Rocket

In the style of classic games like Asteroids, we're going to create a rocket that the user will control with arrow keys, indirectly affecting motion in a realistic manner – by applying thrust in the direction the rocket is facing.

To begin with, let's see how p5 interprets key press signals from the user:

keyIsDown

There are a few ways to retrieve this information, but since different browsers interpret things differently, this seems to be the simplest for our purposes. Instead of using a *keyPressed()* function (which p5 supports, and will run whenever a key is pressed if you define it, just like *mousePressed()* does) we can simply invoke *keyIsDown()* at any suitable point in our code. It will tell us whether a particular key we're listening for is currently down. Unlike *keyPressed()*, which runs once when you hit a key, this will show up *true* every time you ask it provided the key is still down.

```
let x = 300;
let y = 300;

function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
  if (keyIsDown(LEFT_ARROW)){
    x += -2;
  }
  if (keyIsDown(UP_ARROW)){
    y += -3;
  }
  circle(x, y, 50);
}
```

No fancy object-oriented programming here – just keeping track of an *x* and a *y* variable, and changing them if particular keys are down.

Notice that the left arrow and up arrow may be simultaneously pressed, and in that case we get diagonal motion. See what happens if you replace the second *if* with *else if* and try to work out why.

Try adding in the additional commands needed for moving right and down. Can you see where *else if* would be useful here? Choose what behaviour you'd like to happen in the case where someone presses both left and right keys (eg hits left while holding down right) and code accordingly.

Note: for this to work, after hitting 'play' for your sketch, click somewhere in the canvas so that it becomes the 'active window' (otherwise your browser will not know you are trying to interact with the sketch).

Controlling motion realistically

By directly changing the position by a fixed amount, we are essentially directing the velocity of our object. A more realistic behaviour can be achieved by directing not the velocity but the acceleration of our object.

```
let [x,y] = [300, 300];
let [vx, vy] = [0, 0];
let acc = 0.1;

function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
  if (keyIsDown(LEFT_ARROW)){vx -= acc}
  else if (keyIsDown(RIGHT_ARROW)){vx += acc}
  if (keyIsDown(UP_ARROW)){vy -= acc}
  else if (keyIsDown(DOWN_ARROW)){vy += acc}
  x += vx;
  y += vy;
  circle(x, y, 50);
}
```

In this case, I've fixed the amount of velocity increase (named *acc*) to 0.1, so that the circle doesn't speed up too quickly when we hold down an arrow key.

I've tried to make things more readable by simultaneously defining *x* and *y* in an array, and the same with *vx* and *vy*.

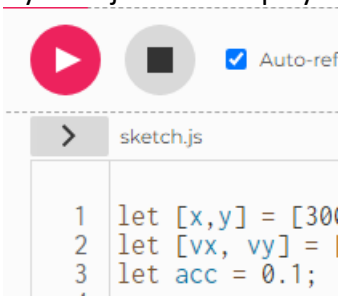
In the same vein, since I'm not doing much under each conditional statement, I've condensed each one into a single line, which aids readability.

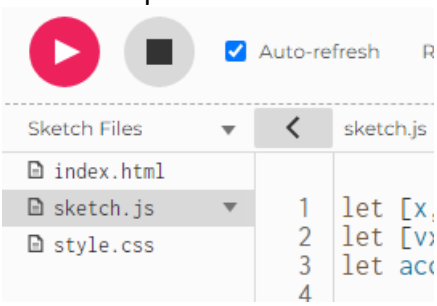
To make this more realistic still, I'd like to keep track of the direction the rocket is facing, so that, in order to accelerate in a given direction, it has to be facing that way. We could use left and right arrows to turn, and the up arrow to exert a thrust in the direction we're facing. But for this to work in practice, we'll need some kind of visual clue as to which way the rocket is facing. A circle isn't going to cut it…

The *Rocket* class – in its own separate file

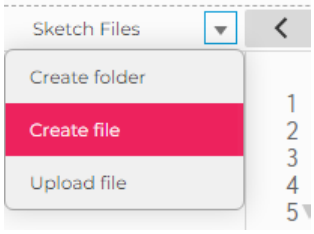
Up until now, we've been putting all our code in a single document, but for ease of editing, especially if things are likely to get complicated, we can make other *.js* documents within our sketch:

First, hit the > symbol just below 'play' to the left of your code. This will expand the files list.

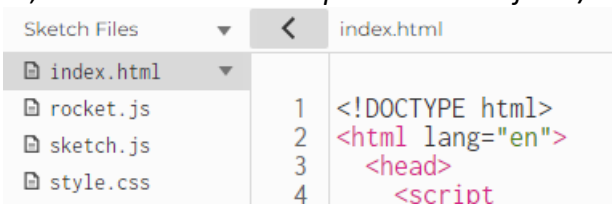




Next, choose the small drop-down arrow beside 'Sketch Files' and choose 'Create File'. Name it with the *.js* file extension, so that p5 knows it contains code to be evaluated, and hit 'Add File'.



Next, we need to add *rocket.js* to the list of JavaScript files the webpage checks, so go to the *index.html* file, and find the line *<script src="sketch.js"></script>*. Add a new line for *rocket.js*.



Tip: while in the middle of editing code, I recommend you disable 'Auto-refresh', because if the code runs and finds a bug, it will jump to the source of the error, which can mean a whole different page to the one you're currently editing:

...

```
10   <body>
11     <script src="sketch.js"></script>
12     <script src="rocket.js"></script>
13   </body>
14 </html>
15
```

Preparing for the *Rocket* class

So that we can see the effect of what we're doing as we go along, I'm going to write the code in *sketch.js* that will deal with the rocket before writing the rocket's code:

```
let rocket;

function setup() {
  createCanvas(400, 400)
  rocket = new Rocket(width/2, height/2, 40);
}

function draw() {
  background(0);
  rocket.update();
  rocket.display();
}
```

Declare a *rocket* variable in the global scope.

Define it in *setup* using sensible parameters (coordinates for initial position, and size).

Each time *draw* loops, we will update the rocket's position, orientation, etc, and then display it. All the details are left for the rocket class itself.

Notice that, now we've decided to move the *Rocket* class to its own dedicated page, the main *sketch* code fits nicely on one screen and is easy to read, check for errors or adapt.

The *Rocket* class code

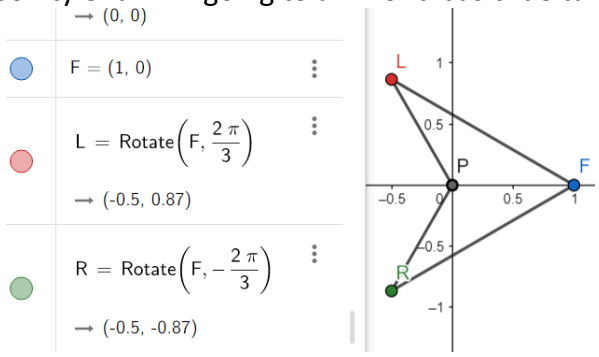
Finally, it's time to start writing the code for our rocket. Let's include all the key information we think we'll need in the *constructor* before worrying about the nuts and bolts of movement and key presses or display:

```
class Rocket{
  constructor(x, y, size){
    this.p = createVector(x, y);
    this.size = size;
    this.v = createVector(0, 0);
    this.a = createVector(0, 0);
    this.heading = createVector(1, 0);
    this.maxA = 0.1;
    this.maxV = 10;
    this.maxTurn = 0.1;
  }
```

We're going to update position indirectly, using the current velocity of the rocket. This, in turn, will be updated according to the current acceleration. We'll build in a way of exerting thrust, and when we do, we'll use *this.heading* to determine which way it acts before updating the acceleration. I've also build in some maximum values for speed, acceleration and amount of turn.

Designing a rocket

From a practical point of view, we want a design with a clear direction, so we know which way it's pointing. Aesthetically, it would be nice if this vaguely resembles some kind of rocket, which is also consistent with a pointy end. I'm going to aim for a basic 'delta' or 'arrowhead' shape, build around an equilateral triangle:



We will need to recreate this shape each time the ship is displayed, using stored information about the centre of the ship (its position) and its heading.

Consider the vectors $\overrightarrow{PF}$, $\overrightarrow{PR}$ and $\overrightarrow{PL}$, and how we might construct them from the heading and position (P) of our rocket.

Remember that we will be keeping track not only of the ship's position, velocity and acceleration, but also – separately – its heading. The direction the ship is moving and the direction the ship is pointing will not necessarily be the same.

It would probably be helpful to store the three points (F, L and R, or *front*, *left* and *right*) as class attributes, and just remember to update them before displaying the rocket each time. In the *Rocket* class:

```
updateCorners(){
  this.front = this.heading.copy().mult(this.size);
  this.right = this.heading.copy().mult(this.size);
  this.left = this.heading.copy().mult(this.size);
  this.right.rotate(2*PI/3);
  this.left.rotate(-2*PI/3);
  this.front.add(this.p);
  this.right.add(this.p);
  this.left.add(this.p);
}
```

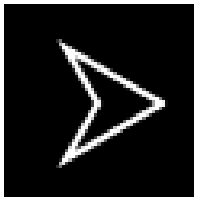
If we call this method every time the rocket updates, it will generate (and save / overwrite class attributes) the *front*, *right* and *left* vectors. Note that I'm using *this.heading*, scaled for the size of the rocket, to give the vector $\overrightarrow{PF}$, and the same for $\overrightarrow{PR}$ and $\overrightarrow{PL}$, but in those cases applying $120°$ rotations.

The final lines turn these vectors, which are relative to P, into absolute position vectors $\overrightarrow{OF}$, $\overrightarrow{OR}$ and $\overrightarrow{OL}$.

```
update(){
  this.updateCorners();
}
display(){
  noFill();
  stroke(255);
  strokeWeight(2);
  beginShape();
  vertex(this.p.x, this.p.y);
  vertex(this.left.x, this.left.y);
  vertex(this.front.x, this.front.y);
  vertex(this.right.x, this.right.y);
  vertex(this.p.x, this.p.y);
  endShape();
}
```

I'll add more to *update* in due course, but for now all I want it to do is update the corners using the code I just wrote.

The *display* method will use *beginShape()* and *endShape()* to draw around my rocket ship:



The physics of motion

We need to add some physics to the *update* method next, so that the ship is capable of movement:

```
update(){
  this.checkKeys();
  this.updateCorners();
  this.a.limit(this.maxA);
  this.v.add(this.a);
  this.v.limit(this.maxV);
  this.p.add(this.v);
  this.a.mult(0);
}

applyForce(f){
  this.a.add(f);
}
```

This code uses the built-in *limit* function for vectors, which does the same job as scaling a vector so it doesn't exceed a given value. A bit more efficient than working out the magnitude, comparing it to the maximum allowed, and then scaling if needed.

We increase velocity by acceleration, then position by velocity. Notice that at the end we set acceleration to zero once more. Only in the case where a force has been applied since the last time *update* was called will there be any acceleration.

Notice that I also included a *checkKeys* method, which I'll write in a moment, to get information from the user before applying the results.

Controlling thrust and checking key presses

```
thrust(){
  let f = this.heading.copy()
  f.mult(this.maxA);
  this.applyForce(f);
}

checkKeys(){
  if (keyIsDown(UP_ARROW)) {this.thrust()}
  if (keyIsDown(LEFT_ARROW)) {
    this.heading.rotate(-this.maxTurn);
  }
  else if (keyIsDown(RIGHT_ARROW)) {
    this.heading.rotate(this.maxTurn);
  }
}
```

If the engines are activated, that means we get maximum thrust (equivalent to maximum acceleration – for simplicity I'm treating mass as 1 unit, turning $F = ma$ into $F = a$) in the direction that the rocket is heading.

We apply thrust whenever the up arrow is down, and I want to rotate my heading vector if the left or right keys are down.

Building in a wrap-around universe

We don't want to lose our rocket, so let's make the screen 'wrap around'.

```
wrap(){
  if (this.p.x > width + this.size){
    this.p.x -= (width + 2 * this.size);
  }
  else if (this.p.x < -this.size){
    this.p.x += (width + 2*this.size);
  }
  if (this.p.y > height + this.size){
    this.p.y -= (height + 2 * this.size);
  }
  else if (this.p.y < -this.size){
    this.p.y += (height + 2*this.size);
  }
}
```

Note that the ship can disappear completely before reappearing, since I'm including *this.size* as a buffer outside the visible screen.

If the universe were a torus, this is precisely the behaviour we would see: leave the screen to the right, and you'll return on the left; go off the bottom and you'll come back from the top.

Don't forget to include a call to this method in *update* so that it actually happens. And note that I'm doing it before updating the corners, so that I don't get the middle of the ship on the opposite side of the screen to the rest of it!

```
update(){
  this.checkKeys();
  this.wrap();
  this.updateCorners();
```

Bonus: gravity!

Gravity acts towards the centre of the screen here, and follows an inverse square law.

```
applyGravity(x=width/2, y=height/2){
  let centre = createVector(x, y);
  let g = p5.Vector.sub(centre, this.p);
  let rSquared = max(g.mag()**2, 1);
  g.normalize().mult(1000/rSquared);
  g.limit(this.maxA);
  this.applyForce(g);
}
```

```
update(){
  this.checkKeys();
  this.wrap();
  this.updateCorners();
  this.applyGravity();
  this.a.limit(this.maxA);
```

Take care to limit gravity, and to not let it become infinite if you happen to be sitting right on top of the sun!