Rocket Part 2

In the previous tutorial, we created a rocket which responds to key presses to alter its direction and apply thrust. This time we're going to add some additional features to our rocket, as well as incorporating some more objects to the environment.

To start with, let's implement a braking mechanism. This will essentially do the same thing as thrust, but in the opposite direction. However, since the intention is to allow the user to bring the rocket's velocity down to zero, we'll implement it slightly differently to thrust.

Braking

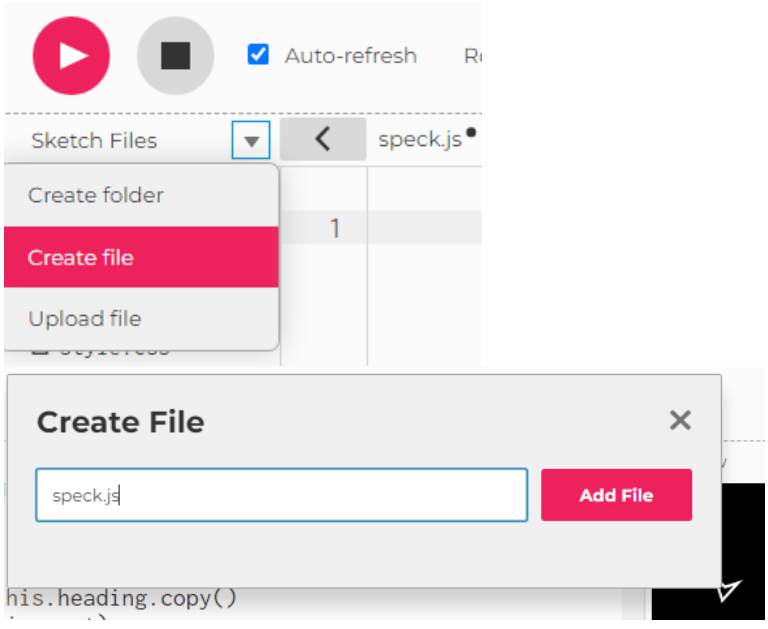
Recall that the built-in *keyIsDown()* function can be used to check at any point whether a given key is currently down. While the down arrow is pressed, I want to apply a braking force which will essentially reduce the magnitude of the velocity by a certain proportion (eg 10%).

```
checkKeys(){
  if (keyIsDown(UP_ARROW)) {this.thrust()}
  else if (keyIsDown(DOWN_ARROW)) {
    this.v.mult(0.9);
  }
```

Within the *Rocket* class code (inside the file *rocket.js*), I'm going to add a couple of lines to the *checkKeys()* method. You can play around with the value of 0.9 to decide what works best for you – for the speeds that my rocket goes, this gives a reasonable stopping distance, which should be enough to significantly aid avoidance of obstacles.

Exhaust

This is definitely a 'nice to have' rather than essential property, but it's not too tricky and makes for a nice effect. I'd like for small particles to be ejected behind the rocket whenever thrust is active. They will need some basic properties of their own, and the rocket can store them and update them within its own class code but they ideally want their own class. I'll call them specks. Let's start by making a new file for them:



Click the < arrow below the red play button if you can't see a list of files, then choose the down arrow next to 'Sketch Files' to create a new file.

Name it *speck.js.*

Don't forget to also head across to *index.html* and add in a line so that the webpage that displays our sketch knows to read this new file as well:

```
<body>
  <script src="sketch.js"></script>
  <script src="rocket.js"></script>
  <script src="speck.js"></script>
</body>
```

Planning the Speck class

Before jumping in, think carefully about what attributes and methods the *Speck* class should have. How will we create new specks, and how will we know when they're expired? How should they move, and what information should they be given when constructed so that they can?

The Speck Class

```
class Speck{
  constructor(position, velocity, lifespan){
    this.p = position.copy();
    this.v = velocity.copy();
    this.lifespan = lifespan;
    this.t = 0;
    this.live = true;
  }
  update(){
    this.p.add(this.v);
    this.t += 1;
    if (this.t > this.lifespan){
      this.live = false;
      return false;
    }
    return true;
  }
  display(){
    let col = map(this.t, 0, this.lifespan, 255, 0);
    noStroke();
    fill(col);
    circle(this.p.x, this.p.y, 3);
  }
}
```

This is a somewhat more basic construction than the ship, since the speck doesn't have to change direction or alter its velocity.

When we generate a speck, it will be told its initial position and velocity, along with a lifespan which will determine how long it stays active ('*live*') for.

The *update* method simply needs to change the position according to the velocity, and increment the timer to keep track of the age of the speck. I'm including a return value as well as setting a *this.live* flag as it might save some looping when the rocket is animating the specks.

Notice that the colour is being set according to how far through its lifespan a speck is, making it bright white when new, fading to black as it reaches the end of its lifespan. This should mean we get specks fading away rather than abruptly disappearing.

Keeping track of the specks

It will be the rocket's job to keep track of its exhaust specks, so we need to make an array to store them in, and make it an attribute of our *Rocket* class:

```
class Rocket{
  constructor(x, y, size){
    this.specks = [];
    this.p = createVector(x, y);
    this.size = size;
```

Initially, the array will be empty, but if it's already set up as a class attribute it'll be ready for when we want to push new specks into it.

Since we want to generate exhaust precisely when the rocket thrusts, we can build the call into the *thrust* method in the *Rocket* class:

```
thrust(){
  let f = this.heading.copy()
  f.mult(this.maxA);
  this.applyForce(f);
  this.createExhaust();
}
createExhaust(){
  let h = this.heading.copy();
  h.mult(-this.maxV);
  h.add(this.v);
  this.specks.push(new Speck(this.left , h, 10));
  this.specks.push(new Speck(this.right, h, 10));
  this.specks.push(new Speck(this.p    , h, 10));
}
```

All I've changed in the *thrust* method is a call to a new method, *createExhaust()*.

This new method also relies on the heading of the rocket, but this time modifies that unit vector so that it matches the maximum possible speed of the ship, but in the exact opposite direction to the ship's heading. I also add the current velocity of the ship since the speck is being ejected from the back of a moving rocket.

Notice how I'm making use of the coordinates saved earlier: *p* is the centre of the ship, while *left* and *right* correspond to the pointy ends of the 'wings' at the back. Each time *thrust* runs, I'll add three new specks to the array.

Garbage Collection

It is very important not to just keep adding specks without taking into account those that have expired. In order for them to move and appear on the screen, I'll need to update them. This should happen within the *update* and *display* methods of the *Rocket* class itself:

```
update(gravity=true){
  this.checkKeys();
  this.wrap();
  if (gravity){this.applyGravity()}
  this.a.limit(this.maxA);
  this.v.add(this.a);
  this.v.limit(this.maxV);
  this.p.add(this.v);
  this.a.mult(0);
  this.updateCorners();
  this.updateSpecks();
}
updateSpecks(){
  let updatedSpecks = [];
  for (let speck of this.specks){
    if (speck.update()){
      updatedSpecks.push(speck);
    }
  }
  this.specks = updatedSpecks;
}
```

I've added a line to the end of the *update* method which runs the dedicated *updateSpecks* method. This, in turn, defined just below, creates a fresh array each time it's called, and *speck.update()* is doing double duty here, since calling this function causes the speck to update its position, but it will also return a *true* or *false* which I can use to determine whether or not a given speck is still *live*. Note that, as it turns out, I'm not using the *this.live* flag that I built in, so I can probably remove that if I want.

Only if a speck is still *live* will it be added to the fresh specks array, which will then be used to overwrite the current specks array stored as *this.specks*.

Finally, we can add a few short lines to the end of the *display* method of the *Rocket* class:

```
for (let s of this.specks){
  s.display();
}
```

Since specks are being updated when the rocket runs its own *update* method, all that remains here is to display them.

Note that, since the specks are created by and stored within the *Rocket* class, there was no need to do anything with the original *sketch.js* file. As far as it knows, it's just updating and displaying a rocket, but the rocket in turn is taking care of the business of generating exhaust. This is the beauty of encapsulation: at each level, we see only what we need to see and no more. A formula 1 driver may very well benefit from understanding the mechanics of a steering rack, but when he's driving all he cares about is that his steering wheel controls the direction of the car.

Incorporating mass

Initially, to keep things simple, I ignored mass by conveniently imagining the rocket to have mass 1 so that $F = ma$ becomes $F = a$. However, it can be quite nice, if you want to simulate different rockets, to take into account the mass. We could even use *this.size* to give us a measure of the relative masses of ships.

```
let rockets = [];
let numRockets = 10;

function setup() {
  createCanvas(400, 400)
  for (let i=0; i<numRockets; i++){
    rockets.push(new Rocket(random(width),
                            random(height),
                            random(5, 20)));
  }
}

function draw() {
  background(0);
  for (let rocket of rockets){
    rocket.update();
    rocket.display();
  }
}
```

Back in *sketch.js*, instead of making just one rocket, we can quite easily churn out a whole fleet. Here I've made 10 rockets, with size varying from 5 to 20 pixels in radius.

Note that our controls are interpreted by every rocket when *update* runs, so your arrow keys will control all rockets simultaneously. If you want to make it even more challenging, give them a random starting velocity instead of just a random starting position.

Note that if I want to disable gravity, I can just use *rocket.update(false)* since I've set that as an optional argument in the *update* method.

Changing maximum acceleration

If we want mass to play a part, we need modify the amount of thrust that we produce. The simplest way is probably just to adjust the maximum possible acceleration and maximum possible speed according to the size right at the beginning:

```
this.m = this.size**3;
this.maxA = 30/this.m;
```

In the *Rocket* class *constructor* method, you can define the mass as the cube of the size, then use that to determine the maximum acceleration.

If you build a few rockets now, you'll see that the small ones are much more nippy, getting up to their top speed very rapidly. If you want an extra challenge, tweak the settings in *applyGravity* to make that respond differently to different masses as well.

Other objects

asteroid.js

```
class Asteroid{
  constructor(p, v, size){
```

I'll leave the details up to you, but see if you can construct a class for an *Asteroid* object that moves at a constant velocity, wrapping around the screen like our rocket. Don't forget if you make a new file like *asteroids.js* to add a reference to it in *index.html*.

Collisions

In order to turn this into a proper asteroids game, we're going to need two more things: collision detection and some form of weapon for destroying asteroids. Let's start with collisions:

```
collidesWith(other){
  return p5.Vector.sub(other.p, this.p).mag() < this.size + other.size;
}
```

This one line method that we could include in the *Rocket* class code essentially calculates the size of the vector between my position and that of any other object with a *p* attribute. Provided that object also has a *size* attribute, I can use it to determine whether the distance is less than the sum of our respective radii.

```
checkCollisions(){
  for (let asteroid of asteroids){
    if (this.collidesWith(asteroid)){
      this.lives -= 1;
    }
  }
}
```

This code within the *Rocket* class will loop through every asteroid and check to see if it's too close. If it is, it'll lose a life. In *constructor* we can set it to have, say, 3 lives initially, and if it loses all 3 we can check by asking if *rocket.lives* is zero. Don't forget to add a call to *checkCollisions* in the *update* method.

```
function draw() {
  background(0);
  for (let asteroid of asteroids){
    asteroid.update();
    asteroid.display();
  }
  let newRockets = [];
  for (let rocket of rockets){
    rocket.update(false);
    rocket.display();
    if (rocket.lives > 0){
      newRockets.push(rocket)
    }
  }
  rockets = newRockets;
}
```

My *draw* loop now updates and displays asteroids.

I also use a similar method to how we removed expired specks of exhaust: a new array is created, and rockets are only transferred to the new array if they still have lives remaining.

Start out with 10 rockets and see how long you can keep them all alive…