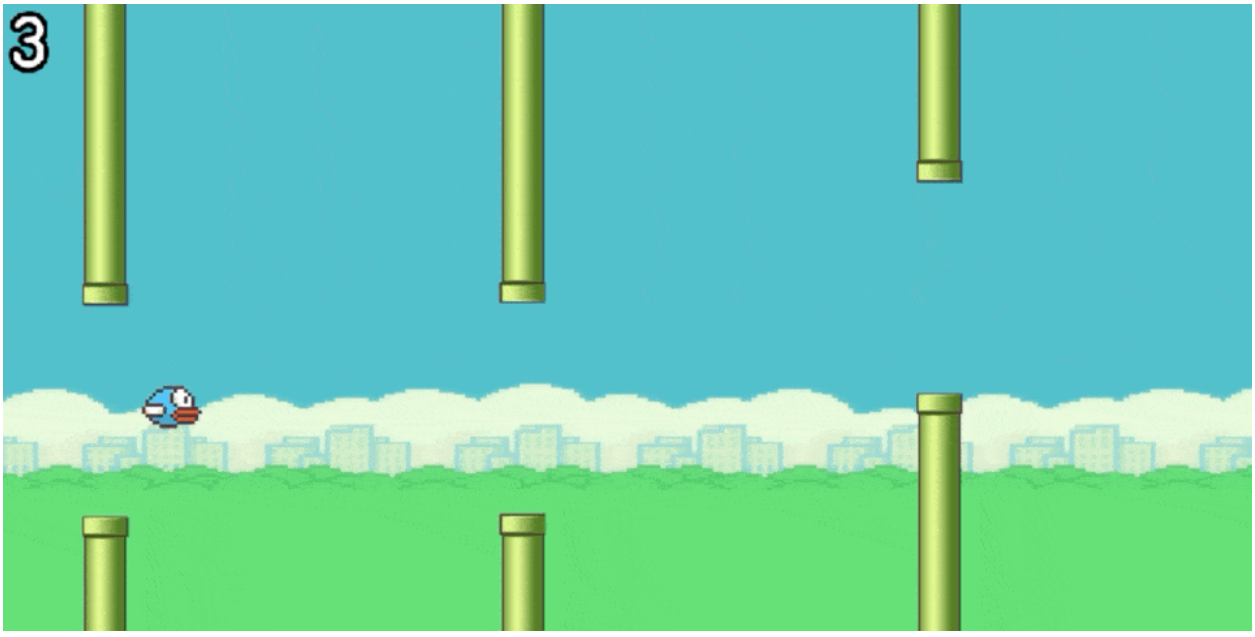
Building Flappy Bird with Kaboom.js

Flappy Bird was a smash hit game for mobile phones back in 2013-2014. The inspiration behind the app was the challenge of bouncing a ping pong ball on a paddle for as long as possible without letting it drop to the ground or shoot off into the air. At the peak of its success, the game creator unexpectedly removed it from all app stores, saying that he felt guilty that the game had become addictive for many people. In the wake of the removal, many clones were made to fill the gap left by the original Flappy Bird. After a few months, the original author released new versions of the game.

Let's take a trip back to 2014 and create our own clone of Flappy Bird using Kaboom! By remaking a game, you can not only learn how to make games, but also extend and change the game in any way you like.



The finished game

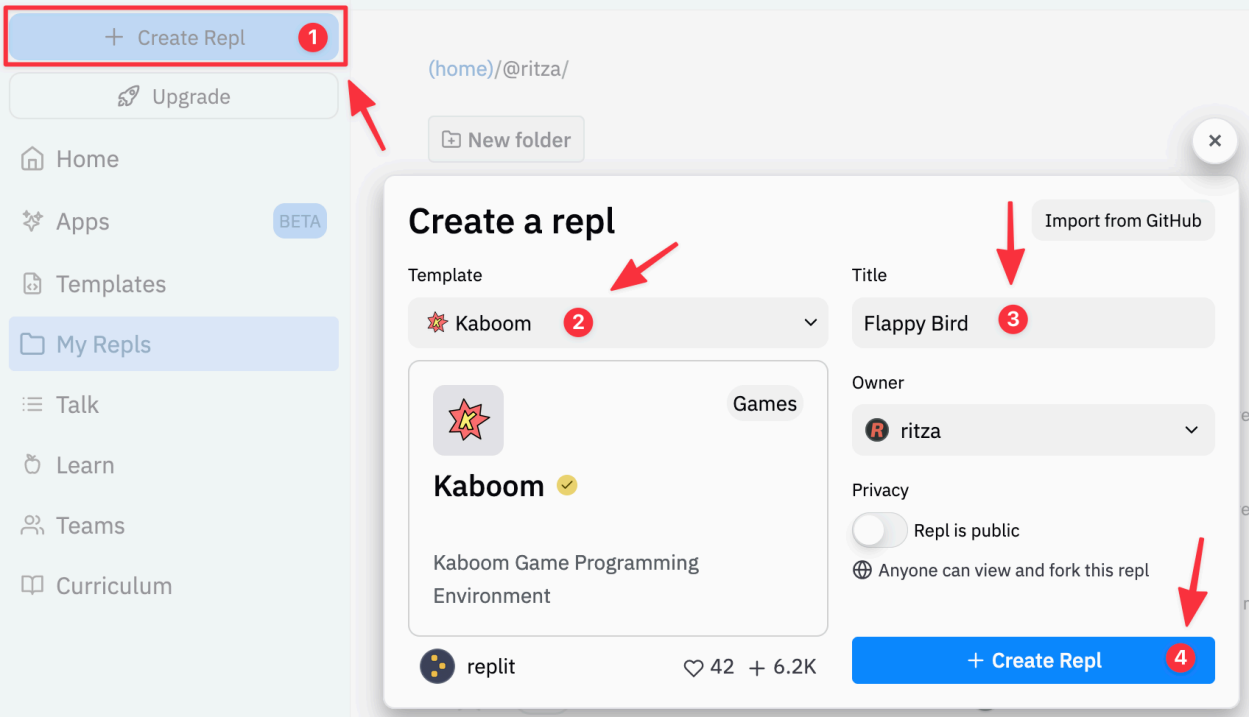
Click to open gif[1]

This article is based on this video tutorial[2], with a few small differences. Mainly, the Flappy assets (graphics and sound) are no longer available by default in the Replit Kaboom asset library, but that's OK because we've included them as a download here[3], so you can still use them.

[1] https://docs.replit.com/images/tutorials/35-flappy-bird/game-play.gif
[2] https://www.youtube.com/watch?v=hgReGsh5xVU
[3] /tutorial-files/flappy-bird-kaboom/flappy-assets.zip

Creating a new project in Replit

Head over to Replit[4] and create a new repl. Choose **Kaboom** as your project type. Give this repl a name, like "Flappy!".

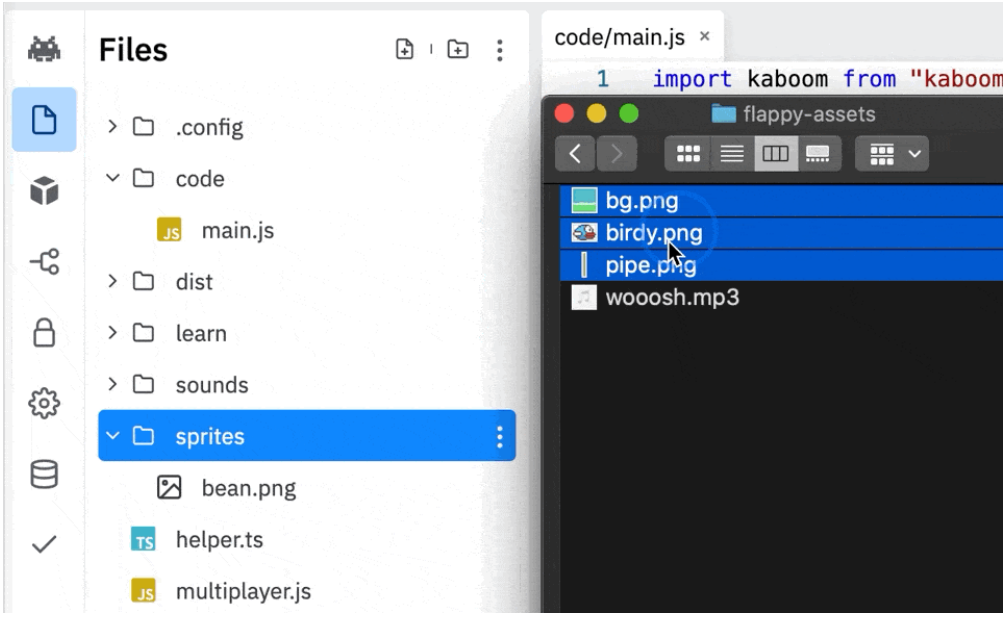


Creating a new repl

After the repl has booted up, you should see a `main.js` file under the "Code" section. This is where we'll start coding. There is already some code in this file, but we'll replace that.

Download the sprites and asset files[5] we need for the game, and unzip them on your computer. In the Kaboom editor, click the "Files" icon in the sidebar. Now drag and drop all the sprites (image files) into the "sprites" folder, and all the sounds (MP3 files) into the "sounds" folder. Once they have uploaded, you can click on the "Kaboom" icon in the sidebar, and return to the "main" code file.

[4] https://replit.com
[5] https://docs.replit.com/tutorial-files/flappy-bird-kaboom/flappy-assets.zip



Upload sprites

Click to open gif[6]

Initializing Kaboom

In the "main" code file, delete all the example code. Now we can add reference to Kaboom, and initialize it:

```
import kaboom from "kaboom";

kaboom();
```

Let's import the game assets (graphics and sound). We can use Kaboom's `loadSprite`[7] and `loadSound`[8] functions:

```
loadSprite("birdy", "sprites/birdy.png");
loadSprite("bg", "sprites/bg.png");
loadSprite("pipe", "sprites/pipe.png");
loadSound("wooosh", "sounds/wooosh.mp3");
```

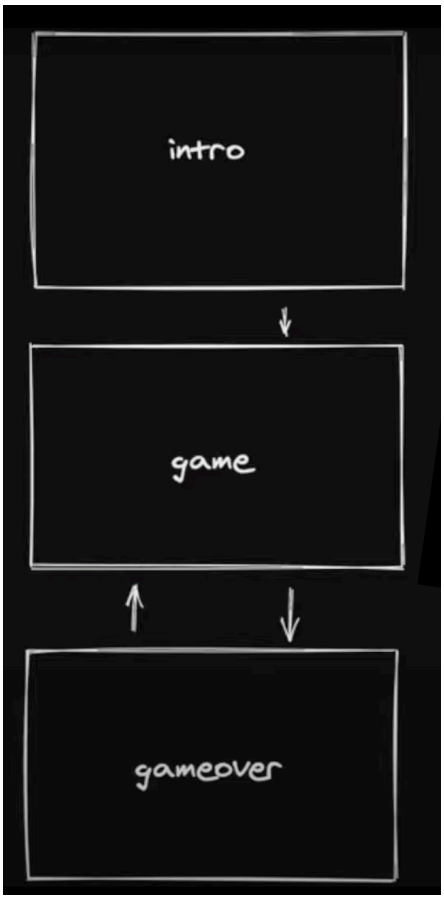
The first argument in each `load` function is the name we want to use to refer to the asset later on in our code. The second parameter is the location of the asset to load.

[6] https://docs.replit.com/images/tutorials/35-flappy-bird/upload-sprites.gif
[7] https://kaboomjs.com/#loadSprite
[8] https://kaboomjs.com/#loadSound

Adding scenes

Scenes[9] are like different stages in a Kaboom game. There are generally three scenes in games:

- The intro scene, which gives some info and instructions, and waits for the player to press “start”.
- The main game, where we play.
- An endgame, or game over scene, which gives the player their score or overall result, and allows them to start again.



Game scenes

For this tutorial, we’ll omit the intro scene, since we already know what Flappy bird is and how to play it, but you can add your own intro scene later!

Let’s add the code for defining each scene:

[9] https://kaboomjs.com/#scene

```
scene("game", () => {

        // todo.. add scene code here
});


scene("gameover", (score) => {

        // todo.. add scene code here
});


go("game")
```

Notice in the `gameover` scene definition, we add a custom parameter, `score`. This is so that we can pass the player's final score to the end game scene to display it.

To start the whole game off, we use the `go`[10] function, which switches between scenes.

Building the game world

Now that we have the main structure and overhead functions out of the way, let's start adding in the characters that make up the Flappy world. In Kaboom, characters are anything that makes up the game world, including floor, platforms, etc., and not only the players and bots. They are also known as "game objects".

We'll start with the background, using the `bg.png` image we added earlier. Add this code to the `game` scene section:

```
add([
        sprite("bg", {width: width(), height: height()})
]);
```

Here we use the `add`[11] function to add a new character to the scene. The `add` function takes an array of components that we can use to give each game character special properties. In Kaboom, every character is made up of one or more components. There are built-in components for many properties, like `sprite`[12], which gives the character an avatar; `body`[13], which makes the character respond to gravity; and `solid`[14], which makes the character solid, so other characters can't move through it.

[10] https://kaboomjs.com/#go
[11] https://kaboomjs.com/#add
[12] https://kaboomjs.com/#sprite
[13] https://kaboomjs.com/#body
[14] https://kaboomjs.com/#solid

Since the background doesn't need to do much, just stay in the back and look pretty, we only use the `sprite`[15] component, which displays an image. The `sprite` component takes the name of the sprite, which we set when we loaded the sprite earlier, and optionally, the width and height that it should be displayed at on the screen. Since we want the background to cover the whole screen, we need to set the `width` and `height` of the sprite to the width and height of the window our game is running in. Kaboom provides the `width()`[16] and `height()`[17] functions to get the window dimensions.

If you press the "Run" button at the top of your repl now, you should see the background of the Flappy world come up in the output section of the repl:

Flappy background with buildings, trees and building sky line

Great! Now let's add in the Flappy Bird. Add this code to the `game` scene:

```
const player = add([
        // list of components
        sprite("birdy"),
        scale(2),
        pos(80, 40),
        area(),
        body(),
]);
```

We use the same `add`[18] function we used for adding the background. This time, we grab a reference, `const player`, to the returned game object. This is so we can use this reference later when checking for collisions, or flapping up when the player taps the space bar.

[15] https://kaboomjs.com/#sprite
[16] https://kaboomjs.com/#width
[17] https://kaboomjs.com/#height
[18] https://kaboomjs.com/#add

You'll also notice that the character we are adding here has many more components than just the `sprite`[19] component we used for the background. We already know what the `sprite` component does, here is what the rest are for:

- The `scale`[20] component makes the sprite larger on screen by drawing it at `2` times the sprite's normal image size. This gives a nice pixelated look, while also making it easier to spot the bird.
- The `pos`[21] component sets the position on the screen that the character should initially be at. It takes X and Y coordinates to specify a position.
- The `area`[22] component gives the sprite an invisible bounding box around it, which is used when calculating and detecting collisions between characters. We'll need this so that we can detect if Flappy flies into the pipes.
- The `body`[23] component makes the character subject to gravity. This means Flappy will fall out of the sky if the player doesn't do anything.

Press `command + s` (Mac) or `control + s` (Windows/Linux) to update the game output window. You should see Flappy added and fall out of the sky very quickly:

Flappy falling out of the sky

Click to open gif[24]

[19] https://kaboomjs.com/#sprite
[20] https://kaboomjs.com/#scale
[21] https://kaboomjs.com/#pos
[22] https://kaboomjs.com/#area
[23] https://kaboomjs.com/#body
[24] https://docs.replit.com/images/tutorials/35-flappy-bird/flappy-falls.gif

Making Flappy fly

Our next task is to save Flappy from plummeting to their death by giving control to the player to flap Flappy's wings. We'll use the spacebar for this. Kaboom has an `onKeyPress`[25] function, which fires a callback with custom code when the specified key is pressed. Add this code to the `game` scene to make Flappy fly when the `space` key is pressed:

```
onKeyPress("space", () => {
        play("wooosh");
        player.jump(400);
});
```

In the callback handler, we first `play`[26] a sound of flapping wings to give the player feedback and add some interest to the game. Then we use the `jump`[27] method, which is added to our player character through the `body`[28] component we added earlier. The `jump` function makes a character accelerate up sharply. We can adjust just how sharp and high the jump should be through the number we pass as an argument to the jump method – the larger the number, the higher the jump. Although Flappy is technically not jumping (you normally need to be on a solid surface to jump), it still has the effect we need.

Update the game output window, and if you press the spacebar now, you'll be able to keep Flappy in the air! Remember to quickly click in the output window as the game starts, so that it gains focus and can detect player input such as pressing the `space` key.

[25] https://kaboomjs.com/#onKeyPress
[26] https://kaboomjs.com/#play
[27] https://kaboomjs.com/#body
[28] https://kaboomjs.com/#body

Flying-flappy

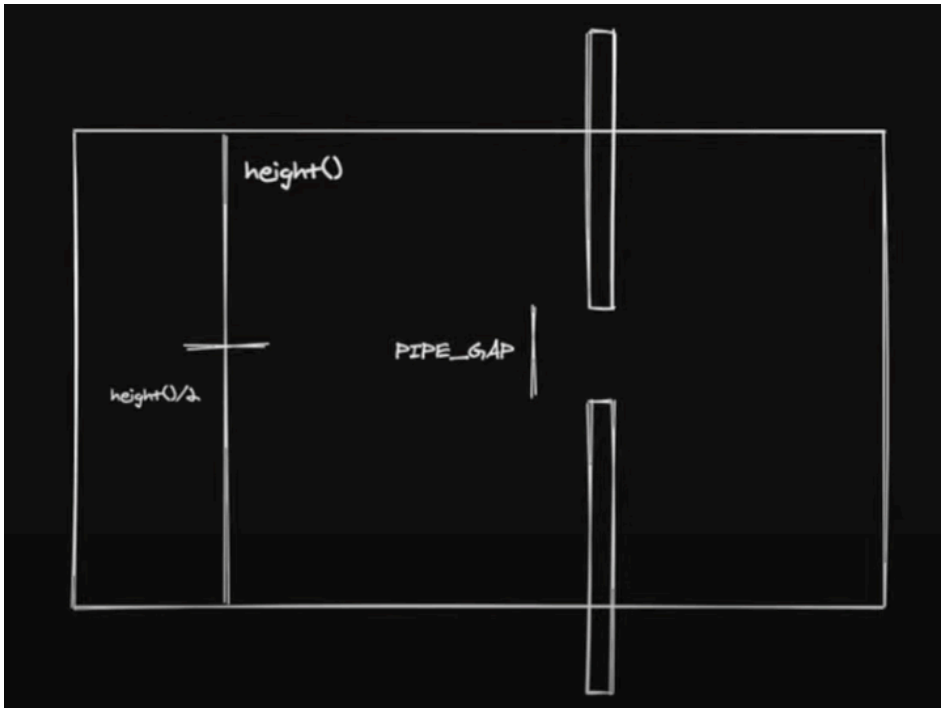
Click to open gif[29]

Adding in the pipes

Now we can get to the main part of the game – adding in the moving pipes that Flappy needs to fly through.

Here is a diagram of the layout of the pipes in the game.

[29] https://docs.replit.com/images/tutorials/35-flappy-bird/flappy-fly.gif



Pipe gap

We want to move the pipe gap, and therefore the pipes, up and down for each new pipe pair that is created. This is so we don't have the gap at the center point of the screen constantly – we want it to be slightly different for each pipe pair that comes along. We do want to keep the gap size consistent though.

Let's start by having the pipe gap in the center of the screen. We'll give the pipe gap a size `PIPE_GAP`. Then to place the pipes, the bottom of the upper pipe should be `PIPE_GAP/2` pixels above the center point of the window, which is `height()/2`. Likewise, the top of the lower pipe should be `PIPE_GAP/2` pixels below the center point of the window, again which is `height()/2`.

This way, we place the pipe so that the pipe gap is in the center of the window. Now we want to randomly move this up or down for each new pair of pipes that comes along. One way to do this is to create a random offset, which we can add to the midpoint to effectively move the midpoint of the window up or down. We can use the Kaboom `rand`[30] function to do this. The `rand`[31] function has two parameters to specify the range in which the random number should be.

Let's put that all together. The Y-position of the lower pipe can be calculated as:

```
height()/2 + offset + PIPE_GAP/2
```

Remember, the top of the window is `y=0`, and the bottom is `y=height()`. In other words, the lower down on the screen a position is, the higher its `y` coordinate will be.

For the upper pipe, we can calculate the point where the bottom of the pipe should be like this:

[30] https://kaboomjs.com/#rand
[31] https://kaboomjs.com/#rand

```
height()/2 + offset - PIPE_GAP/2
```

Kaboom has an `origin`[32] component that sets the point a character uses as its origin. This is `topleft` by default, which works well for our lower pipe, as our calculations above are calculating for that point. However, for the upper pipe, our calculations are for the bottom of the pipe. Therefore, we can use the `origin`[33] component to specify that.

Since we want the pipes to come from the right of the screen toward the left, where Flappy is, we'll set their X-position to be the `width()`[34] of the screen.

To identify and query the pipes later, we add the text tag `"pipe"` to them.

Finally, since we need to create many pipes during the game, let's wrap all the pipe code in a function that we will be able to call at regular intervals to make the pipes.

Here is the code from all those considerations and calculations. Insert this code to the `game` scene:

```
const PIPE_GAP = 120;

function producePipes(){
        const offset = rand(-50, 50);

        add([
          sprite("pipe"),
          pos(width(), height()/2 + offset + PIPE_GAP/2),
          "pipe",
          area(),
        ]);

        add([
          sprite("pipe", {flipY: true}),
          pos(width(), height()/2 + offset - PIPE_GAP/2),
          origin("botleft"),
          "pipe",
          area()
        ]);
}
```

Now we need to do a few more things to make the pipes appear and move.

To move the pipes across the screen, we can use the `onUpdate`[35] function to update all pipes' positions with each frame. Note that we only need to adjust the `x` position of the pipe. Add this code to the `game` scene part of your code:

[32] https://kaboomjs.com/#origin
[33] https://kaboomjs.com/#origin
[34] https://kaboomjs.com/#width
[35] https://kaboomjs.com/#onUpdate

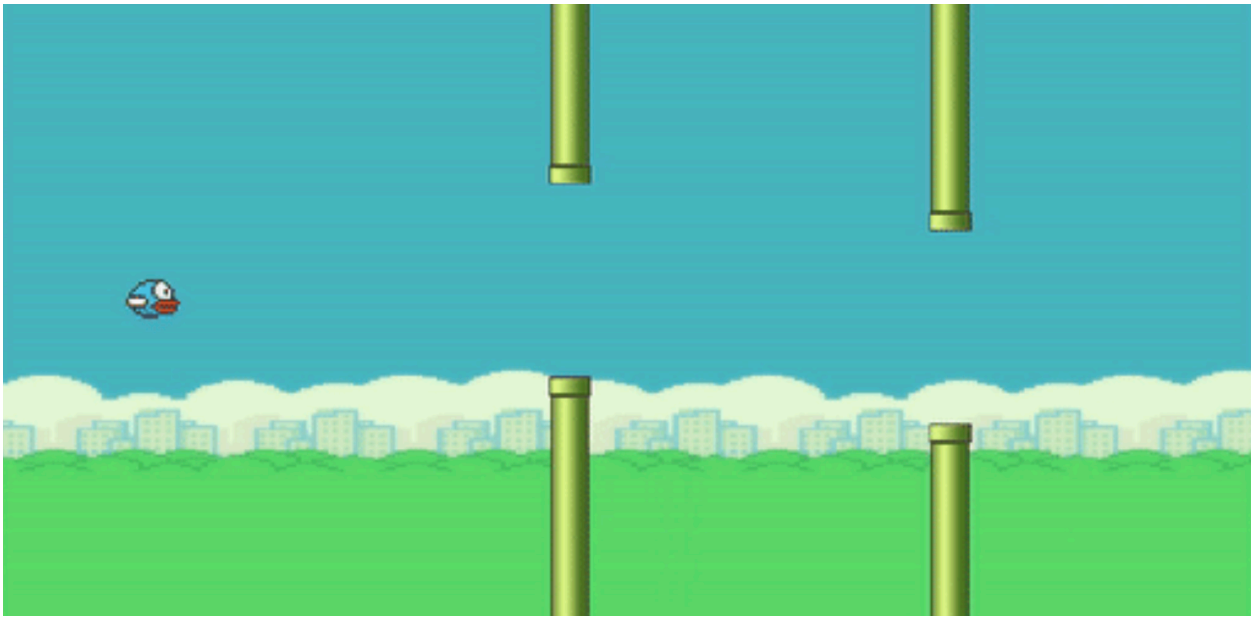
```
onUpdate("pipe", (pipe) => {
        pipe.move(-160, 0);
});
```

Next we'll generate pipes at a steady rate. We can use the `loop`[36] function for this. Add the following to the `game` scene part of the code:

```
loop(1.5, () => {
        producePipes();
});
```

This calls our `producePipes()` function every `1.5` seconds. You can adjust this rate, or make it variable to increase the rate as the game progresses.

Update the game output window now and you should see the pipes being generated and moving across the screen. You can also fly Flappy, although crashing into the pipes does nothing for now.



Moving pipes

Click to open gif[37]

Flappy is flapping and the pipes are piling on. The next task is to detect when Flappy flies past a pipe, increasing the player's score.

[36] https://kaboomjs.com/#loop
[37] https://docs.replit.com/images/tutorials/35-flappy-bird/moving-pipes.gif

Adding in scoring

When Flappy flies past a pipe, the player's score is incremented. To do this, we'll need to keep track of which pipes have gone past Flappy. Let's modify the pipe-generating function `producePipes` to add a custom property called `passed` to the pipes. It should look like this now:

```
function producePipes() {
        const offset = rand(-50, 50);

        add([
          sprite("pipe"),
          pos(width(), height() / 2 + offset + PIPE_GAP / 2),
          "pipe",
          area(),
          {passed: false}
        ]);

        add([
          sprite("pipe", { flipY: true }),
          pos(width(), height() / 2 + offset - PIPE_GAP / 2),
          origin("botleft"),
          "pipe",
          area(),
        ]);
}
```

Next, we'll add in a variable to track the `score`, and a text element to display it on screen. Add this code to the `game` scene:

```
let score = 0;
const scoreText = add([
        text(score, {size: 50})
]);
```

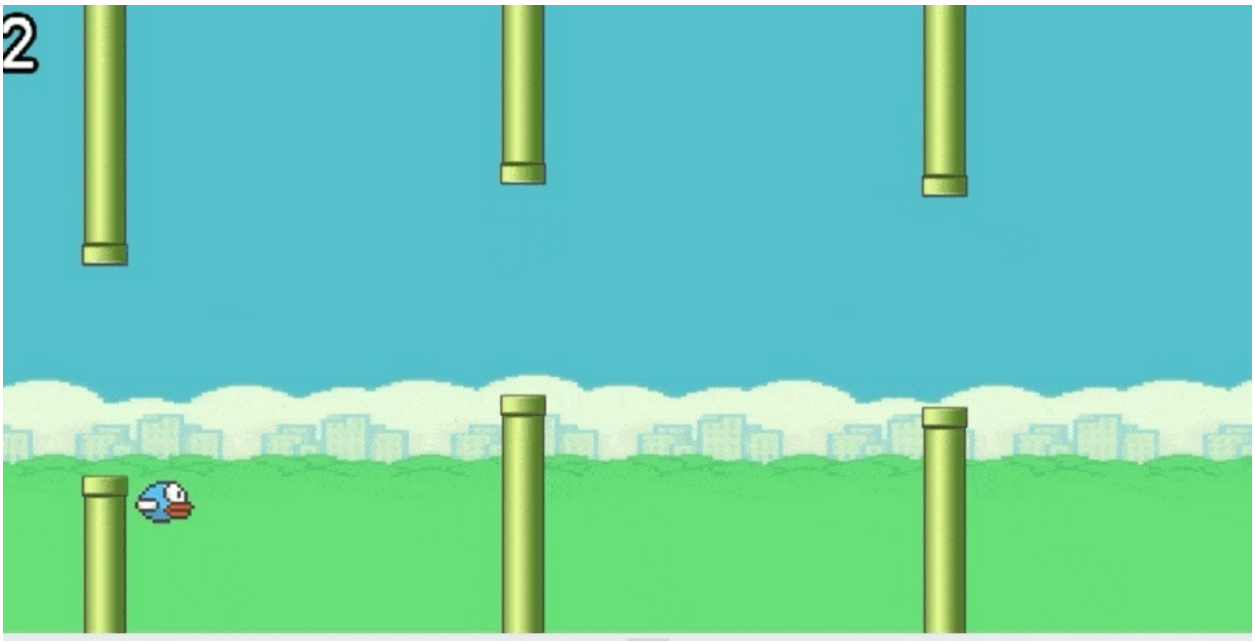
Now we can modify the `onUpdate()` event handler we created earlier for moving the pipes. We'll check if any pipes have moved past Flappy, and update their `passed` flag, so we don't count them more than once. We'll only add the `passed` flag to one of the pipes, and detect it, so as not to add a point for both the upper and lower pipe. Update the `onUpdate` handler as follows:

```
onUpdate("pipe", (pipe) => {
        pipe.move(-160, 0);

        if (pipe.passed === false && pipe.pos.x < player.pos.x) {
          pipe.passed = true;
          score += 1;
          scoreText.text = score;
        }
});
```

This checks any pipe that we haven't marked as `passed` (`passed === false`) to see if it has passed Flappy (`pipe.pos.x < player.pos.x`). If the pipe has gone past, we add `1` to the score and update the score text onscreen.

If you update the game output window now, you should see the score increase as you fly past each pipe.



Score increasing

Click to open gif[38]

Collision detection

Now that we have scoring, the last thing to do is collision detection – that is, checking if Flappy has splatted into a pipe. Kaboom has a `collides`[39] method that is added with the `area`[40] collider

[38] https://docs.replit.com/images/tutorials/35-flappy-bird/score-increase.gif
[39] https://kaboomjs.com/#onCollide
[40] https://kaboomjs.com/#area

component. We can use that to call a function when the player collides with any character with the `"pipe"` tag. Add this code to the `game` scene:

```
player.collides("pipe", () => {
        go("gameover", score);
});
```

In the collision handler, we use the `go`[41] function to switch to the `gameover` scene. We don't have anything in that scene yet, so let's update that to show a game over message and the score. We can also keep track of the high score to compare the player's latest score to. Update the `gameover` scene as follows:

```
let highScore = 0;
scene("gameover", (score) => {
  if (score > highScore) {
    highScore = score;
  }

  add([
    text(
      "gameover!\n"
      + "score: " + score
      + "\nhigh score: " + highScore,
      {size: 45}
    )
  ]);

  onKeyPress("space", () => {
    go("game");
  });
});
```

First, we create a `highScore` variable where we can track the top score across multiple game plays. Then, in our `gameover` scene, we check if the latest score passed in is bigger than the `highScore` we have recorded. If it is, the `highScore` is updated to the latest score.

To show a "game over" message, and the player's score along with the high score, we use the `add`[42] function to add a `text`[43] component to a new game object or character. We also make the font `size` large-ish for this message.

[41] https://kaboomjs.com/#go
[42] https://kaboomjs.com/#add
[43] https://kaboomjs.com/#text

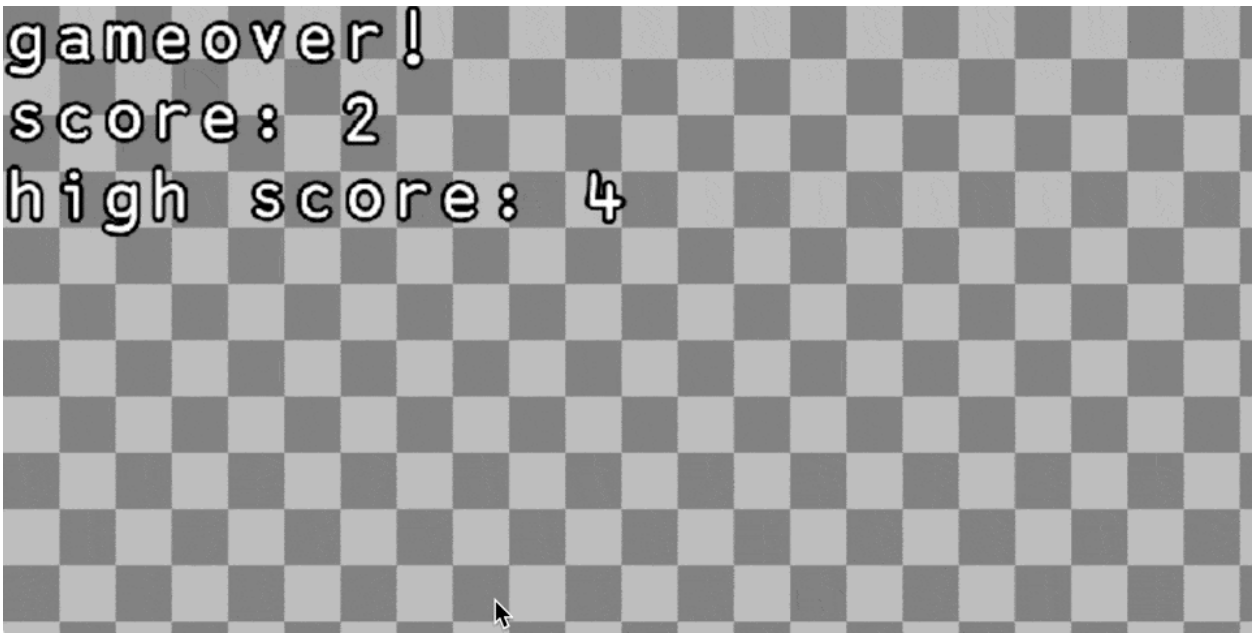
Let's include a quick way for the player to play again and try to beat their score. We use the `onKeyPress`[44] to listen for the player pressing the `space` bar. In our key-press handler, we `go`[45] back to the main `game` scene, to start the game all over again.

We also need to end the game if Flappy flies too high out of the screen, or plummets down off the screen. We can do this by adding a handler for the player's `onUpdate`[46] event, which is called each frame. Here we can check if Flappy's position is beyond the bounds of the game window. Add this code to the `game` scene:

```
player.onUpdate(() => {
        if (player.pos.y > height() + 30 || player.pos.y < -30) {
          go("gameover", score);
        }
});
```

This gives a margin of 30 pixels above or below the window, to take account of Flappy's size. If Flappy is out of these bounds, we `go`[47] to the `gameover` scene to end the game.

Update the game output window again and test it out. If you fly into a pipe now, or flap too high, or fall out of the sky, you should be taken to the game over screen:



Game over screen

Click to open gif[48]

[44] https://kaboomjs.com/#onKeyPress
[45] https://kaboomjs.com/#go
[46] https://kaboomjs.com/#add
[47] https://kaboomjs.com/#go
[48] https://docs.replit.com/images/tutorials/35-flappy-bird/game-over.gif

Next steps

Here are some ideas you can try to improve your clone of the Flappy Bird game:

- Make the game play faster as the player gets a higher score. You can do this by updating the speed that the pipes move by making the speed parameter passed to the `pipe.move` method a variable, which increases as the player score increases.
- Add some different types of obstacles, other than the pipes, for Flappy to try to avoid.
- Use the Kaboom sprite editor[49] to create your own graphics for your Flappy world!
- Add in some more sound effects and play some game music using the `play`[50] function.

Code

You can find the code for this tutorial on Replit[51]

[49] https://docs.replit.com/tutorials/kaboom-editor
[50] https://kaboomjs.com/#play
[51] https://replit.com/@ritza/Flappy-Bird