Graphics Programming

Course Notes

Felipe Balbi

May 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);
```

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.

```
/* ... */

var Body = Matter.Body;

/* ... */

var angularVelocity = 0.1;
var angle = 0;
var propeller;

/* ... */

function setup() {
  /* ... */

  propeller = Bodies.rectangle(width / 2, height / 2, 300, 15,
                               { isStatic: true, angle: angle });
  World.add(engine.world, [propeller]);

  /* ... */
}

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.

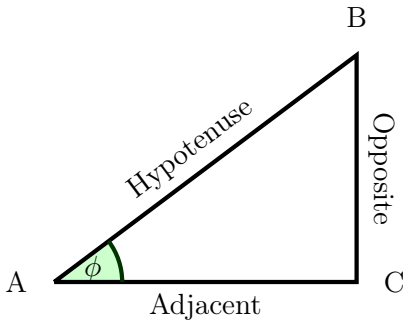

Figure 1: Right-angle Triangle

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:

```
function setup() {
  createCanvas(500, 500);
  background(0);
  angleMode(DEGRESS);
}

function draw() {
  background(0);

  translate(width / 2, height / 2);

  fill(255);
  var radius = 200;

  for (var theta = 0; theta < 360; theta +=20) {
    var x = radius * cos(theta);
    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);
```

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

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

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.

```
function setup() {
  createCanvas(900, 600);
  background(0);
  angleMode(DEGREES);
}

function draw() {
  background(0);

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

  var amp = width / 2;
  var period = 120;
  var phase = 0;
  var freq = 1;
  // var locX = sin(360 * frameCount/period + phase) * amp;
  var locX = sin(freq * frameCount/period + phase) * amp;

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

Using additive synthesis

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

```
function setup() {
  createCanvas(900, 600);
  background(0);
  angleMode(DEGREES);
}

function draw() {
  background(0);

  stroke(255);
  translate(0, height / 2);

  beginShape();
  for (var x = 0; x <= width; x++) {
    var wave1 = sine(x + frameCount) * height / 4;
    var wave2 = sine(x * 10 + frameCount) * height / 20;
    var wave3 = noise(x / 10 + frameCount / 100) * 100;
    vertex(x, wave1 + wave2 + wave3);
  }
  endShape();
}
```

Week 9

Key Concepts

- use trigonometry to make shapes
- use oscillation to code movement
- implement generative systems using additive synthesis
- implement recursive systems to make fractals

History of fractals

A fractal is a geometric shape that can be split into parts, each of which is a smaller version of the whole.

Figure 2 shows an example of a fractal.

Fractals appear everywhere in Nature: Clouds, Mountains, Moon Surface, Leaves of plants, Romanesco Broccoli, and many more.

Fractal structures have been used creatively in special effects techniques.

Sierpinski carpet

The Sierpinski carpet is a plane fractal first described by Wacław Sierpiński in 1916. The fractal starts with a square that's then cut into nine congruent sub squares in a 3×3 grid and a central subsquare is removed.

The same procedure is, then, applied recursively to the remaining of subsquares *ad infinitum*. A visualization of the Sierpinski carpet can be seen in figure 2.

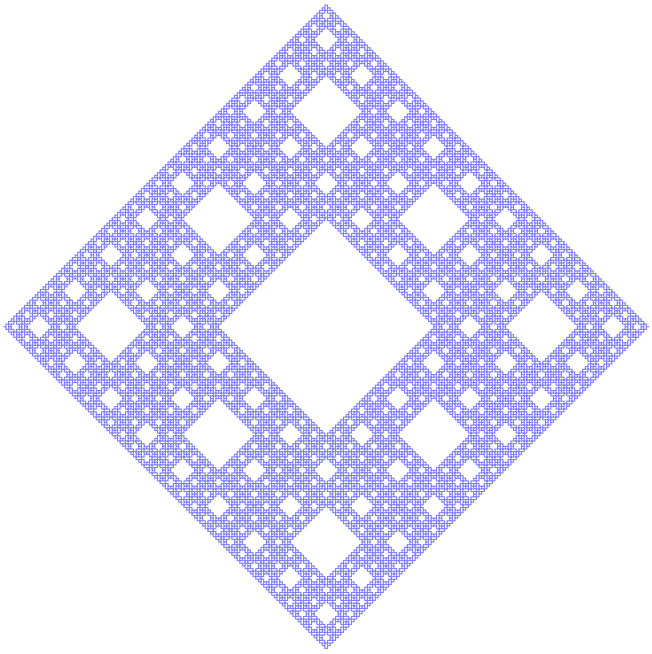

Figure 2: Sierpinski Carpet

The idea of subdividing a shape into smaller copies off itself and remove one of the copies can be applied to other shapes, such as the equilateral triangle (known as the Sierpinski Triangle).

We can implement these constructs in `P5.js` canvas.

```
var startSize;

function setup() {
  createCanvas(900, 600);
  background(255);

  fill(0);
  noStroke();
  noSmoote();
  rectMode(CENTER);

  startSize = pow(3, 6);
}

function draw() {
  translate(width / 2, height / 2);
  square(startSize);
  noLoop();
```

```
}

function square(startSize) {
  var side = side / 3;

  if (side >= 1) {
    rect(0, 0, side, side);

    push();
    translate(-side, 0);
    square(side);
    pop();

    push();
    translate(-side, -side);
    square(side);
    pop();

    push();
    translate(0, -side);
    square(side);
    pop();

    push();
    translate(side, -side);
    square(side);
    pop();

    push();
    translate(side, 0);
    square(side);
    pop();

    push();
    translate(side, side);
    square(side);
    pop();

    push();
    translate(0, side);
    square(side);
    pop();

    push();
```

```
63      translate(-side, 0);
64      square(side);
65      pop();
66    }
67  }
```

Week 10

Key Concepts

- explain what procedural generation for games is and what the pros and cons are
- identify ways Andy Lomas' work is linked to the syllabus

Interviewing a pro - Alan Zucconi

Alan Zucconi discusses his experience being an independent game developer.

Interviewing a pro - Andy Lomas

Andy Locam discusses his experience being a computational artist and educator. He has worked as a computer graphics supervisor for the Matrix films and Avatar.

Week 11

Key Concepts

- explain the difference between 2D and 3D graphics
- use 3D primitives, lights and materials to program simple animations
- implement 3D animations by manipulating the camera parameters

Introduction to 3D graphics

It's time to introduce a new dimension *<insert awesome explosions here>*.

As a primer, the 3D graphics in our scene will be rendered by WebGL. By using WebGL, we get access to GPU-accelerated graphics.

To augment P5js with 3D capabilities, we simply add the third parameter `WEBGL` to our call to `createCanvas()`, like show below:

```
function setup() {
  createCanvas(500, 500, WEBGL);
}
```

The 3D canvas in P5js is slightly different from the 2D canvas. In 2D, the coordinate *(0, 0)* is the top-left of the screen, whereas in 3D that point sits in the center of the canvas.

In the following example, we create a 2D rectangle and apply a 3D rotation about the X axis.

```
function setup() {
  createCanvas(900, 600, WEBGL);
  angleMode(DEGREES);
}

function draw() {
  background(125);

  rectMode(CENTER);
  rotateX(frameCount);
  rect(0, 0, 100, 100);
}
```

We can replace the rectangle with a 3D object and rotate on all 3 axes.

```
function setup() {
  createCanvas(900, 600, WEBGL);
  angleMode(DEGREES);
}

function draw() {
  background(125);

  rectMode(CENTER);
  rotateX(frameCount);
  rotateY(frameCount);
  rotateZ(frameCount);
  box(200);
}
```

Introduction to materials and lights

Materials are an enhanced of textured mapping and a prerequisite for advanced shanding effects.

In the simulated 3D world, we define materials and lights and it is the 3D engine that calculates what the scene will look like.

The simplest material is the *Normal Material* which assigns a color relative to the object center for every 3D vertex. We assign the Red channel to X, Green channel to Y and Blue channel to Z. Ambient material reflects the light around it; specular material is like ambient but *shinier*.

P5 has 3 types of light:

Ambient Light floods scene from all directions

Point Light has a position and emits light in all directions

Directional Light infinitely far away, emits light towards a direction

```
function setup() {
  createCanvas(900, 600, WEBGL);
}

function draw() {
  background(125);

  ambientMaterial(255);
  pointLight(255, 0, 0, 0, -200, 100);
  pointLight(255, 0, 0, mouseX - width / 2,
```

```
            mouseY - height / 2, 100);
  directionalLight(0, 0, 255, 1, 0, 0);
  sphere(100, 50, 50);
}
```

Camera

The rendered on the 3D scene created a default camera. The camera is the point from which we are viewing the world.

To change the camera settings, we use the `camera()` function which takes three parameters:

1. Position
2. Viewing Direction
3. Up Direction

These parameters should be self-explanatory: i.e. the position describes where in the 3D the camera is placed, the viewing direction describes which direction the camera is looking at and the up direction describes the orientation of the camera.

The example below, shows some of these features in action:

```
function setup() {
  createCanvas(900, 600, WEBGL);
  angleMode(DEGREES);
}

function draw() {
  background(125);

  var xLoc = cos(frameCount) * height;
  var yLoc = sin(frameCount) * 300;
  var zLoc = sin(frameCount) * height;
  camera(xLoc, yLoc, zLoc, 0, 0, 0, 0, 1, 0);

  normalMaterial();
  torus(200, 50, 50, 50, 50);
  translate(0, 100, 0);
  rotateX(90);
  fill(200);
  plane(500, 500);
}
```

Perspective

There are four characteristics of perspective:

1. Field of View
2. Aspect Ratio
3. Near Plane
4. Far Plane

In P5 the default Field Of View is 60°. We want the camera to be the same aspect ratio as the canvas, that's achieved with $\frac{width}{height}$.

```
function setup() {
  createCanvas(900, 600, WEBGL);
  angleMode(DEGREES);
}

function draw() {
  background(125);

  camera(0, 200, height, 0, 0, 0, 0, 1, 0);
  perspective(80, width / height, mouseY, mouseX);
  normalMaterial();

  for (var i = -600; i <= 600; i += 150) {
    push();
    translate(i, 0, 0);
    box(80, 80, 500);
    pop();
  }
}
```

Week 12

Key Concepts

- explain the difference between 2D and 3D graphics
- use 3D primitives, lights and materials to program simple animations
- implement 3D animations by manipulating the camera parameters

Simple texture coding

We can also use images as textures for 3D objects in P5. To do that we can use the texture function.

Here's a simple example showing how to apply an image as a texture to a 3D object:

```
let img;

function preload() {
  img = loadImage('assets/rocks.jpg');
}

function setup() {
  createCanvas(900, 600, WEBGL);
  angleMode(DEGREES);
}

function draw() {
  background(0);

  noStroke();
  texture(img);
  rotateY(frameCount);
  box(300);
}
```

Graphics as texture

We can also generate images programmatically and aply them as textures to 3D objects. Like shown below:

```
let buffer;

function setup() {
  createCanvas(900, 600, WEBGL);
  noStroke();
  buffer = createGraphics(300, 300);
  buffer.background(255);
  angleMode(DEGREES);
}

function draw() {
  background(125);

  buffer.fill(255, 0, 255);
  buffer.noStroke();
  buffer.ellipse(random(buffer.width), random(buffer.height), 10, 10);

  rotateY(frameCount);
  texture(buffer);
  sphere(100, 30, 30);
}
```

Week 13

Key Concepts

- explain the difference between bitmap and vector graphics
- use principles of colour theory to generate colour procedurally
- use images from the hard disk or from the webcam
- access and manipulate pixels directly

Colours

Daylight (or white light) is a combination of multiple wavelengths. This can be demonstrated by using a prism.

Our eyes have three photosensitive receptors: rods, cones, and intrinsically photosensitive retinal ganglion cells. The latter are believed to be more related to the circadian rhythm of our body than sight.

Rods are more involved with vision in low-light and, therefore, contribute little to colour vision. Cones are, therefore, our main colour receptors. There are three types of them: S, M, and L. Each type is sensitive to one of *Short*, *Medium*, and *Long* wavelenghts.

Our computer screen take advantage of that by having three different LEDs in each pixel (Red, Green, and Blue). Each of the three colours in a pixel can get a different brightness value. The same happens in P5js when we select a colour for an object. The RGB colourspace is referred to as an *Additive Colour* system.

In P5js, the brightness value of each colour channel ranges from 0 to 255. We refer to this is *24-bit colour*, because each colour needs 8 bits.

We can use P5js to traverse the RGB colourspace.

```
function seatup() {
  createCanvas(500, 500);

  colorMode(RGB);

  for (var r = 0; r < 256; r++) {
    for (var g = 0; g < 256; g++) {
      stroke(r, g, 0);
      point(r, g);
    }
```

```
  }

  noLoop();
}

function draw() {

}
```

It turns out that finding a specific color in RGB space is somewhat difficult. The HSB (Hue Saturation Brightness) space is a little easier. The example below plots a part of the HSB colourspace.

```
function seatup() {
  createCanvas(500, 500);

  colorMode(HSB);

  for (var h = 0; h < 360; h++) {
    for (var s = 0; s < 100; s++) {
      stroke(h, s, 100);
      point(h, s);
    }
  }

  noLoop();
}

function draw() {

}
```

Images

There are two general categories of images: bitmaps and vector images. Bitmaps are resolution-dependant, while vector images can be scaled at will without loosing resolution.

PNG is an open standard which supports an extra channel, the alpha channel, for transparency.

```
var img;

function preload() {
  img = loadImage("assets/rockets.png");
}
```

```
6
7  function setup() {
8    createCanvas(720, 400);
9  }
10
11 function draw() {
12   background(255);
13
14   image(img, mouseX, mouseY);
15 }
```

Using a webcam

We can collect the data coming from a webcam into a P5js sketch. Below, we have a simple sketch with the basics of how to access the webcam.

```
1  var vide;
2
3  function setup() {
4    createCanvas(640, 480);
5    pixelDensity(1);
6    video createCapture(VIDEO);
7    video.hide();
8  }
9
10 function draw() {
11   background(255);
12
13   /* Origin of image in the center */
14   imageMode(CENTER);
15
16   /* Translate to center of the screen */
17   translate(width / 2, height / 2);
18
19   /* Horizontal flip */
20   scale(-1, 1, 1);
21
22   /* Display image at (0, 0) */
23   image(video, 0, 0);
24 }
```

Pixels

Bitmaps are made of pixels. Pixels are essentially the smallest controllable element of a picture represented on the screen. The number of bits used to describe a colour depth.

Pixels are displayed in a grid with `(0, 0)` being the top-left corner, however they are stored in memory as a long array. Therefore, whenever we want to access a single pixel, we need to convert a 2D coordinate to a 1D coordinate. The equation is simple, see below:

$$pixelIndex = imgWidth \cdot y + x$$

We must also remember that each pixel needs 4 bytes of information due to *RGBA* encoding. Therefore, we must update our equation produce the corrected version shown below:

$$pixelIndex = (imgWidth \cdot y + x) \cdot 4$$

What follows is an example of applying this knowledge:

```
var img;

function preload() {
  img = loadImage('assets/rockets.png');
}

function setup() {
  createCanvas(600, 400);
  pixelDensity(1);
}

function draw() {
  background(255);
  image(img, 0, 0);

  img.loadPixels();

  var index = (img.width * mouseY + mouseX) * 4;
  var redC = img.pixels[index + 0];
  var greenC = img.pixels[index + 1];
  var blueC = img.pixels[index + 2];
  var alphaC = img.pixels[index + 3];

  fill(redC, greenC, blueC, alphaC);
```

```
25    rect(mouseX, mouseY, 50, 50);
26  }
```