Creative Coding - Week 8

Physics and Game Feel with p5.js

Lecturer: Dr Michael Cook (`mike.cook@kcl.ac.uk`)

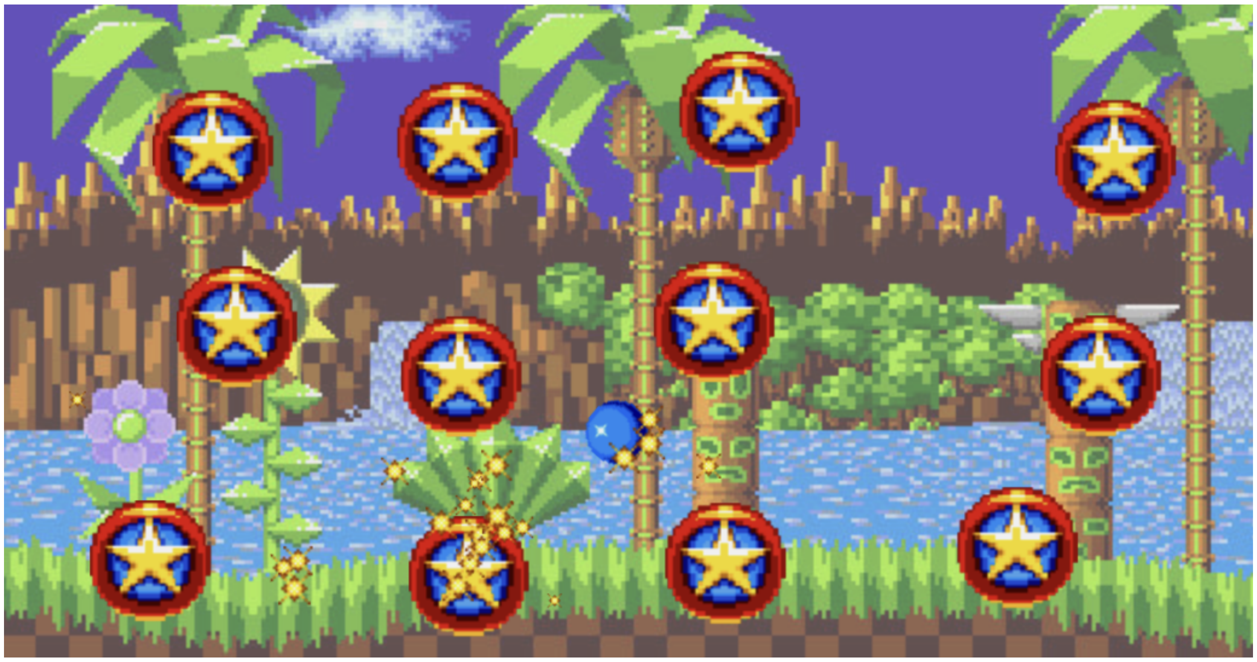


Figure 1: A physics-driven game with particle effects and bouncy orbs.

This week is the third and final week looking at game design, and the final week of the creative coding course as a whole. Well done for getting this far, and thank you so much for taking part! It's been a lot of fun going through this with you. This week we'll introduce another way to make games and interactive media, and talk a little bit about where to go next.

When it comes to making games people often want to know what the 'best' game-making tool or library is. Sometimes they want to know what is the easiest tool for porting your game to consoles or mobile, what tools are good for making commercial games, or what tools the current viral hit game was made in. For most people though, the best tool is the one you're *comfortable* using. If you like using your tools, and you understand them, then you'll be able to make things faster; you'll have the energy to finish them; you'll know what things the tool is good at and what things are harder to do; and you'll be able to improve your skills as a designer and developer. To that end, this week we're going back to p5.js, the tool we used in Weeks 1 and 2, and we're going to use an add-on library that makes game-making easier. Almost any programming language you can think of has game-making tools and libraries for it, so no matter what you like, you can find something to make games with.

Let's get started!

1 p5.js and p5.play

p5.js has a huge community and, as a result, a lot of add-ons and extensions. These include ml5.js (for machine learning), p5.speech (for generating and recognising speech), and p5.play (for making games) – the latter we'll be using today. To use libraries like this you usually just need to add a few lines to the `index.html` file in your p5.js project, but for today we'll be using a template that has already added these in (you can go and look to see how they work).

p5.play adds a number of things to p5 that we didn't have before, including:

- Quick creation of objects that stay on the screen and move around.
- Creation of objects with shared properties.
- Help with common input types/techniques.
- A 2D physics system for collision and movement.

This last point is particularly important for some kinds of game. p5.play uses Box2D, a physics system developed in 2007 that has been ported to almost every programming language you can think of, and used in thousands of games, including Angry Birds, Shovel Knight and in the Unity game engine. It is very powerful, very efficient, and we'll use it today.

Our goal today is to look at how physics games are made, and get used to the idea of creating objects and defining how they interact with each other. To get started, we're going to use a special p5.js template that includes p5.play in it.

Click here to open the template.

1.1 Creating Objects

Previously in p5.js, if we wanted to display something on the screen we would draw it in the `draw()` function. If we wanted the object to move across the screen, we would need to record its position somewhere else and then draw it each frame, like so:

```
let x = 100, y = 100, r = 20
function draw(){
    circle(x, y,r);
    x += 1
    y += 1
}
```

p5.play will handle this for us. All of the objects in our game will be represented as Sprite objects. In game design 'sprite' is a word used to describe a 2D image, used to represent part of a game like the player. In game development it has come to describe any self-contained object in a 2D game, so our sprite objects will contain lots more than just an image - it'll also record things like its position, rotation, movement and more. So instead of the above, we can write the following instead:

```
function setup(){
    new Canvas(400, 400);
    new Sprite(200, 200, 20);
}
```

This will add a new Sprite object to our scene, at the co-ordinates (200,200). Because we gave it a single extra parameter, it will be a circle of diameter 20. p5.play will create this Sprite and then keep track of it, always drawing it on screen where it should be, no matter what happens to it.

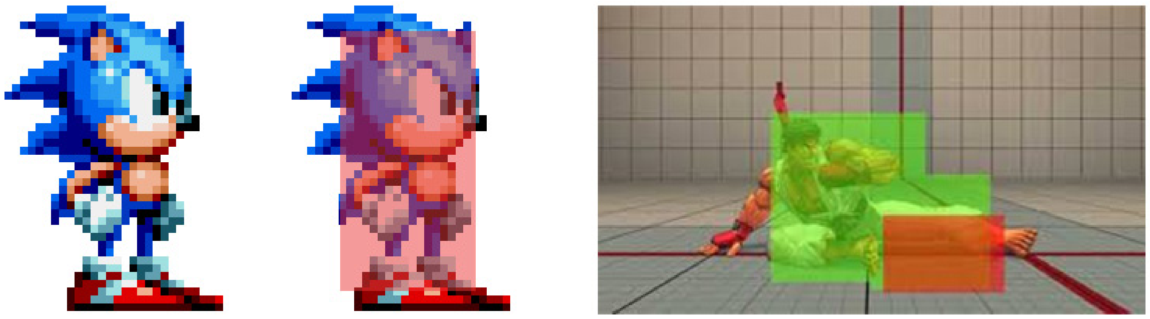


Figure 2: Left: Sonic's sprite from *Sonic Mania*. An (approximated) size of the actual collider for damage is shown overlaid. Right: Coloured boxes in *Street Fighter* indicate areas that deal damage (red) and areas that can be damaged (green), also represented as boxes.

1.2 Adding Physics

All Sprite objects are also part of p5p's physics system, which means they have a mass, a velocity, and so on. If we adjust this, the object will start moving. For example, try running this code:

```
function setup(){
    new Canvas(400, 400);
    var s = new Sprite(200, 200, 20);
    s.vel.x = 1
    s.vel.y = 1
}
```

This changes the x- and y- velocity of the circle, meaning it will move across and down. Again, p5.play will keep checking the status of this object every frame, and move it according to the physics simulation.

What happens if we put an object in the way? Click here to try the code below out.

```
function setup(){
    new Canvas(400, 400);
    //obstacles
    new Sprite(100, 110, 15);
    new Sprite(200, 190, 15);

    var s = new Sprite(30, 30, 30);
    s.vel.x = 1
    s.vel.y = 1
}
```

You should see the moving, larger circle bump into the smaller ones and knock them out of the way. By default, all sprites are part of the physics simulation, and all of them interact with each other.

1.3 Colliders and Collider Types

In game engines like this, physics objects often include a property or object called a *collider*. Colliders are the shapes that physics engines actually work with, and they then relay those effects to the game object as a whole. For example, in *Sonic Mania*, Sonic's collider might touch some spikes, which then tells the game object representing Sonic that he has taken damage. Physics engines use colliders for several reasons. A major one is efficiency. Checking collision between rectangles is a lot faster than checking collisions between a complex pixel shape like Sonic's sprite.

A second reason colliders look different is for game design tweaks. Colliders can be made bigger or smaller than reality in order to make the game easier or harder. For example, Sonic's collider doesn't quite touch his feet or the top of his head, which means the player can get very close to spikes or enemies and still escape, which is more exciting for the player and less frustrating. In p5.play, colliders for basic shapes are the exact same shape as the object itself (unless we tell it otherwise).

There are a few different types of collider in p5.play:

- Dynamic – The default type of collider. Collides with things, is affected by physics.
- Static – Other physics things can interact with it, but it cannot be moved.
- Kinematic – Can be moved using code, but is not affected by physics interactions.
- None – Has no collider, ignores all physics.

To test this out, let's change one of the colliders to static (this is a slight modification of the template linked above):

```
function setup(){
    new Canvas(400, 400);
    //obstacles
    var o = new Sprite(100, 110, 15);
    o.collider = 'static';
    new Sprite(200, 190, 15);

    var s = new Sprite(30, 30, 30);
    s.vel.x = 1
    s.vel.y = 1
}
```

Run this program. What happens?

1.4 Task 1: Pinball

Let's put this into practice! I'd like you to make a simple pinball game. I'll provide a little template, you need to add the following features:

- Create 5-10 obstacles for the pinballs to bounce off of. Create them in the `setup()` function, place them around the middle of the
- When the player clicks the mouse, create a new object. This object should appear at the position of the mouse, and should be a circle of 20 pixels in diameter. Check the `draw()` function to see a template for checking mouse clicks.

Find the template here. When you're done, you should be able to click to create balls, which fall and bounce off the obstacles you spawn. The start of a game!

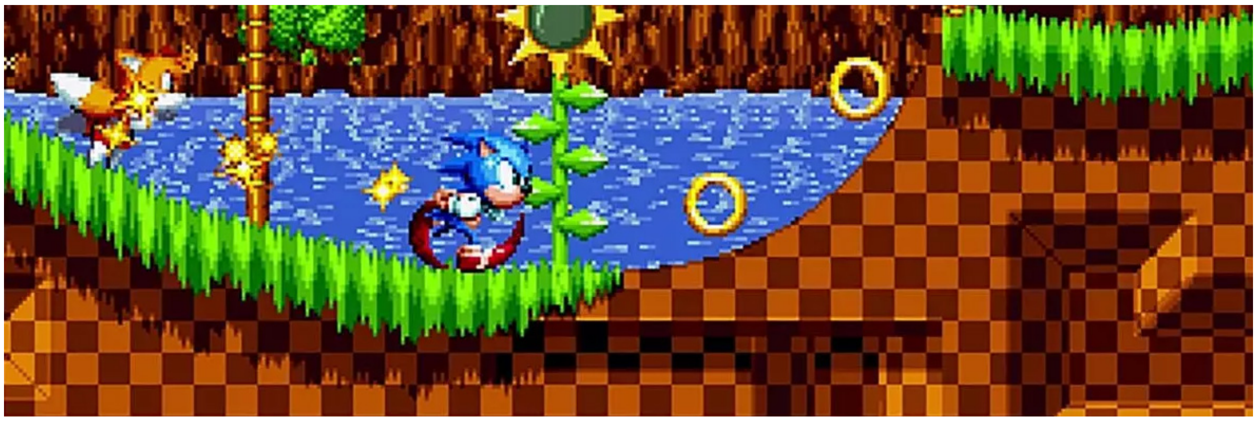


Figure 3: A screenshot from *Sonic Mania*.

1.5 Collisions and Groups

One of the main reasons we use a physics system is to detect interactions between two objects. There are two main types of physics interaction: collisions, when two solid objects hit each other, and overlaps, when a solid object passes over an object it doesn't collide with. In Figure 3 you can see a screenshot from *Sonic Mania*. Sonic *collides* with the floor - it pushes him up and stops him from falling down. But he doesn't collide with gold rings - instead, the game detects when he *overlaps* then and makes them disappear (to make it look like he picks them up).

We can add some collision logic to our game. First, let's define what happens when a ball collides with an obstacle. We can do this by writing a function with two arguments, the two objects that are colliding. Let's just make it move the obstacle down the screen each time:

```
function ballHitObstacle(ball, obs){
    //move the obs down
    obs.y += 10;
}
```

We can check when obstacles are colliding using the `.collided()` method, but we have many obstacles so there are a lot of checks to type in, and we don't know how many balls the player has spawned so we don't know how many there are to check. What we *really* want to say is 'this happens whenever *any* ball hits *any* obstacle'. In p5.play, we do this with *groups*.

A group is a set of sprites that all share properties. Whenever we want to have a set of objects all behave the same way, we create a group to represent them. We can create a group and set its properties just like a sprite:

```
let obstacles, balls;
function setup(){
    new Canvas(480, 640);
    obstacles = new Group();
    obstacles.collider = "static";
    //...
```

And then whenever we want to make a new member of this group, we call it from the group itself:

```
    new obstacles.Sprite(200, 250, 50)
    new obstacles.Sprite(300, 450, 50)
    new obstacles.Sprite(400, 350, 50)
```

This saves a lot of time! Now we just set the collider type to static once, and all obstacles share the same collider. We can do the same with other properties, let's make a group for the balls too:

```
    balls = new Group()
    balls.color = "#ff0000"
```

Now every new member of the balls group will be red. Now we've got groups, we can set a collision rule for both groups:

```
balls.collide(obstacles, bounce)
```

This says, whenever anything from the `balls` group hits anything from the `obstacles` group, call the `bounce` function. Now it doesn't matter how many balls the player spawns, or which obstacles they hit, the collision logic will always be applied consistently. You can find the full example using Groups here. Have a play around testing the sketch, and try changing some of the properties of the groups.

1.6 Velocity

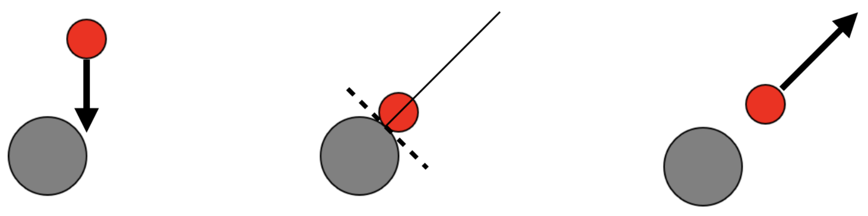
Velocity is a way to describe how an object in a physics system is moving. We used it a bit earlier to make one of the balls move, before we added gravity. Every frame (every time the code runs the update loop) objects will move a distance and in a direction based on their current velocity. We can change an object's velocity by setting its `vel` component. Try changing the definition of `bounce()` to this:

```
function ballHitObstacle(ball, obs){
  ball.vel.y -= 5
  ball.vel.x += 2
}
```

Try running this. You'll see the balls now bounce quite aggressively upwards, and a little to the right! This is cool, but the physics aren't quite right – they'll bounce upwards even if they hit the side of an obstacle. Let's fix this.

1.7 Task 2: Bounce Trajectories

For this task, I want you to rewrite `bounce()` so that the ball receives a burst of velocity *directly away* from the obstacle. Like so:



To do this, you'll need to set the `x` and `y` velocity of the ball when it bounces, just like before. But instead of using fixed values, the values will depend on the angle at which the two objects

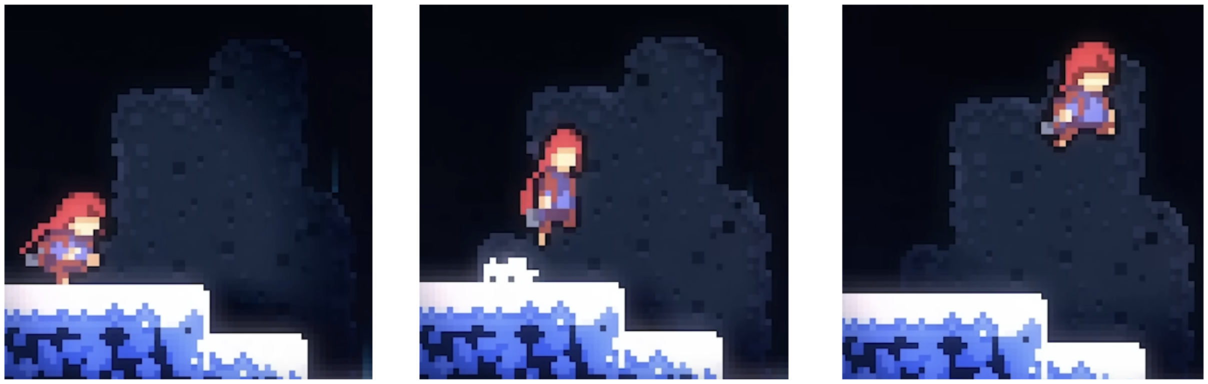


Figure 4: Three screenshots from the platformer *Celeste*, showing the main character jumping. Notice how in the second image the character has been squished horizontally, and stretched vertically, to accentuate the jump. There's also a dust cloud where she jumped from.

collide. We want the overall velocity of the ball to be set to **5**, but you'll need to figure out a way to decompose this into the x and y components of its velocity.

To help, there is a function called `angleTo` that will give you the angle of a line drawn between two objects. For example, `obs.angleTo(ball)` will give you the angle of the normal line shown in the image above. You'll also need to use some trigonometric functions!

Remember – if you need a hand, just ask me.

2 Juice

Our physics demo now feels quite good - things bounce off each other and ping all around the place. If you want to see my version, or just want to skip to this section of the task, you can click here (I added in randomly placed obstacles to make it feel a bit more dynamic).

As a final topic for today, we're going to look at a concept in game design known as *juice*. Juice is the name given to a range of small effects, behaviours and interactions that make a game feel alive, responsive, reactive - *juicy*. I first came across the term in a talk called 'Juice It Or Lose It' by Martin Jonasson and Petri Purho, which you can still watch online here.

Juice includes things such as:

- Particle effects: like dust kicking into the air when a character starts running, or sparks flying when a sword hits metal.
- Smearing: changing the shape of something to accentuate movement or impact.
- Easing: modulating movement with mathematical functions so that it is nonlinear - things feel slippy and bouncy.
- Frame holds: pausing the game for a single frame (or two) to accentuate an event.

In figure 4 you can see a few examples of this from Celeste, which has incredibly good juice and general 'game feel'. We encountered this last week because it was originally prototyped in PICO-8! It's an incredible game.

So let's add some juice! Click here to get an improved version of our demo from earlier, with some fresh art and sounds!

2.1 Task 3.1: Squishiness

A very simply way to add juice is to make objects react to things by changing shape – squishing, stretching, wobbling or bouncing. So let's add some juice to the obstacles when they're hit. We want them to be made smaller when they get hit, and then return to normal over the course of a few frames. We can change the size of a sprite by adjusting its `scale` property:

```
obs.scale = 1.0
```

First, go back to the function where we make the ball bounce, and add a new line that changes the obstacle's scale property to 80-90% of normal.

Second, we now want the size of the obstacle to go back to normal, but we want it to happen slowly. To do this we're going to use *linear interpolation* or 'lerp'. Lerps are a way to smoothly change a variable from one value to another. A lerp takes three arguments:

```
myVariable = lerp(min, max, amount)
```

It returns a value that is `amount`% of the way between `min` and `max`. So `lerp(10, 20, 0.5)` will return a number that is 50% of the way between 10 and 20, i.e. 15. If we do this repeatedly by a small amount, we get a very nice and smooth change happening. Try it out – in the `draw()` method I've put a for loop that loops over all the obstacles. Write a `lerp` call that changes the `scale` of an obstacle from whatever it currently is (minimum) to 1.0 (maximum) by a small amount - I recommend something like `0.2`. To access an obstacle in the for loop, just use `obstacles[i]`.

Here's my solution if you want to see it.

2.2 Task 3.2: Frame Lock

Games sometimes pause for a few frames when something significant happens to emphasise the moment. We can do that too! We'll use a variable to track how many frames to pause for.

First, create a new global variable called `freezeFrames`, and in our collision function set it to something like 5. Then, we want to pause all game logic and systems. We can do this by setting something called `world.timeScale` to zero. This basically sets the game into slow motion, but still keeps the program executing. Great, our game now freezes! Now we need to unfreeze it...

Second, in the `draw()` function, if `freezeFrames` is bigger than zero, reduce it by one. Then, if it is reduced to zero, set `world.timeScale` back to 1. This means that each frame we tick down, and when we hit zero again the game resumes. Five frames is more than enough to see the difference. Hit play and see what happens!

Again, here's my solution for this bit. Getting juicier!

2.3 Task 3.3: Particles!

Finally, let's add some sparkles when we hit an obstacle. Particle systems come in all shapes and sizes and are used *everywhere* in games, even when things don't look like particles (like falling leaves or sometimes even trees!) In p5.play we don't have dedicated particle systems, so instead we'll use a regular group of sprites. I've already loaded in an image for you to use, called `particle.png`.

First, create a new Group in the `setup()` function, just like the balls and obstacles. Set the `image` property of the group to be the filename of the particle image. We want these particles to be less affected by gravity so they float a little – set the `gravityScale` property to something smaller than one, I recommend 0.3.

Second, we want this group to not collide with other game objects. To do this, we need to tell p5.play that it *overlaps* with them instead. Add code similar to the following at the end of your `setup()` function (assuming your group is called `particles`:

```
particles.overlap(balls)
particles.overlap(obstacles)
particles.overlap(particles)
```

Third, we now want to spawn a bunch of particles whenever a bouncer gets hit. In the collision function, create a for loop that runs at least five times, and create a new sprite in the particles group you made. We also want each particle to move in a slightly random direction, so set it's `vel.x` and `vel.y` to have random values.

Check your code to see if it runs! You can find my final version here. I've added a few bonus tweaks to the particles to make them a bit fancier.

Those are the basics of physics and juice in games! Well done! If you've finished earlier, play around and see what else you can do with this demo. Suggestions:

- Create points scoring for each bouncer hit, or some kind of target system.
- Let me place bouncers with right-click instead of creating them randomly, make it a physics toy.
- Have buckets at the bottom with different points, like a pachinko game.

3 What Next

Thanks for taking part in the creative coding course! I hope you found at least one tool that you really love in this series, and you take it forward and keep making things with it. Making stupid things for no particular reason is one of life's greatest pleasures, and it can lead to all sorts of unexpected surprises, joys and sometimes even new life directions!

Here are some things I recommend you do, whether you want to work in the arts/creative industries, or you just like making little things:

- **Register an account on itch.io** if you like making games. It is the modern home of game making on the web, it will let you share your work for free, find other people's work, and discover communities.
- **Find local communities you can join.** King's has a Game Development Society (more focused on 'hardcore' game development). There is the London Pattern Club which is interested in algorithms and crafting/art. There is London Algorhythms and Todepond which are livecoder communities. Meet people outside of King's and learn from them!

- **Find reasons to make things.** Game jams are good reasons to start and finish a small project quickly. Monthly challenges like Genuary help you create alongside other people. Or set a challenge yourself, like Saskia Freeke.
- **Encourage friends to make things too.** Nothing is more encouraging than making and learning alongside other people, and collaborating on something together is extremely rewarding.
- **Make it yourself.** I know lots of people like using generative AI these days. If you use it for other parts of your life, I would say: avoid using it for some stuff. At the very least, making little bits of art or music by yourself is very meaningful. Doing stuff by hand teaches you a lot, and lets your brain get into it.
- **Enjoy what you do.** Don't give up at the first sign of trouble, but also don't feel you need to chase a goal just because other people say you should. You don't need to make stuff commercially. You don't need to be good. You don't need to make games. You don't need to make digital things. Find what you enjoy, and stick to it.
- **Keep in touch.** Creative communities are important, and the good ones persist forever. Please let me know where you end up!

It's been a great pleasure working with you through this course! I hope wherever you go next is rewarding and fun, and I'll perhaps see some of you in a future course soon...