the sprites in the group (like there is a collision). In this variation we put all the targets into a group and then if a bullet is touching one we call the **collect** function to remove the target and play a sound.

let gun, targets, mono;

function setup() {

 new Canvas();

 background(255);

 mono = new p5.MonoSynth();

 gun = new Sprite();

 gun.width = 50;

 gun.height = 10;

 gun.x = 30;

 gun.y = windowHeight / 2;

 gun.collider = "none";

 targets = new Group();

 // Set default values for each target in the group

 targets.width = 20;

 targets.height = 20;

 targets.x = windowWidth - 50;

 while (targets.length < 20) {

 let target = new targets.Sprite();

 target.y = (targets.length+1)\*windowHeight/20;

 }

}

function draw() {

 clear();

 gun.rotateTowards(mouse, 0.1, 0);

 if (mouse.pressing()) {

 let circle = new Sprite();

 circle.diameter = 10;

 circle.x = gun.x;

 circle.y = gun.y;

 circle.mass = 100;

 circle.moveTowards(mouse, 0.03); // Impulse

 circle.overlaps(targets, collect);

 }

}

function collect(projectile, target) {

 target.remove();

 mono.play(300, 1, 0, 1/4);

}

**Loading pre-made sprites**

You can also load pre-made sprites to have more engaging graphics!

I used <http://piskelapp.com> to make these simple sprites. You can find online sprite editors and free pre-made sprites with some google searching. Here is the smily sprite. It is 200x200 pixels. This is kind of large, usually our sprites will be smaller (the default on piskelapp is 32x32). The background is transparent. Images should be saved as .png.

smily0



smily1



surprise0



We could make a set of sprites with fancier animations, like a character walking, something exploding, etc.

I also made two rectangle sprites, in slightly different shades of red:

rect0



rect1



I dragged all these files into the “Files” tab so they are accessible to p5.js.



Each sprite is able to have multiple animations. Let’s make it so we have two sprites:

 smily:

 “eyes” – loop between smily0 and smily1

 “surprise” – loop between surprise0 and itself, so just one image

rectangle:

 “colors” – loop between rect0 and rect1

 When would you use multiple animations? For example, if you have a player character, you might have a “run left” animation that is separate from “run right” or “jump”. In this case I only made a “surprise” and “eyes” animations for the smily face.

To animate these sprites, make global variables for each sprite, and then load them in the preload function. Frame delay controls how long to wait before switching to the next frame in the animation, so the rectangle will animate more slowly than the smily face. In loadAnimation we give the first frame and the last frame. The files should be numbered accordingly in between.

let smily, rectangle;

function setup() {

 new Canvas();

 background(255);

 // x y w h collider

 smily = new Sprite(100, 100, 200, 200, "dynamic");

 smily.addAni("eyes", "smily0.png", 1); // 2 images (0 to 1.png)

 smily.addAni("surprised", "surprise0.png", 0); // 1 image (0.png)

 smily.ani = "eyes";

 smily.ani.frameDelay = 100;

 rectangle = new Sprite(600,200,300,200, "dynamic");

 rectangle.addAni("rectangle", "rect0.png", 1); // 1 image (0.png)

 rectangle.ani = "rectangle";

 rectangle.ani.frameDelay = 200;

}

function draw() {

 clear();

 smily.ani.play();

 rectangle.ani.play();

}



We can make the smily sprite move around with the arrow keys by checking for keypresses in the draw function. This first version just adds to the x and y coordinates:

function draw() {

 clear();

 if (kb.pressing("right")) {

 smily.position.x = smily.position.x + 5;

 }

 if (kb.pressing("left")) {

 smily.position.x = smily.position.x - 5;

 }

 if (kb.pressing("down")) {

 smily.position.y = smily.position.y + 5;

 }

 if (kb.pressing("up")) {

 smily.position.y = smily.position.y - 5;

 }

 smily.ani.play();

 rectangle.ani.play();

}

Another way we can move a sprite is by assigning a velocity.

function draw() {

 clear();

 if (kb.pressing("right")) {

 smily.vel.x = smily.vel.x + 0.5;

 }

 if (kb.pressing("left")) {

 smily.vel.x = smily.vel.x - 0.5;

 }

 if (kb.pressing("down")) {

 smily.vel.y = smily.vel.y + 0.5;

 }

 if (kb.pressing("up")) {

 smily.vel.y = smily.vel.y - 0.5;

 }

 smily.ani.play();

 rectangle.ani.play();

}

Finally, lets show how we can change the animation for a sprite. We simply set the ani property to the desired animation. Recall that the smily sprite has two animations, “eyes” and “surprised”. The following example changes the smily to “surprised” if it collides with the rectangle:

function draw() {

 clear();

 if (kb.pressing("right")) {

 smily.vel.x = smily.vel.x + 0.5;

 }

 if (kb.pressing("left")) {

 smily.vel.x = smily.vel.x - 0.5;

 }

 if (kb.pressing("down")) {

 smily.vel.y = smily.vel.y + 0.5;

 }

 if (kb.pressing("up")) {

 smily.vel.y = smily.vel.y - 0.5;

 }

 **if (smily.collides(rectangle)) {**

 **smily.ani = "surprised";**

 **}**

 smily.ani.play();

 rectangle.ani.play();

}

This program is saved in the classroom folder as “Sprite Animation”

**Pixel Art**

P5play also has a way to create a sprite from text, where a letter specifies each pixel color. See <https://p5play.org/learn/sprite.html?page=3> for more information.

## Coding Challenge!

We’ve covered a lot!

Create a character sprite with at least two images in its animation and be able to move it around through a simple maze. The character shouldn’t be able to move through the maze walls, although you could add a “ghost” mode to go through walls if you like.

You don’t need to submit it yet (the project will continue in the next challenge).

Check out Quinton Ashley’s excellent p5play sketches at <https://openprocessing.org/user/350295?view=sketches&o=36> which includes a platformer, domino tower, and more!