Graphics Programming Course Notes

Felipe Balbi

May 10, 2020

Contents

Contents

Week 1

Key Concepts

- explain how transformations work

- describe how classes work

- use transformations to program a basic solar system

Welcome to graphics programming

We will use `p5.js` and the `brackets.io` editor.

Getting started on the module

Download the emptyExample.zip file from the link provided.

Basically, it's a follow-along coding session. A good remark is to refer to the documentation whenever we have doubts.

Using transformations

A `p5.js` sketch is made out of a canvas whose pixels can be addressed much like on a graph paper.

We can use `scale()`, `translate()`, and `rotate()` to apply transformations to the canvas. The functions `push()` and `pop()` let us create a *sandbox* of where transformations and styles will be applied.

Object Oriented Programming in Javascript (OOP)

Using the `class` keyword, we can define classes in JavaScript.

Week 2

Key Concepts

- describe how vectors work
- apply vector arithmetic
- implement simple systems that use vectors

Using vectors

Vectors have a direction and a magnitude. The `p5.js` library has a `vector` class for us to use.

Instead of calculating and updating each component of position, velocity, acceleration, friction, we can use vectors to raise the level of abstraction.

We can create a new vector with `createVector()` function.

Vector addition and subtraction

To add two vectors, we use the `add()` function which is part of the vector. Similarly for subtraction, we use the `sub()` function.

For example:

```
function draw() {
  vec = createVector(width / 2, height / 2);
  vec2 = p5.Vector.random2D();

  vec.add(vec2);
  v2.sub(vec);
}
```

Vector scaling

To scale a vector, we can multiply or divide the vector by a scaler. We can achieve this with `mult()` and `div()` functions.

Calculating magnitude and normalising

We can get the magnitude with `mag()`. We can normalize a vector with `normalize()`.

Acceleration 101

Acceleration is the rate of change of velocity of an object over time. Velocity is the rate of change of the location of an object over time.

When we want to update location based on velocity in `p5.j5` we use:

```
1  location.add(velocity)
```

Similarly, when we want to update velocity based on acceleration, we use:

```
1  velocity.add(acceleration)
```

Week 3

Key Concepts

- explain how forces work

- use physics concepts in animation scenarios

- implement simple physics systems

Introduction to forces

We'll see how forces relate to acceleration and how to simulate them in a simple game engine.

A force is a vector that causes an object with mass to accelerate.

With that in mind, we will try to have objects react to forces applied to them.

A quick recap of classical Newton Laws is necessary.

Newton's First Law An object at rest remains at rest and an object in motion remains in motion.

Newton's Second Law $Force = Mass \times Acceleration$

Newton's Third Law For every action, there is an equal and opposite reaction.

For now we will assume that all our objects have a mass of 1. This will simplify our calculations by not having to divide anything by the mass.

To implement Newton's Second Law in `P5.js` we will simply add all forces acting on an object to the object's acceleration. Like the code snippet below:

```
1  applyForce(force) {
2    this.acceleration.add(force)
3  }
```

With this, we can apply many different forces to the same object. Imagine we have a `car` object, we could apply several forces very easily with the method above:

```
1  car.applyForce(gravity)
2  car.applyForce(friction)
3  car.applyForce(wind)
4  car.applyForce(engine)
```

Coding gravity and friction

Gravity is one of the 4 fundamental forces of the universe. It's a curvature in the Space-time fabric of the Universe which causes two objects to attract to one another.

In `P5.js`, gravity is essentially a vector without an `x` component. We will use a vector of size `(0, 0.1)` but we could simulate different gravity by changing the `y` component.

While this is enough to simulate gravity by itself, it doesn't look realistic because we're not simulating the friction of the ball with the air or the friction of the ball with the floor.

We can easily do that by creating new vectors and adding them to the ball with `applyForce()`.

To calculate the friction we follow a simple method:

1. Get the velocity vector

2. Calculate the opposite vector

3. Scale by a friction coefficient

4. Apply to Object

For our purposes, all surfaces have the same friction coefficient of `0.01`. Therefore, we can calculate friction with the following code:

```
1  let friction = ball.velocity.copy()
2  friction.mult(-1)
3  friction.normalize()
4  friction.mult(0.01)
5  ball.applyForce(friction)
```

Introducing collision detection

Collision detection is the computational problem of detecting the intersection of two or more objects.

Collision detection is a complex problem that grows with the amount of objects in the scene. To manage the complexity, the problem is broken down into two phases: Broad and Narrow.

During the broad phase we find pairs of rigid bodies that might be colliding with one another. We employ space partitioning and/or bouding boxes to simplify this method.

After we have reduced the number of comparisons during the broad phase, the narrow phase will kick in and employ shape-specific collision detection. In essence, we should look at all points of object A and check if it's inside the boundaries of object B.

For example, to check if a point is inside a circle, we simply check if the distance betwee the center of the circle and that point is less than the radius of the circle.

In `P5.js`, we can use the `dist()` function for this:

```
1  if (dist(pointX, pointY, circleX, circleY) < circleRadius) {
2    /* point is inside circle */
3  }
```

We can expand this to check collision between two circles. In summmary, we just check if the distance between the centers of both circles is less than the sum of the radii of both circles.

```
1  if (dist(circleAX, circleAY, circleBX, circleCY)
2      < circleARadius + circleBRadius) {
3    /* circles are colliding */
4  }
```

Week 4

Key Concepts

- describe what physics engines are and what they do

- describe the basic elements of matter.js

- implement simple physics systems using matter.js

Introduction to matter.js

A physics engine simplifies the work of simulating physical forces and interactions.

When dealing with complex shapes, sophisticated algorithms must be used to calculate collision between objects, that's where a physics engine comes in.

Instead of computing all object locations and collisions, we ask the physics engine what we should do and just draw the object at the exact location.

Matter.js is a simple library implementing a 2D physics engine.

matter.js resources

Below are some matter.js resources which you might find useful as you're working on the programming exercises.

I would advise that you click on a some of them right now and browse for a few minutes.

- Matter.js website for a 2D physics engine for the web

- Matter.js mixed shape demo

- Matter.js API documentation

- Matter.js Wiki pages

- Information on how to use Matter.js

- Link to the samples directory for examples

- Github page

Basic elements of matter.js

Integrating `Matter.js` with our `P5.js` sketches is a simple task. To make it easier, we will create some variables to alias `Matter.js` elements:

```
1  let Engine = Matter.Engine
2  let Render = Matter.Render
3  let World = Matter.World
4  let Bodies = Matter.Bodies
```

From that point on, we need to create some `Bodies` and add them to the `World` before being able to run the `Engine`. Let's do that:

```
1   let engine;
2   let box1;
3
4   function setup() {
5     createCanvas(900, 600);
6
7     /* Create an Engine */
8     engine = Engine.create();
9
10    /* Create a square */
11    box1 = Bodies.rectangle(200, 200, 80, 80);
12
13    /* Add the square to the world */
14    World.add(engine.world, [box1]);
15  }
```

With this piece of code we have setup the world for running our 2D simulation. Next step is to update and draw the box in the world:

```
1   function update() {
2     background(0);
3     Engine.update(engine);
4
5     push();
6     fill(255);
7     let pos = box1.position;
8     translate(pos.x, pos.y);
9     rotate(box1.angle);
10    rect(0, 0, 80, 80);
11    pop();
12  }
```

11

After these two functions, we should have a box falling forever without a ground to collide. Adding a ground we have:

```javascript
let Engine = Matter.Engine
let Render = Matter.Render
let World = Matter.World
let Bodies = Matter.Bodies

let engine;
let ground;
let box1;

function setup() {
  createCanvas(900, 600);

  /* Create an Engine */
  engine = Engine.create();

  /* Create a square */
  box1 = Bodies.rectangle(200, 200, 80, 80);

  /* Create a ground */
  let options = {
    isStatic: true,
    angle: Math.PI * 0.6
  };
  ground = Bodies.rectagle(400, 500, 810, 10, options);

  /* Add the square to the world */
  World.add(engine.world, [box1, ground]);
}

function update() {
  background(0);
  Engine.update(engine);

  push();
  rectMode(CENTER);
  fill(255);
  let pos = box1.position;
  translate(pos.x, pos.y);
  rotate(box1.angle);
  rect(0, 0, 80, 80);
  pop();
```

```
42
43    push();
44    rectMode(CENTER);
45    fill(255);
46    let groundPos = ground.position;
47    translate(groundPos.x, groundPos.y);
48    rotate(ground.angle);
49    rect(0, 0, 810, 10);
50    pop();
51  }
```

To simplify the code a little, we can draw shapes using their vertices. Like shown below:

```
1   let Engine = Matter.Engine
2   let Render = Matter.Render
3   let World = Matter.World
4   let Bodies = Matter.Bodies
5
6   let engine;
7   let ground;
8   let box1;
9
10  function setup() {
11    createCanvas(900, 600);
12
13    /* Create an Engine */
14    engine = Engine.create();
15
16    /* Create a square */
17    box1 = Bodies.rectangle(200, 200, 80, 80);
18
19    /* Create a ground */
20    let options = {
21      isStatic: true,
22      angle: Math.PI * 0.6
23    };
24    ground = Bodies.rectagle(400, 500, 810, 10, options);
25
26    /* Add the square to the world */
27    World.add(engine.world, [box1, ground]);
28  }
29
30  function drawVertices(vertices) {
```

```
31    beginShape();
32    vertices.forEach(v => {
33      vertex(v.x, v.y);
34    })
35    endShape(CLOSED);
36  }
37
38  function update() {
39    background(0);
40    Engine.update(engine);
41
42    fill(255);
43    drawVertices(box1.vertices);
44
45    fill(125);
46    drawVertices(ground.vertices);
47  }
```

Adding other types of bodies

`Matter.js` has several types of bodies as can be seen from the documentation.

Adding and deleting multiple bodies

To simplify the test of adding multiple objects, we can create a helper function that creates new objects for us:

```
1  var boxes = []
2
3  function generateObject(x, y) {
4    var b = Bodies.rectangle(x, y, random(10, 30), random(10, 30),
5                            { restitution: 0.8, friction: 0.5 });
6    boxes.push(b);
7    World.add(engine.world, [b]);
8  }
```

After that, we need to draw our boxes by updating our `draw()` function:

```
1  function draw() {
2    /* ... */
3
4    fill(255);
5    for (var i = 0; i < boxes.length; i++) {
6      drawVertices(boxes[i].vertices);
```

```
7    }
8
9    /* ... */
10   }
```

The only thing left is to destroy objects that are outside of the user's view:

```
1    function isOffScreen(body) {
2      let pos = body.position;
3      return pos.y > height || pos.x < 0 || pos.x > width;
4    }
```

With that helper function, we can update `draw()` again:

```
1    function draw() {
2      /* ... */
3
4      fill(255);
5      for (var i = 0; i < boxes.length; i++) {
6        drawVertices(boxes[i].vertices);
7
8        if (isOffScreen(boxes[i])) {
9          World.remove(engine.world. boxes[i]));
10         boxes.splice(i, 1);
11         i -= 1;
12       }
13     }
14
15     /* ... */
16   }
```

Introducing constraints

A constraint is an entity that connects two bodies together. It has no geometry; its only purpose is to tie two objects together.

We add a constraint by providing two bodies and two points on those bodies to which the constraint is attached.

To use constraint, we need an alias for it at the top of our file:

```
1    let constraint = Matter.Constraint
```

After that, we create objects like before. The final step is to connect the objects together:

```
1  constraint1 = Constraint.create({
2    bodyA: objectA,
3    point1A: {x: 0, y: 0},
4    bodyB: objectB,
5    point1B: {x: -10, y: -10},
6  })
```

Adding mouse interaction

We can add mouse interaction using a mouse constraint. Documentation can be found here.

Much like before, we start by creating an alias:

```
1  let MouseConstraint = Matter.MouseConstraint;
2  let Mouse = Matter.Mouse;
```

After that, we create the mouse object by passing the html canvas element:

```
1  var mouse = Mouse.create(canvas.elt);
2  var mouseParams = {
3    mouse: mouse,
4  };
5  var mouseConstraint = MouseConstraint.create(engine, mouseParams);
6  mouseConstraint.mouse.pixelRatio = pixelDensity();
7  World.add(engine.world, mouseConstraint);
```

Week 5

Key Concepts

- describe what physics engines are and what they do

- describe the basic elements of matter.js

- implement simple physics systems using matter.js

Animating a static object - propeller

Matter.js also has the hability of giving bodies angular velocity. We can use that to simulate a propeler object.

Below we can see an example of how to achieve a rotating properller that pushes objects touching it.

```
1   /* ... */
2
3   var Body = Matter.Body;
4
5   /* ... */
6
7   var angularVelocity = 0.1;
8   var angle = 0;
9   var propeller;
10
11  /* ... */
12
13  function setup() {
14    /* ... */
15
16    propeller = Bodies.rectangle(width / 2, height / 2, 300, 15,
17                                { isStatic: true, angle: angle });
18    World.add(engine.world, [propeller]);
19
20    /* ... */
21  }
22
23  function draw() {
```

```
24    /* ... */
25
26    fill(255);
27    drawVertices(propeller.vertices);
28    Body.setAngle(propeller, angle);
29    Body.setAngularVelocity(propeller, angularVelocity);
30    angle += angularVelocity;
31
32    /* ... */
33  }
```

Week 6

Key Concepts

- explain what generative art and design is

- identify important characteristics of generative art

- apply randomness and noise to create simple generative systems

Introduction to generative art and design

Generative Art refers to art that has been created with the use of an autonomous system. It means an autonomous systems (possibly a computer system) determines features of an artwork which would, generally, require decisions made by an artist.

When we write code, in general, we're dealing with very precise constructs: the exact location of a square in a P5JS canvas, for example. Art, on the other hand, is very subjective and debatable.

Generative Art is where programming and art collide. We employ strictly defined computer procedures and use them to create unpredictable and expressive artwork.

In summary, we set rules in a series of logical decisions and mathematical formulae and that guides the generation of different shapes and forms.

Generative Art can be employed in procedural generation of game maps. For example, that was used in the old River Raid game in the form of an LFSR for generating the maps and enemies in a quasi-random method. Similarly, Minecraft generates its worlds procedurally.

Nature and use of randomness

Randomness is one of the main pillars of genetive art. It's the easiest way to hand over the control of a process to an algorithm.

A popular way to generate random numbers, is to measure the decay in radioactive material over a short timescale. Such a system is guaranteed to be random to everyone.

In general, our computers generate what we call *pseudorandom numbers*. They are "random enough" for most applications; i.e., they give the feeling of randomness while still being predictable. To do this, such algorithms use an initial value, called the *seed*. If the seed is known, the entire sequence can be regenerated.

Introduction to Perlin noise

Perlin Noise was created by Ken Perlin in the early 1980s, first used in the film Tron (1982).

Perlin Noise allows for a more organic appearance and produces a naturally ordered sequence of pseudorandom numbers.

2D noise

Perlin noise can also be modeled as a 2-dimensional plane. The `noise()` function is P5JS can take an optional `y` parameter.

3D noise

Finally, Perlin noise can also be treated as a volume. All we have to do is pass a 3rd optional argument `z1 to the =noise()` function in P5JS and we will take samples from the noise volume.

In basic, we treat the `z` coordinate as time, almost as if we were taking slices of time on the noise volume. A simple trick is to use `frameCount` as the `z` coordinate, seen as that gets incremented at each frame. To make the steps smaller, we can divide `frameCount` by a somewhat large constant, like `50`.

Week 7

Key Concepts

- explain what generative art and design is

- identify important characteristics of generative art

- apply randomness and noise to create simple generative systems

This week is only a programming assignment

Week 8

Key Concepts

- use trigonometry to make shapes

- use oscillation to code movement

- implement generative systems using additive synthesis

- implement recursive systems to make fractals

Trigonometry refresher

Trigonometry will enable us to achieve some cool things, with regards to generative art and procedural generation.

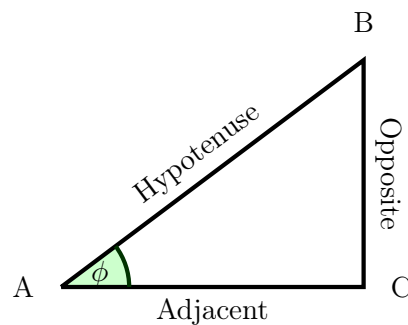Figure 1 shows a right-angle triangle to help us refresh the basics of trigonometry.

Figure 1: Right-angle Triangle

Given the triangle in figure 1, we can state the following:

- $\sin \phi = \frac{opposite}{\text{hypotenuse}}$

- $\cos \phi = \frac{adjacent}{\text{hypotenuse}}$

- $\tan \phi = \frac{opposite}{\text{adjacent}}$

Polar to cartestian coordinates

So far we have been dealing only with Screen Coordinates, which is similar to Cartesian Coordinates. The difference between Cartesian Coordinates is that Y grows upward, while Scren Coordinates have Y growing downwards.

Cartesian (and Screen) Coordinates work very well in a rectangular plane.

Another way to think of points in a 2D space is by considering the length of a vector and the angle formed between our vector and x axis. That's the essence of the Polar Coordinate system, in which points are determined by the distance from the origin in an angle from the polar axis.

The radial coordinate is often denotes by r and the angle θ.

While Polar Coordinates make far easier to describe rotational movement, all the primitives in `P5.js` work with Cartesian Coordinates. This is where Trigonometry helps us.

We can use Trigonometry to convert from Polar Cartesian and back.

Below, we can find a table of equivalence for converting Polar to Cartesian.

Polar To Cartesian	Cartesian To Polar
$x = r\cos(\theta)$	$r = \sqrt{x^2 + y^2}$
$y = r\sin(\theta)$	$\theta = \tan^{-1}\left(\frac{y}{x}\right)$

Coding a circle

Let's say we want to arrange a bunch of circles around a center. In other words, a circle of circles.

We can achieve this using polar coordinates:

```
1   function setup() {
2     createCanvas(500, 500);
3     background(0);
4     angleMode(DEGRESS);
5   }
6
7   function draw() {
8     background(0);
9
10    translate(width / 2, height / 2);
11
12    fill(255);
13    var radius = 200;
14
15    for (var theta = 0; theta < 360; theta +=20) {
16      var x = radius * cos(theta);
17      var y = radius * sin(theta);
```

```
18
19      ellipse(x, y, 15, 15);
20    }
21  }
```

Given that circle of circles, how would we go about rotating it?

```
1   function setup() {
2     createCanvas(500, 500);
3     background(0);
4     angleMode(DEGRESS);
5   }
6
7   function draw() {
8     background(0);
9
10    translate(width / 2, height / 2);
11
12    fill(255);
13    var radius = 200;
14    var theta = frameCount;
15
16    var x = radius * cos(theta);
17    var y = radius * sin(theta);
18
19    ellipse(x, y, 15, 15);
20  }
```

We ca use this to produce the Archimedean Spiral:

```
1   function setup() {
2     createCanvas(500, 500);
3     background(0);
4     angleMode(DEGRESS);
5   }
6
7   function draw() {
8     noStroke();
9     translate(width / 2, height / 2);
10
11    fill(255);
12    var radius = frameCount / 10;
13    var theta = frameCount;
14
15    var x = radius * cos(theta);
```

```
16    var y = radius * sin(theta);

17

18    ellipse(x, y, 15, 15);
19  }
```

Oscillation for movement

We can make use of sin and cos functions to produce oscillating movements like a pendulum. Both of these functios oscillate between -1 and 1.

The wave that is formed by plotting the sine function has an amplitude (the distance between the x axis and largest value) and a period (the amount of cycles within a time unit).

The period is the inverse of frequency, i.e. $f = \frac{1}{T}$, where f is the frequency in Hertz (Hz) and T is time in seconds (s).

Another important quantity is the phase of a sine wave. Phase denotes where the object is in the cycle.

Coding oscillation

Say we want to move a ball from left to right in an oscillating motion.

```
1   function setup() {
2     createCanvas(900, 600);
3     background(0);
4     angleMode(DEGREES);
5   }

6

7   function draw() {
8     background(0);

9

10    fill(255);
11    translate(width / 2, height / 2);

12

13    var amp = width / 2;
14    var period = 120;
15    var phase = 0;
16    var freq = 1;
17    // var locX = sin(360 * frameCount/period + phase) * amp;
18    var locX = sin(freq * frameCount/period + phase) * amp;

19

20    ellipse(locX, 0, 30, 30);
21  }
```

Using additive synthesis

For occasions when we want to add sine waves together and, perhaps, add noise on top, we can use additive synthesis.

```
1  function setup() {
2    createCanvas(900, 600);
3    background(0);
4    angleMode(DEGREES);
5  }
6
7  function draw() {
8    background(0);
9
10   stroke(255);
11   translate(0, height / 2);
12
13   beginShape();
14   for (var x = 0; x <= width; x++) {
15     var wave1 = sine(x + frameCount) * height / 4;
16     var wave2 = sine(x * 10 + frameCount) * height / 20;
17     var wave3 = noise(x / 10 + frameCount / 100) * 100;
18     vertex(x, wave1 + wave2 + wave3);
19   }
20   endShape();
21 }
```