Week 26

Lets take our game to the next level: Ready Player One

Create a new variable called player and add the following code to the create function.

```
player = this.physics.add.sprite(100, 450, 'dude');
player.setBounce(0.2);
player.setCollideWorldBounds(true);


this.anims.create({
    key: 'left',
    frames: this.anims.generateFrameNumbers('dude',
{ start: 0, end: 3 }),
    frameRate: 10,
    repeat: -1
});


this.anims.create({
    key: 'turn',
```

```
    frames: [ { key: 'dude', frame: 4 } ],
    frameRate: 20
});

this.anims.create({
    key: 'right',
    frames: this.anims.generateFrameNumbers('dude',
{ start: 5, end: 8 }),
    frameRate: 10,
    repeat: -1
});
```

There are two separate things going on here: the creation of a Physics Sprite and the creation of some animations that it can use.

Physics Sprite

The first part of the code creates the sprite:

```
player = this.physics.add.sprite(100, 450, 'dude');
player.setBounce(0.2);
player.setCollideWorldBounds(true);
```

This creates a new sprite called player, positioned at 100 x 450 pixels from the bottom of the game. The sprite was created via the Physics Game Object Factory (this.physics.add) which means it has a Dynamic Physics body by default.

After creating the sprite it is given a slight bounce value of 0.2. This means when it lands after jumping it will bounce ever so slightly. The sprite is then set to collide with the world bounds. The bounds, by default, are on the outside of the game dimensions. As we set the game to be 800 x 600 then the player won't be able to run outside of this area. It will stop the player from being able to run off the edges of the screen or jump through the top.

Animations

If you glance back to the preload function you'll see that 'dude' was loaded as a sprite sheet, not an image. That is because it contains animation frames. This is what the full sprite sheet looks like:

There are 9 frames in total, 4 for running left, 1 for facing the camera and 4 for running right. Note: Phaser supports flipping sprites to save on animation frames, but for the sake of this tutorial we'll keep it old school.

We define two animations called 'left' and 'right'. Here is the left animation:

```
this.anims.create({
    key: 'left',
    frames: this.anims.generateFrameNumbers('dude', { start: 0, end: 3 }),
    frameRate: 10,
    repeat: -1
});
```

The 'left' animation uses frames 0, 1, 2 and 3 and runs at 10 frames per second. The 'repeat -1' value tells the animation to loop.

This is our standard run-cycle and we repeat it for running in the opposite direction, using the key 'right' and a final one for 'turn'.

Extra Info: In Phaser 3 the Animation Manager is a global system. Animations created within it are globally available to all Game Objects. They share the base animation data while managing their own timelines.

This allows you to define a single animation once and apply it to as many Game Objects as you require. This is different to Phaser 2 where animations belonged specifically to the Game Objects they were created on.

Phaser has support for a variety of different physics systems, each acting as a plugin available to any Phaser Scene. At the time of writing it ships with Arcade Physics, Impact Physics and Matter.js Physics. For the sake of this tutorial we will be using the Arcade Physics system for our game, which is simple and light-weight, perfect for mobile browsers.

When a Physics Sprite is created it is given a body property, which is a reference to its Arcade Physics Body. This represents the sprite as a physical body in Phasers Arcade Physics engine. The body object has a lot of properties and methods that we can play with.

For example, to simulate the effects of gravity on a sprite, it's as simple as writing:

```
player.body.setGravityY(300)
```

This is an arbitrary value, but logically, the higher the value, the heavier your object feels and the quicker it falls.

The reason for this is that we're not yet testing for collision between the ground and the player.

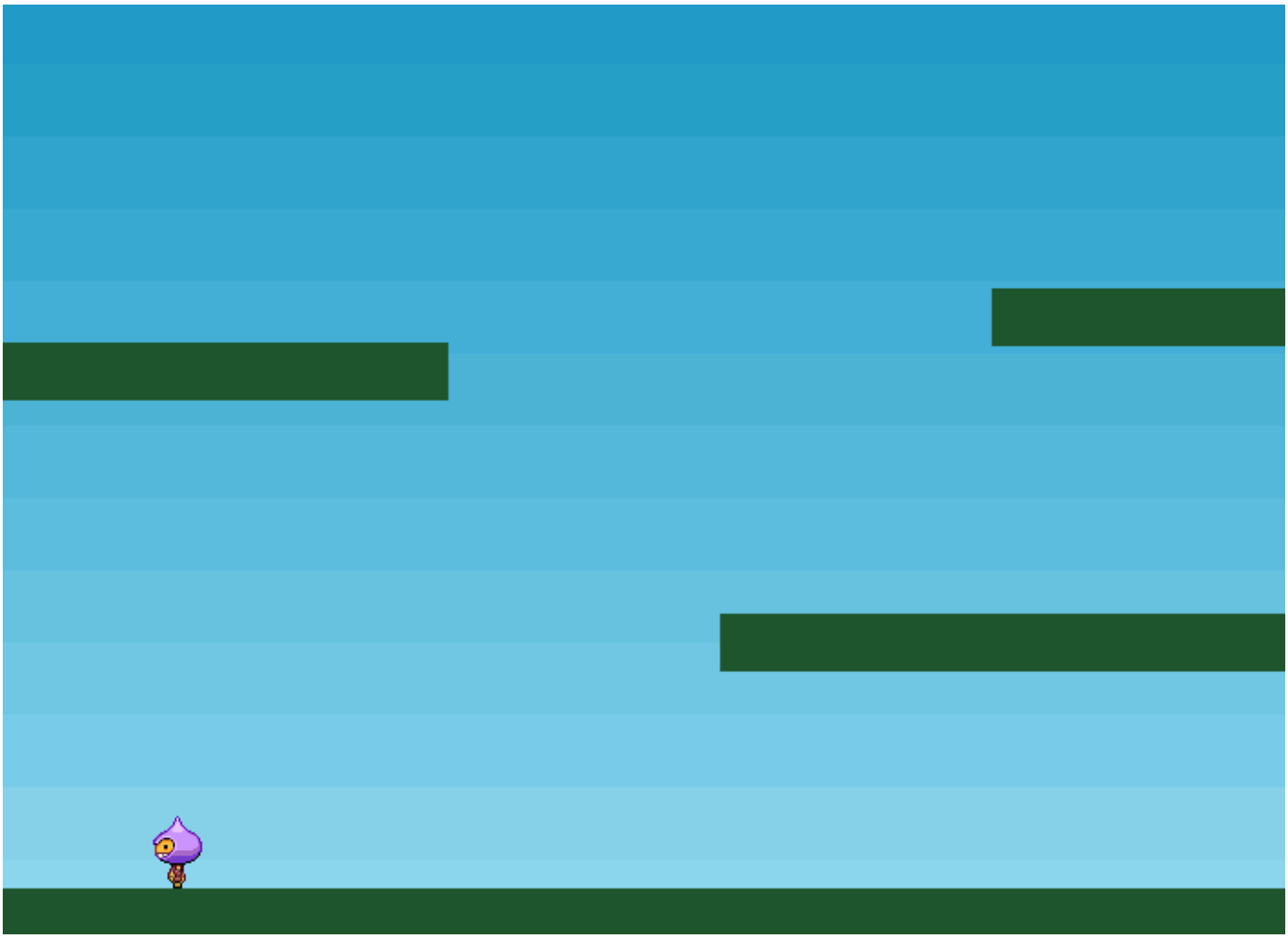
We already told Phaser that our ground and platforms would be static bodies. Had we not done that, and created dynamic ones instead, then when the player collided with them it would stop for a moment and then everything would have collapsed. This is because unless told otherwise, the ground sprite is a moving physical object and when the player hits it, the resulting force of the collision is applied to the ground, therefore, the two bodies exchange their velocities and ground starts falling as well.

In order to allow the player to collide with the platforms we can create a Collider object. This object monitors two physics objects (which can include Groups) and checks for collisions or overlap between them. If that occurs it can then optionally invoke your own callback, but for the sake of just colliding with platforms we don't require that:

that:

```
this.physics.add.collider(player, platforms);
```

The Collider is the one that performs the magic. It takes two objects and tests for collision and performs separation against them. In this case we're giving it the player sprite and the platforms Group. It's clever enough to run collision against all Group members, so this one call will collide against the ground and all platforms. The result is a firm platform that doesn't collapse:



```
function create ()
    {
        this.add.image(400, 300, 'sky');

        platforms = this.physics.add.staticGroup();

        platforms.create(400, 568, 'ground').setScale(2).refreshBody();

        platforms.create(600, 400, 'ground');
        platforms.create(50, 250, 'ground');
        platforms.create(750, 220, 'ground');

        player = this.physics.add.sprite(100, 450, 'dude');

        player.setBounce(0.2);
        player.setCollideWorldBounds(true);

        this.anims.create({
            key: 'left',
            frames: this.anims.generateFrameNumbers('dude', { start: 0, end: 3 }),
            frameRate: 10,
            repeat: -1
        });

        this.anims.create({
            key: 'turn',
            frames: [ { key: 'dude', frame: 4 } ],
            frameRate: 20
        });
```

```
    this.anims.create({
        key: 'right',
        frames: this.anims.generateFrameNumbers('dude', { start: 5, end: 8 }),
        frameRate: 10,
        repeat: -1
    });

    this.physics.add.collider(player, platforms);
    cursors = this.input.keyboard.createCursorKeys();
}
```

ADD This line as the last line in the function create ()

Lets look at the function update();