Graphics Programming Course Notes

Felipe Balbi

April 27, 2020

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:

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

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

```
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:

```
let friction = ball.velocity.copy()
friction.mult(-1)
friction.normalize()
friction.mult(0.01)
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:

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

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.

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

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:

```
let Engine = Matter.Engine
let Render = Matter.Render
let World = Matter.World
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:

```
let engine;
let box1;

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

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

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

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

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:

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

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

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

```
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);
```

```
  }

  /* ... */
}
```

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

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

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

```
function draw() {
  /* ... */

  fill(255);
  for (var i = 0; i < boxes.length; i++) {
    drawVertices(boxes[i].vertices);

    if (isOffScreen(boxes[i])) {
      World.remove(engine.world. boxes[i]));
      boxes.splice(i, 1);
      i -= 1;
    }
  }

  /* ... */
}
```

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:

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

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

```
constraint1 = Constraint.create({
  bodyA: objectA,
  point1A: {x: 0, y: 0},
  bodyB: objectB,
  point1B: {x: -10, y: -10},
})
```

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:

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

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

```
var mouse = Mouse.create(canvas.elt);
var mouseParams = {
  mouse: mouse,
};
var mouseConstraint = MouseConstraint.create(engine, mouseParams);
mouseConstraint.mouse.pixelRatio = pixelDensity();
World.add(engine.world, mouseConstraint);
```