Make JavaScript Games With Replit And Kaboom

Replit

Make Javascript Games

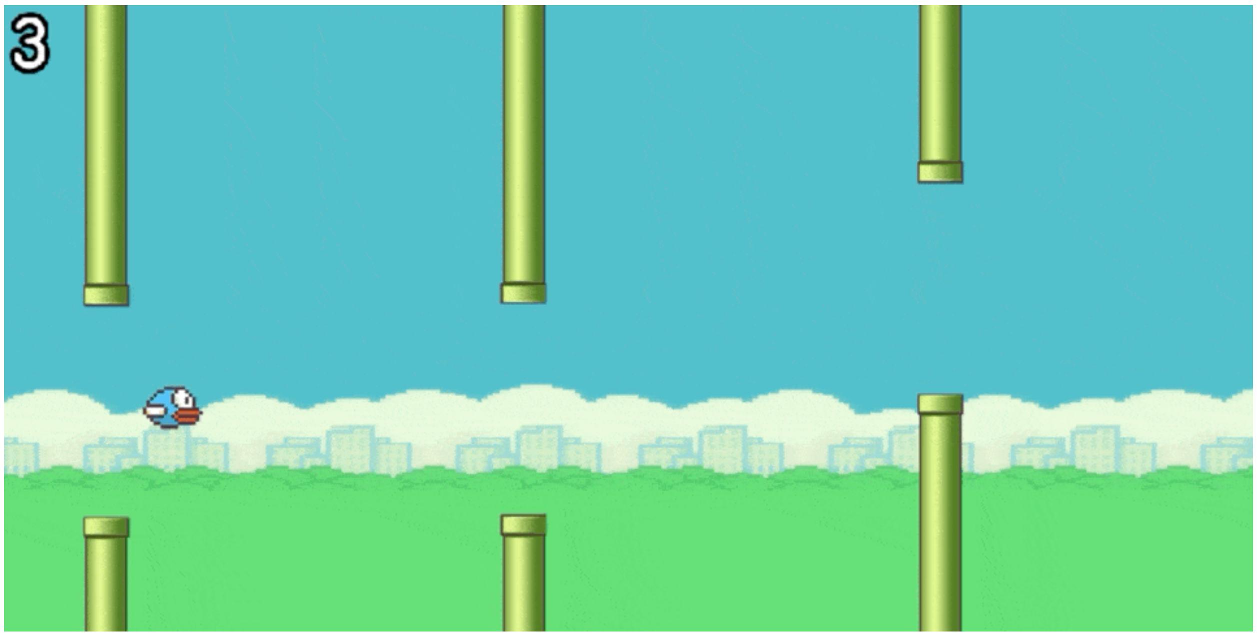
Ritza

Contents

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

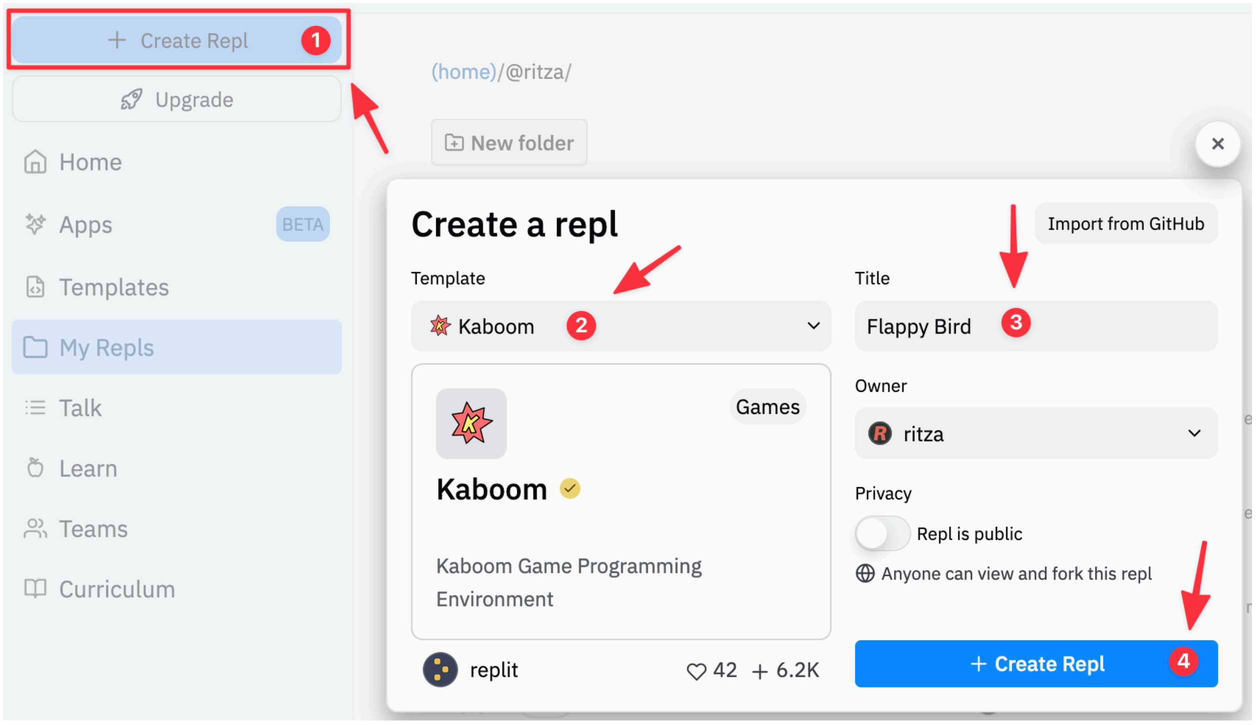
Click to open gif[1]

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!".

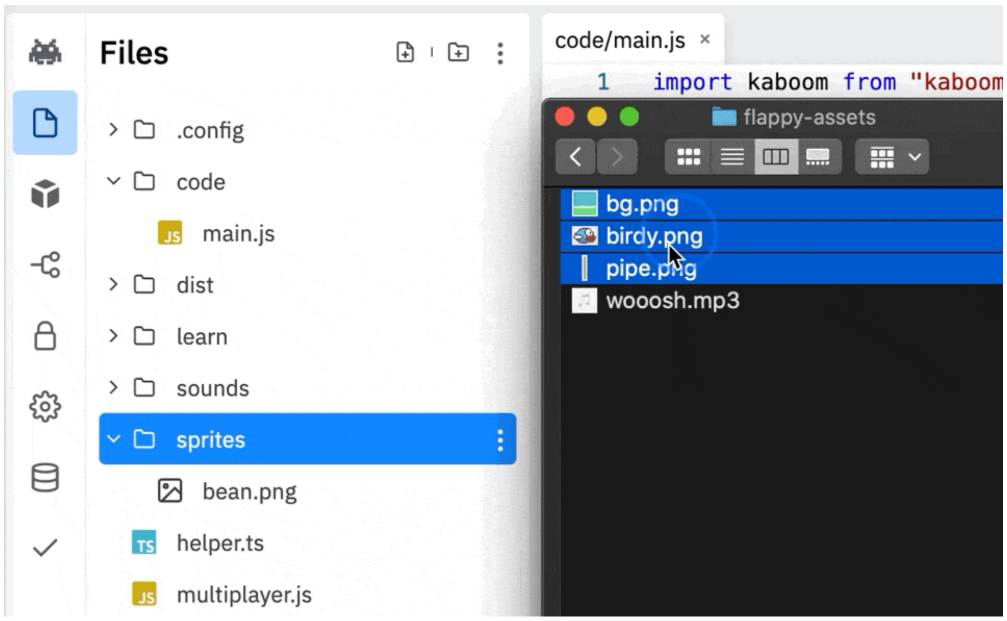


Creating a new repl

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");
```

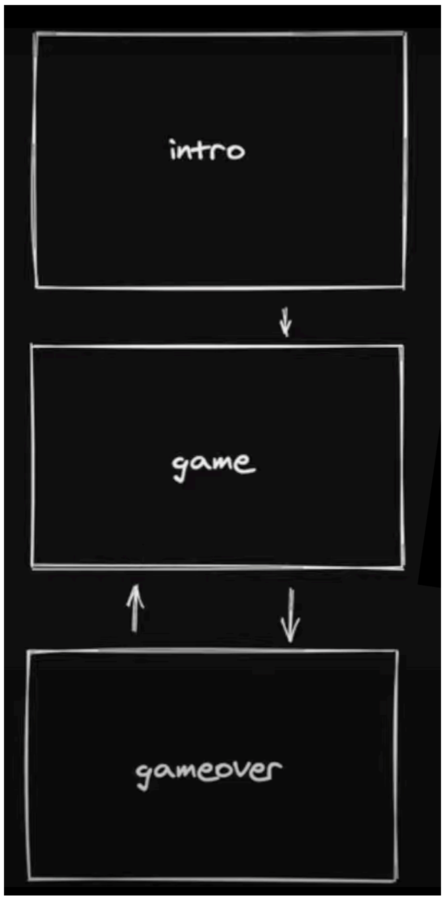
The first argument in each `load` function is the name we want to use to refer to the asset later on in our code. The second parameter is the location of the asset to load.

[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(),
]);
```

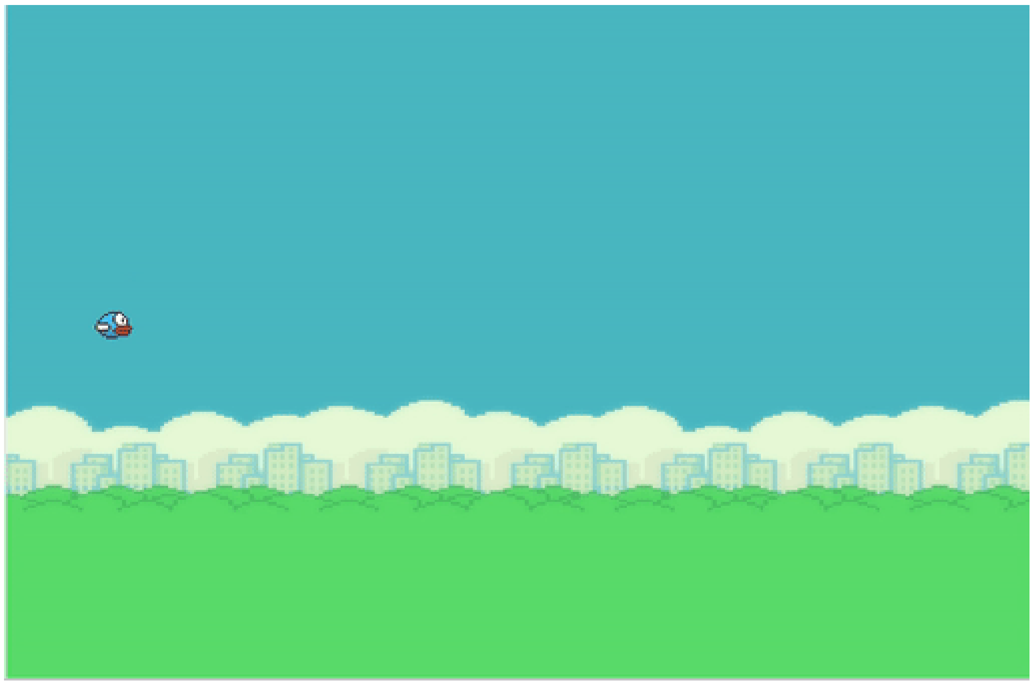
We use the same `add`[18] function we used for adding the background. This time, we grab a reference, `const player`, to the returned game object. This is so we can use this reference later when checking for collisions, or flapping up when the player taps the space bar.

[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

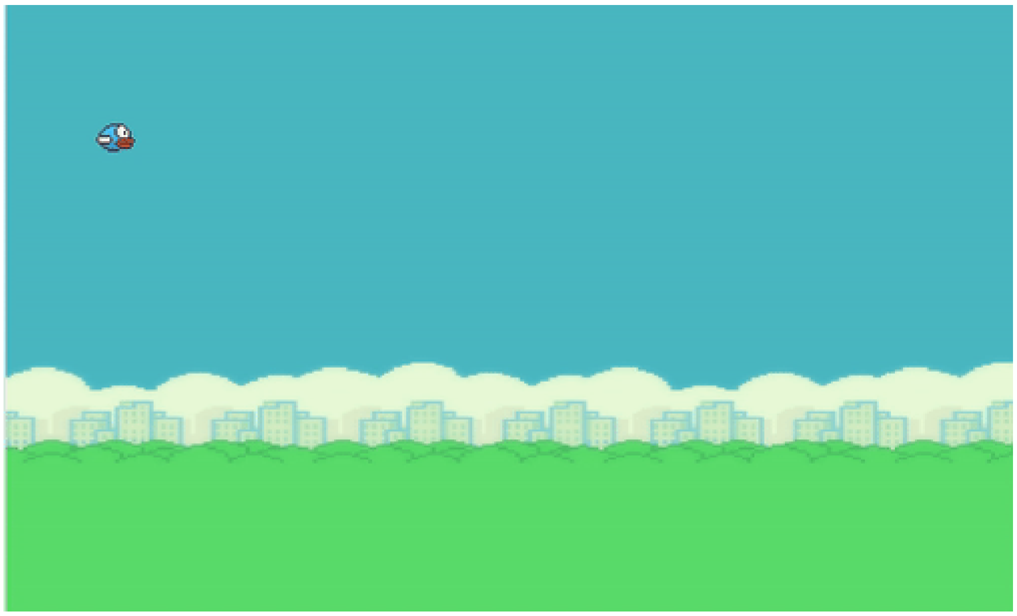
Our next task is to save Flappy from plummeting to their death by giving control to the player to flap Flappy's wings. We'll use the spacebar for this. Kaboom has an `onKeyPress`[25] function, which fires a callback with custom code when the specified key is pressed. Add this code to the `game` scene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

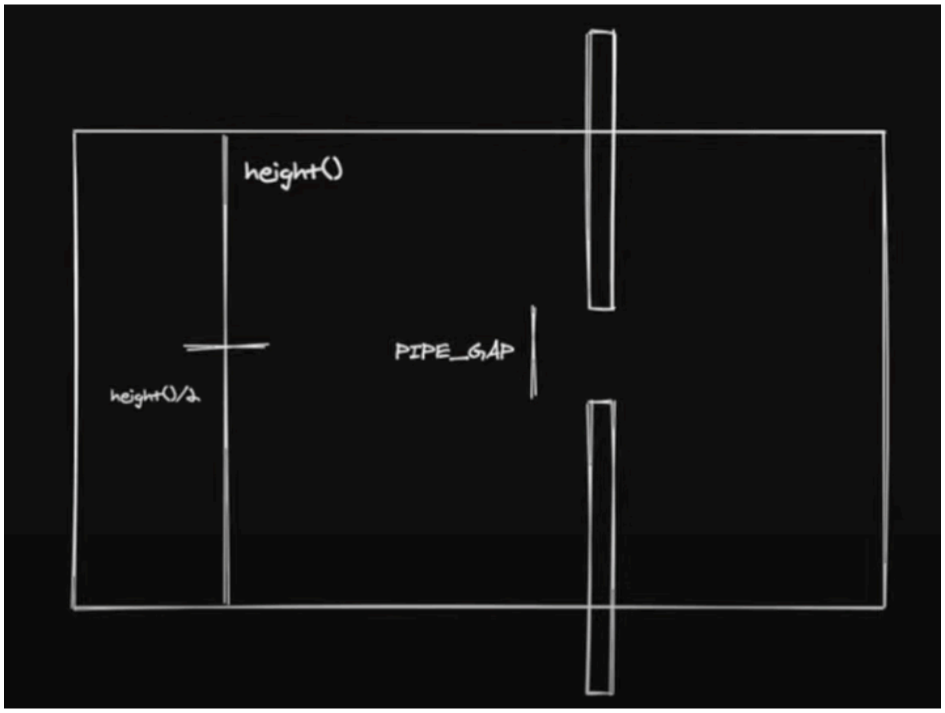
Click to open gif[29]

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

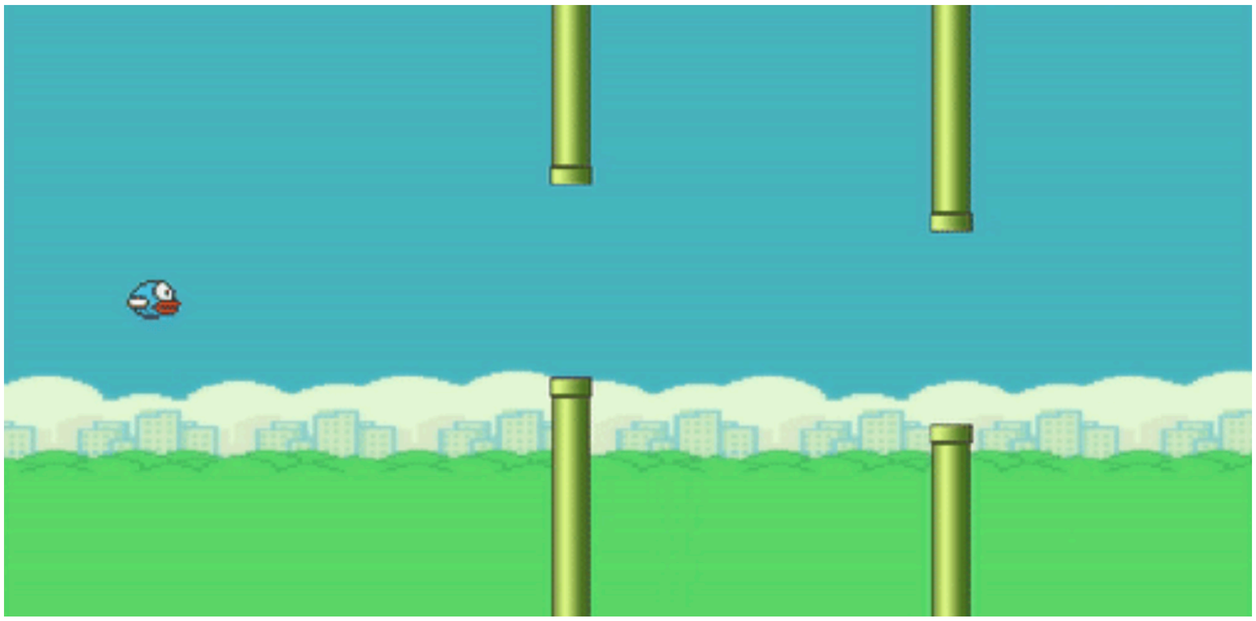
```
onUpdate("pipe", (pipe) => {
        pipe.move(-160, 0);
});
```

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:

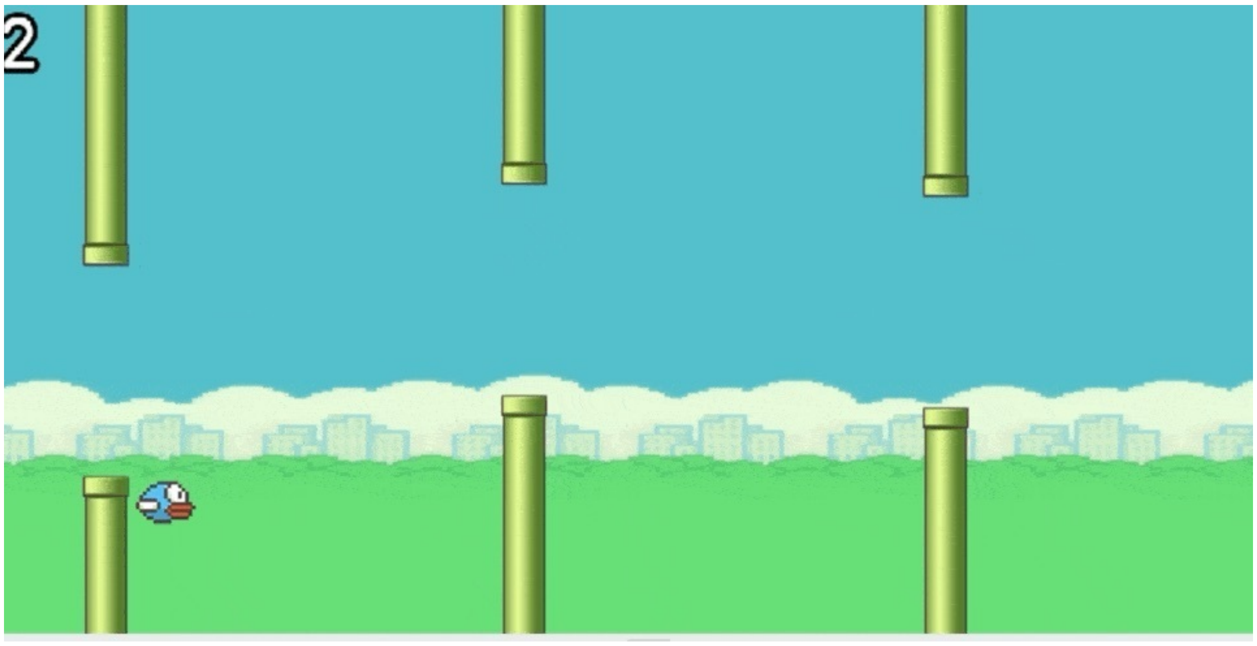
```
let score = 0;
const scoreText = add([
        text(score, {size: 50})
]);
```

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

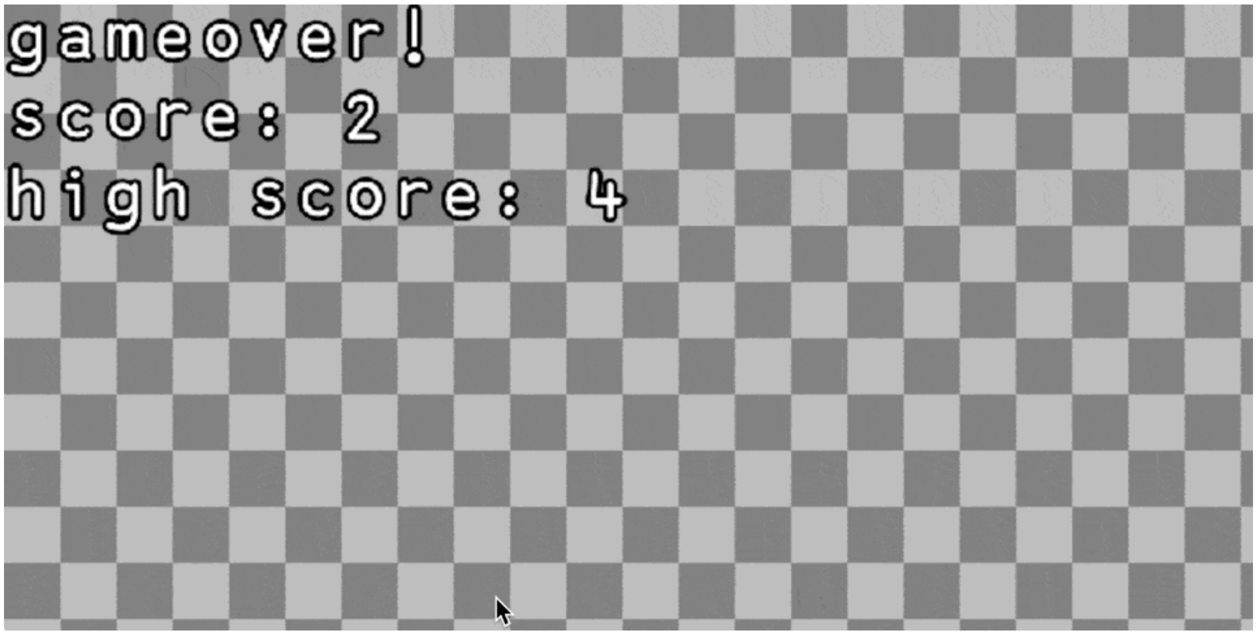
Let's include a quick way for the player to play again and try to beat their score. We use the `onKeyPress`[44] to listen for the player pressing the `space` bar. In our key-press handler, we `go`[45] back to the main `game` scene, to start the game all over again.

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

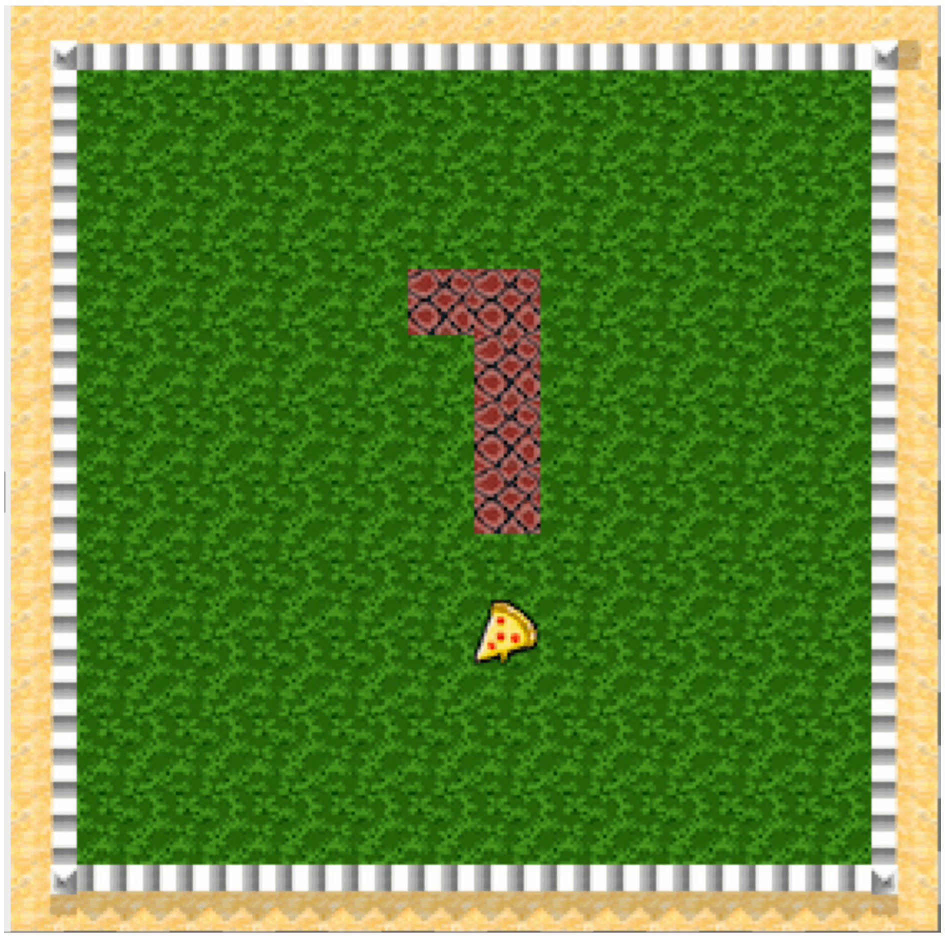
You can find the code for this tutorial on Replit[51]

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

Building Snake with Kaboom.js

Snake was an incredibly popular game, mostly remembered from 1990s era cell phones. At the time, it was often the only game you'd find on a phone. In the most basic form, it's a super simple game, but still wildly entertaining. It's also a great game to build when you are learning the basics of game making.

In this tutorial, we'll implement Snake using Kaboom.js[52] built into Replit[53]



The finished game

Click to open gif[54]

[52]https://kaboomjs.com
[53]https://replit.com
[54]https://docs.replit.com/images/tutorials/21-snake-kaboom/updated-graphic.gif

Overview and Requirements

We'll use the Replit[55] web IDE to create our version of Snake. If you don't already have a Replit account, create one now[56].

Let's think a bit about what we need to do. Snake, at its core, is a series of blocks representing a snake moving around a grid, with the player controlling the direction. It also has simple rules – when the snake touches the sides of the screen, it dies. If the snake crosses itself, it also dies. If the snake eats some food (a different type of block), it grows by 1 block. The food then re-appears at another random place on the screen.

A few components we will need to build are:

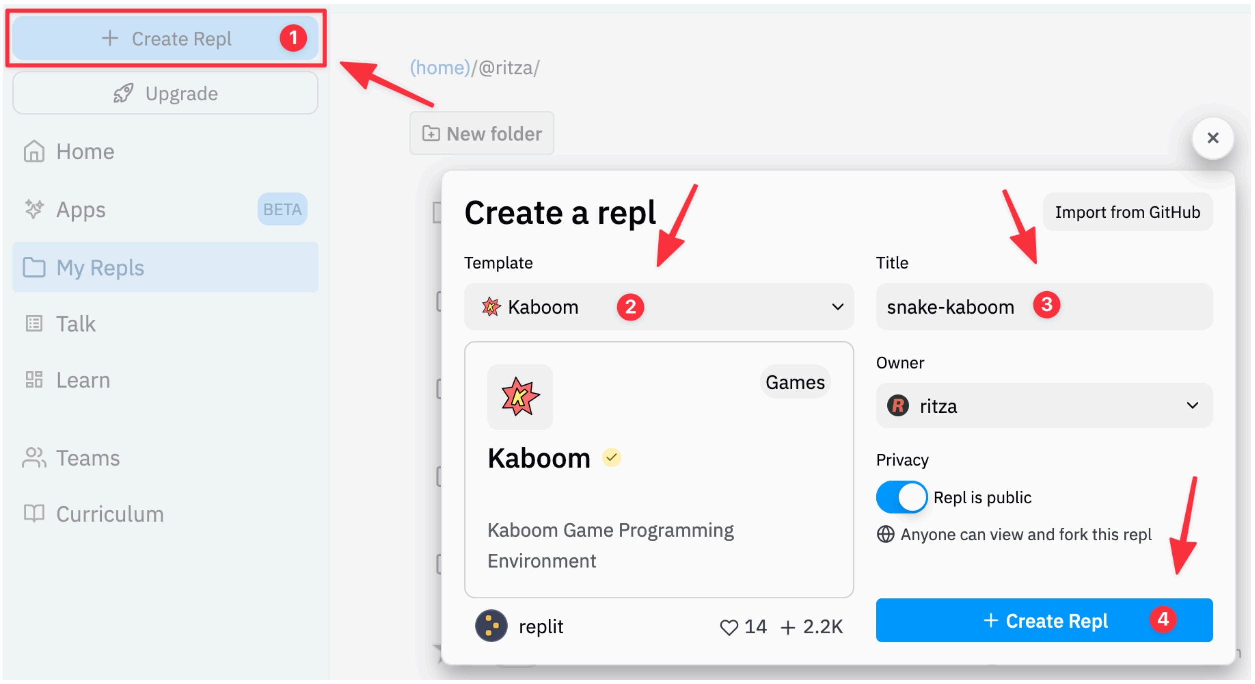
- A way to draw the blocks and move them on the screen.
- A way to get steering directions from the player to the snake.
- A way to determine if the snake has gone out of bounds of the screen.
- A way to determine if the snake has crossed over itself (or "bitten itself", as another analogy).
- A way to randomly place the food on the screen.
- A way to determine if the snake has eaten, or touched, the food.
- A way to grow the snake.

That's a lot to think about! Let's get started and create a project in Replit[57].

Creating a New Project

Log into your Replit[58] account and create a new repl. Choose **Kaboom** as your project type. Now, give this repl a name, like "snake-kaboom".

[55]https://replit.com
[56]https://replit.com/signup
[57]https://replit.com
[58]https://replit.com



Create a repl

Template: Kaboom 2

Title: snake-kaboom 3

Owner: ritza

Privacy: Repl is public

Anyone can view and fork this repl

Creating a new repl

After the repl has booted up, you should see a `main.js` file under the "Scenes" section. This is where we'll start coding.

Getting Started with Kaboom.js

Kaboom.js[59] is a JavaScript library that contains many useful features to make simple in-browser games. It has functionality to draw shapes and sprites (the images of characters and game elements) to the screen, get user input, play sounds, and more. We'll explore these features and learn how they work by using some of them in our game.

Kaboom.js also makes good use of JavaScript's support for callbacks[60]; instead of writing loops to read in keyboard input and check if game objects have collided (bumped into each other), Kaboom.js uses an event model, where it tells us when such an event has occurred. Then we can connect up callback functions[61] that act on these events.

Using Kaboom.js in Replit takes care of all the boilerplate initialisation code, as well as asset loading, so we can concentrate on writing the game logic and making game graphics and sound.

[59]https://kaboomjs.com
[60]https://developer.mozilla.org/en-US/docs/Glossary/Callback_function
[61]https://developer.mozilla.org/en-US/docs/Glossary/Callback_function

Creating the Game Map

To start, we can get our game board, or map drawn on the screen. This will define the edges of the board so that if the snake crashes into them, we can detect and end the game.

Kaboom.js has built-in support for defining game maps, using a text array and the function `addLevel`[62]. This takes away a lot of the hassle normally involved in loading and rendering maps.

Replace the example code in `main.js` file with the following to create the game board:

```
import kaboom from "kaboom";

kaboom();

const block_size = 20;

const map = addLevel([
    "==============",
    "=            = ",
    "=            = ",
    "=            = ",
    "=            = ",
    "=            = ",
    "=            = ",
    "=            = ",
    "=            = ",
    "=            = ",
    "=            = ",
    "=            = ",
    "=            = ",
    "==============",
  ], {
  width: block_size,
  height: block_size,
  pos: vec2(0, 0),
  "=": () => [
    rect(block_size, block_size),
    color(255,0,0),
    area(),
    "wall"
  ]
});
```

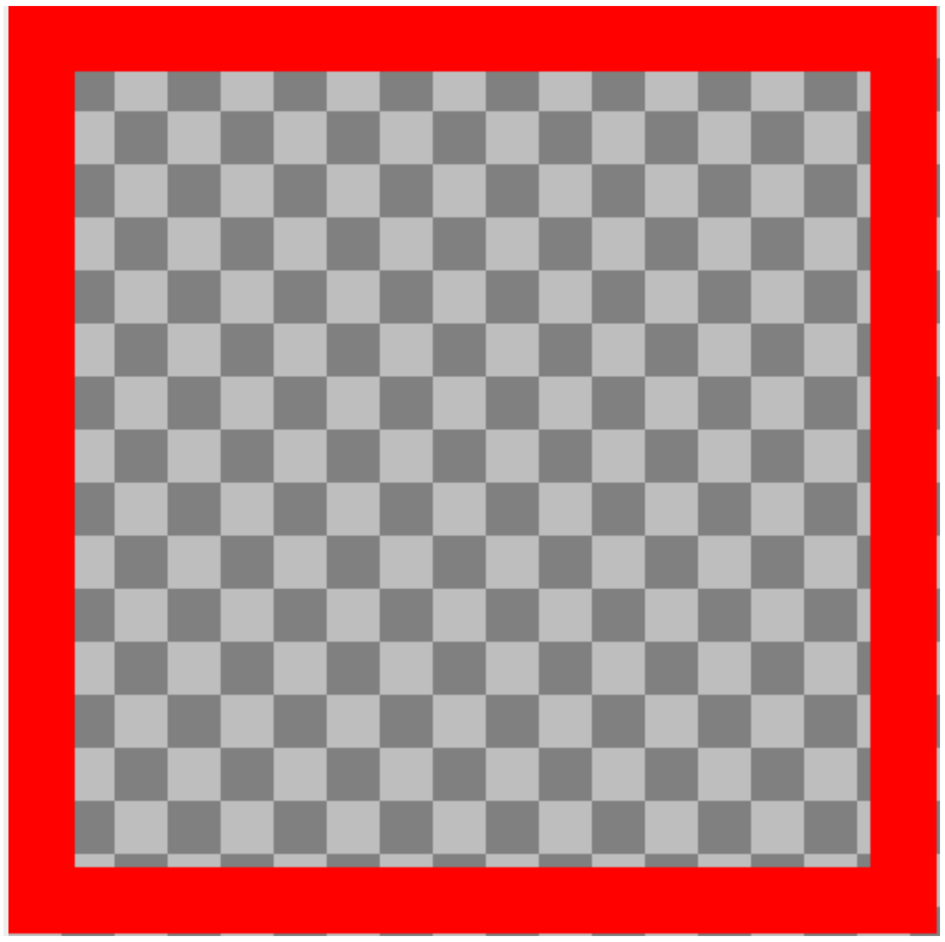
[62]https://kaboomjs.com/doc/#addLevel

On the first line we import the kaboom library, and then initialize the context by calling `kaboom()`. This will give us a blank canvas with a nice checkerboard pattern. We then create a constant for the size of each block on our grid. This is just so we don't need to keep typing in the number, and also helps if we want to experiment later with different block sizes etc.

Then we create the game map. The map, or level design, is expressed in an array of strings. Each row in the array represents one row on the screen. So, we can design visually in text what the map should look like. The `width` and `height` parameters specify the size of each of the elements in the map. The `pos` parameter specifies where on the screen the map should be place – we choose `(0,0)`, which is the top left of the screen, as the starting point for the map.

Then Kaboom.js allows us to specify what to draw for each symbol in the text map. We're only using one symbol here, `=`, but you can make maps out of many different elements – e.g., a symbol for a wall, a symbol for water, a symbol for a health kit and so on. To tell Kaboom.js what to draw for the symbol, we add the symbol as a key, as in `=`, and then specify parameters for it. In this code, we draw a red rectangle as each piece of the boundary wall. The `area()` component generates the collision area which will be useful when we want to check for collision between the snake and wall later on. The string `wall` assigns a tag to each of the pieces of wall drawn, which will also help us with collision detection later on.

If we run this code, we should see the outline of a red square on the screen, representing the map.



Boundary wall

Adding the Snake

Now that we have a map, let's add the snake. The snake is made up of a number of blocks moving together. We'll need to keep track of these so that we can move them together, so an array[63] would be a good data structure to use here.

We also need to start the snake off with a given size, position and direction to move in on the map. It can return to these each time the game ends as well. So we should make a function that we can call whenever a new game starts, or the old one ends, to reset the snake to a default position and size.

We'll need to add a few variables and constants that our snake drawing function will use. Add these definitions above the `const block_size = 20;` we added earlier:

[63]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

```
const directions = {
  UP: "up",
  DOWN: "down",
  LEFT: "left",
  RIGHT: "right"
};

let current_direction = directions.RIGHT;
let run_action = false;
let snake_length = 3;
let snake_body = [];
```

First, we define an object with properties for each of the allowable directions the snake can move in. This makes code that checks and changes directions easy to read and change, compared to just using numbers or strings to define the directions. The variable `current_direction` tracks the direction the snake is moving at any given time. We choose a starting direction, `RIGHT`, as its default. `run_action` is a flag variable that we'll use to flag if we are in the actual game, or setting up, or ending the game. The variable `snake_length` keeps track of how long the snake tail has become – we start it at a chosen value of 3. Finally, `snake_body` holds all the screen objects that make up the snake's body.

Now we can add a function to spawn, and respawn, the snake.

Let's add this function:

```
function respawn_snake(){
  destroyAll("snake");

  snake_body = [];
  snake_length = 3;

  for (let i = 1; i <= snake_length; i++) {
      let segment = add([
          rect(block_size ,block_size),
          pos(block_size ,block_size *i),
          color(0,0,255),
          area(),
          "snake"
      ]);
      snake_body.push(segment);
  };
  current_direction = directions.RIGHT;
}

function respawn_all(){
  run_action = false;
    wait(0.5, function(){
        respawn_snake();
        run_action = true;
    });

}

respawn_all();
```

First, the function gets rid of any existing snake segment objects by using the Kaboom.js `destroyAll` function[64]. This removes any object with the given tag from the game. Then we reset our segment array to an empty array, and the snake length back to the default.

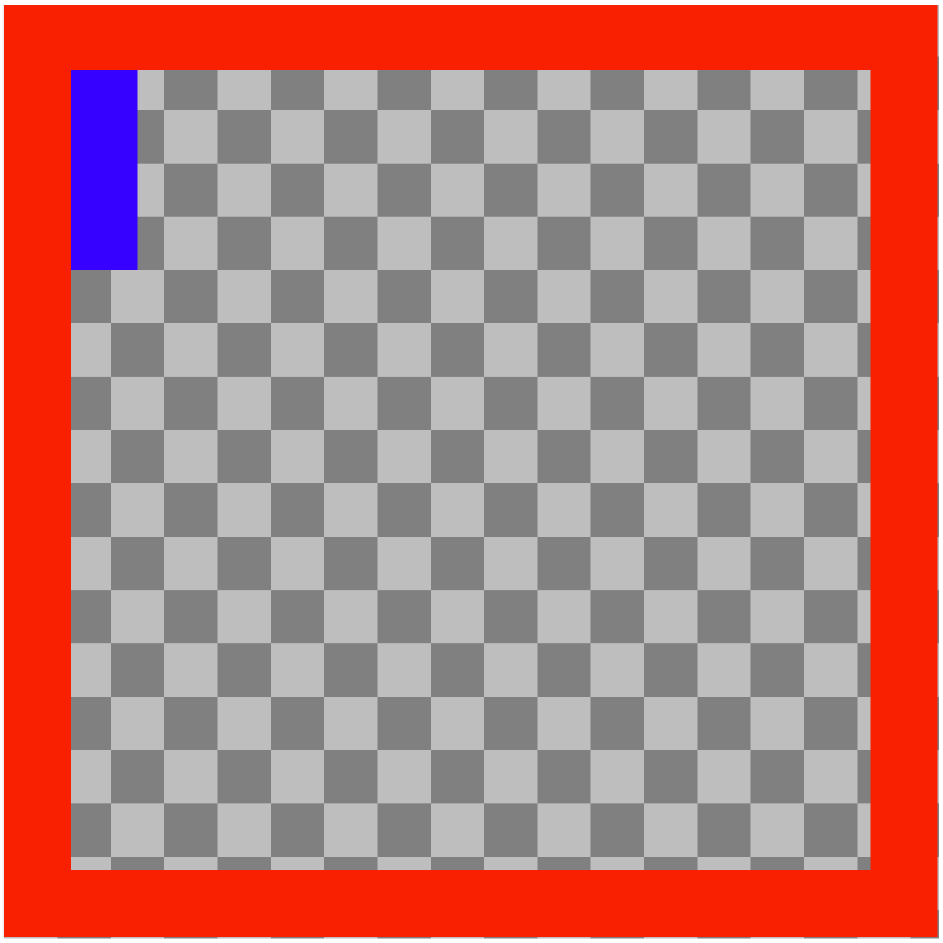
Then the function sets up a loop to create new snake segments, up to the length we specified. It does this by calling the Kaboom.js `add`[65] method, which adds a new object to the game. `add` takes a few parameters, as components of the object to create. We pass in components to specify how to draw the object (using `rect`[66]), its color, and a tag "snake" to identify the segments when we are checking collisions, and updating/removing segments. We also specify the position for the segment we create. To create the starting snake, we just ensure it is at least one `block_size`, or block, from the left side, and then add each subsequent segment one more block down per loop. This gives a straight snake pointing down to start. Then we add the new segment to our `snake_body` array to keep track of it.

Finally, we set a default starting direction for the snake to move in.

You'll notice that we also add in a function `respawn_all`, and a call to the function `respawn_snake`. We'll use the `respawn_all` function to call all of our other respawn functions. Currently we have one for the snake, but we'll also need one for the food when we add it. In the `respawn_all` function, we also take care to set the `run_action` flag to false, so that no updates are made while we are setting up or resetting objects. We also wrap the calls in a Kaboom.js `wait` function[67], with a small delay of 0.5 seconds. This is because when we detect a "game over" condition, we don't immediately want to reset the game, as it could be a bit disorienting to a player.

Running the code now, you should see a blue line at the top-left side of the map.

[64]https://kaboomjs.com/doc/#destroyAll
[65]https://kaboomjs.com/doc/#add
[66]https://kaboomjs.com/doc/#rect
[67]https://kaboomjs.com/doc/#wait



Static snake

Moving the Snake

Now that we've got map boundaries, and a snake drawn on the screen, we can work on getting player input and moving the snake around.

Kaboom.js has a function `onKeyPress`[68], which can call a supplied function whenever a particular key is pressed. We'll use that to determine which way the player wants the snake to go. Add this code to get user direction input:

[68]https://kaboomjs.com/doc/#onKeyPress

```
onKeyPress("up", () => {
    if (current_direction != directions.DOWN){
        current_direction = directions.UP;
    }
});

onKeyPress("down", () => {
    if (current_direction != directions.UP){
        current_direction = directions.DOWN;
    }
});

onKeyPress("left", () => {
    if (current_direction != directions.RIGHT){
        current_direction = directions.LEFT;
    }
});

onKeyPress("right", () => {
    if (current_direction != directions.LEFT){
        current_direction = directions.RIGHT;
    }
});
```

For each of the named "arrow" keys, we set up a function to call if the key is pressed. In each of these functions, we check to ensure that the new direction input is not the complete opposite direction to which the snake is currently moving. This is because we don't want to allow the snake to reverse. If the input direction is a legal move, we update the `current_direction` property to the new direction.

Now we need to think about how to make the snake appear to move on the screen. A way to do this is to check which direction the snake is heading, and add a block in front of the snake in that direction. Then we'll need to remove a block at the tail-end of the snake. We'll need to do this a few times in a second so that the snake appears to be moving smoothly. Kaboom.js has a function `onUpdate`[69] which can be used to update game objects on each frame. Add the following code, which uses the `onUpdate` function, to move the snake:

[69] https://kaboomjs.com/doc/#onUpdate

```
let move_delay = 0.2;
let timer = 0;
onUpdate(()=> {
     if (!run_action) return;
     timer += dt();
     if (timer < move_delay) return;
     timer = 0;

    let move_x = 0;
    let move_y = 0;

    switch (current_direction) {
        case directions.DOWN:
            move_x = 0;
            move_y = block_size;
            break;
        case directions.UP:
            move_x = 0;
            move_y = -1*block_size;
            break;
        case directions.LEFT:
            move_x = -1*block_size;
            move_y = 0;
            break;
        case directions.RIGHT:
            move_x = block_size;
            move_y = 0;
            break;
    }

    // Get the last element (the snake head)
    let snake_head = snake_body[snake_body.length - 1];

    snake_body.push(add([
        rect(block_size,block_size),
        pos(snake_head.pos.x + move_x, snake_head.pos.y + move_y),
        color(0,0,255),
        area(),
        "snake"
    ]));

    if (snake_body.length > snake_length){
        let tail = snake_body.shift(); // Remove the last of the tail
        destroy(tail);
    }

});
```

We set the action to run every 0.2 seconds, or 5 times a second to get smooth movement. Since the `action` function updates game objects on each frame we use the `dt()` function[70] to get the time that has elapsed between the previous and current frame, so that we can keep track if 0.2 seconds has elapsed for us to move the snake. If the desired delay has not elapsed we exit early without updating anything otherwise we reset the timer and execute the code to move the snake. You can try experiment with different times to see the effect on the game by adjusting the value of the `move_delay` variable. We also check the flag variable `run_action` we defined earlier – if it is false, we exit early without updating anything. Then, the function defines 2 local variables, `move_x` and `move_y`, which is used to determine where to place the 'next' block relative to the head of the snake.

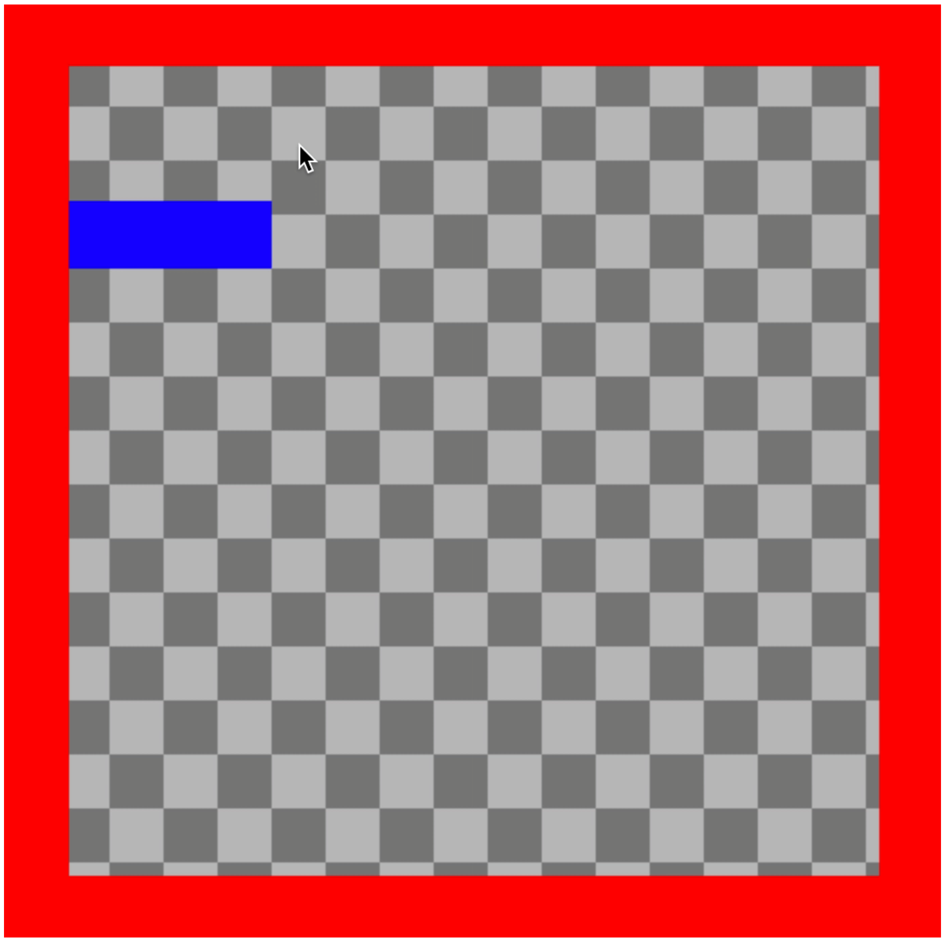
Then the function switches on the value of the current direction the snake is heading in. For each direction, the `move_x` and `move_y` are set to either 0, block_size or -1 * block_size. If the snake is moving left or right, we add or subtract a block from the x dimension accordingly. The same occurs if the snake is moving up or down, but in the y dimension.

After the switch, we get the current snake head by indexing the last element in the snake body array. Now that we have both the current snake head position, and the position amount relative to the snake head to move in, we can create the new snake head by adding a new block game object. This is similar to the code we used in `respawn_snake`.

Now all that remains is to remove a block at the tail end of the snake, using the built-in array `shift` function[71], which removes the first element from an array, and returns that element. Because our 'oldest' part of the snake, also known as a tail, is the first element, we call shift on the array, and then the Kaboom.js `destroy` function[72] to get rid of the segment. We only do this if the current length of the snake body array is greater than our determined snake length. This means if we increase `snake_length`, the overall length of the snake on the screen will also increase. We can use this when we add food to the game.

Running the project now and clicking into the game screen should allow you to move the snake around. Note that there isn't collision detection yet, so the snake can go out of bounds without consequence.

[70]https://kaboomjs.com/doc/#dt
[71]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/shift
[72]https://kaboomjs.com/doc/#destroy



Moving the snake

Click to open gif[73]

Adding Snake Food

We have a snake, it moves, and a player can steer it. Let's add some food that the snake can eat, which will cause it to grow longer. Whenever the snake eats the food, we'll have to respawn the food again, so let's write the food creation code in a function as well, like we did for the snake:

[73] https://docs.replit.com/images/tutorials/21-snake-kaboom/snake-move.gif

```
let food = null;

function respawn_food(){
    let new_pos = rand(vec2(1,1), vec2(13,13));
    new_pos.x = Math.floor(new_pos.x);
    new_pos.y = Math.floor(new_pos.y);
    new_pos = new_pos.scale(block_size);

    if (food){
        destroy(food);
    }
    food = add([
                rect(block_size ,block_size),
                color(0,255,0),
                pos(new_pos),
                area(),
                "food"
            ]);
}
```

Firstly, we set up a variable `food` so that we can keep track of food objects we create. You can move this variable up to where we declared other variables, like `block_size` and `snake_body` if you want to keep them all neatly in the same place.

Then the function `respawn_food` does a few things. In the game of snake, once a food block is eaten, another one appears at a random location on the grid. This means we'll need a random number generator to determine the location to place the food. Kaboom.js has a function called `rand`[74] which we can use to find a random position on the screen to place the food. We need both random `x` and `y` co-ordinates – conveniently, the `rand` function can accept 2D vectors as the start and end amount for the random range to generate numbers in, and will then return another 2D vector as a result.

Why do we choose a range of 1-13 for the random position of the food? If you look at the map we added earlier, it is 14 spaces across and 14 spaces down. These are the dimensions of our screen in grid blocks. Because we don't want to draw the food over the boundaries, we use 1-13 to choose blocks within the map. Now, the `rand` function returns real numbers, with decimals, not integers. This means we need to add the `Math.floor` call to truncate any decimals off the random numbers, as we don't want to place the food halfway through a particular grid block. We also need to convert from our grid co-ordinates to regular screen pixels. This is done by multiplying each co-ordinate by the `block_size`, which specifies the size of each grid block in pixels. We make use of the Kaboom.js `scale` method[75] on the `vec2` class to perform the multiplication.

The next part of the function checks if the `food` variable already contains an existing food object. If it

[74]https://kaboomjs.com/doc/#rand
[75]https://kaboomjs.com/doc/#vec2

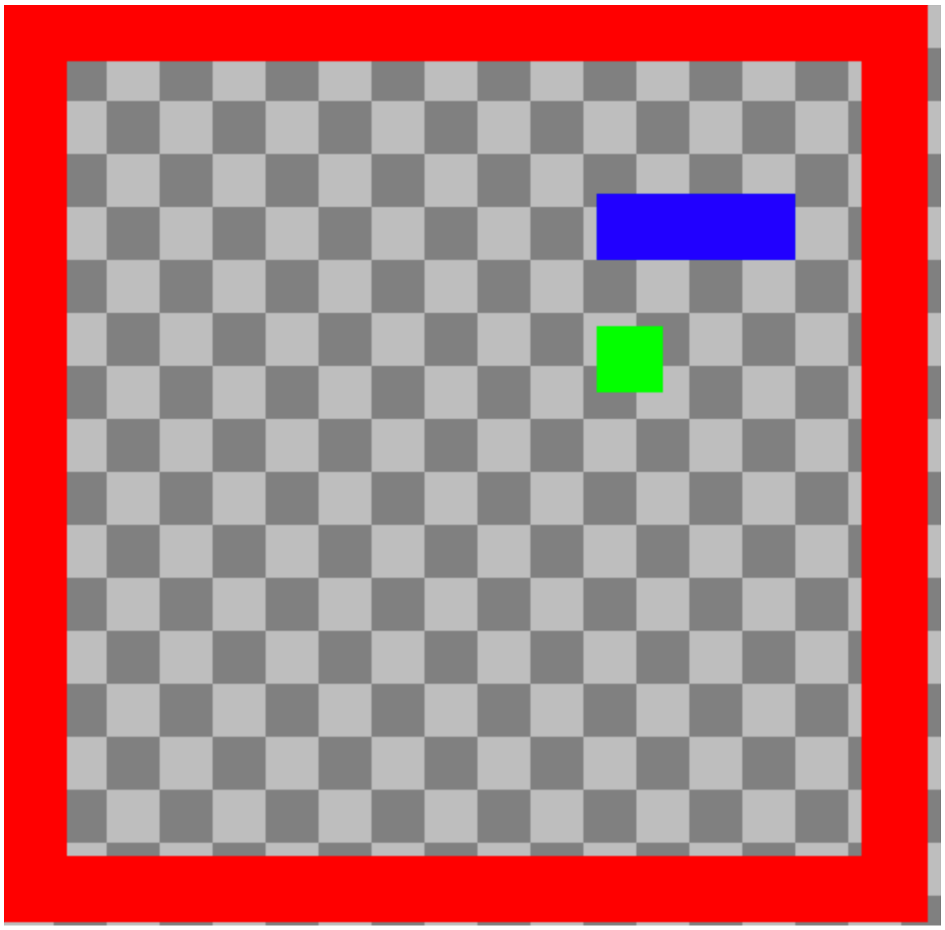
does, we call `destroy`[76] to remove that food from the game. Finally, the function creates a new food object by calling the Kaboom.js `add`[77] function to create a new food object at the random position we made.

To call this new `respawn_food` function, we need to update our `respawn_all` function, like this:

```
function respawn_all(){
  run_action = false;
    wait(0.5, function(){
        respawn_snake();
        respawn_food();
        run_action = true;
    });
}
```

Running the game now shows a green food block positioned somewhere randomly on the map:

[76]https://kaboomjs.com/doc/#destroy
[77]https://kaboomjs.com/doc/#add



Adding food

Detecting Collisions

Now that we have all the objects our game needs – a boundary wall, a snake, and food – we can move on to detecting interactions, or collisions, between these objects.

Kaboom.js has a useful function for helping with this: `onCollide`[78]. The function takes in 2 tags for different game object types, and calls a provided callback function if there is a collision of the objects.

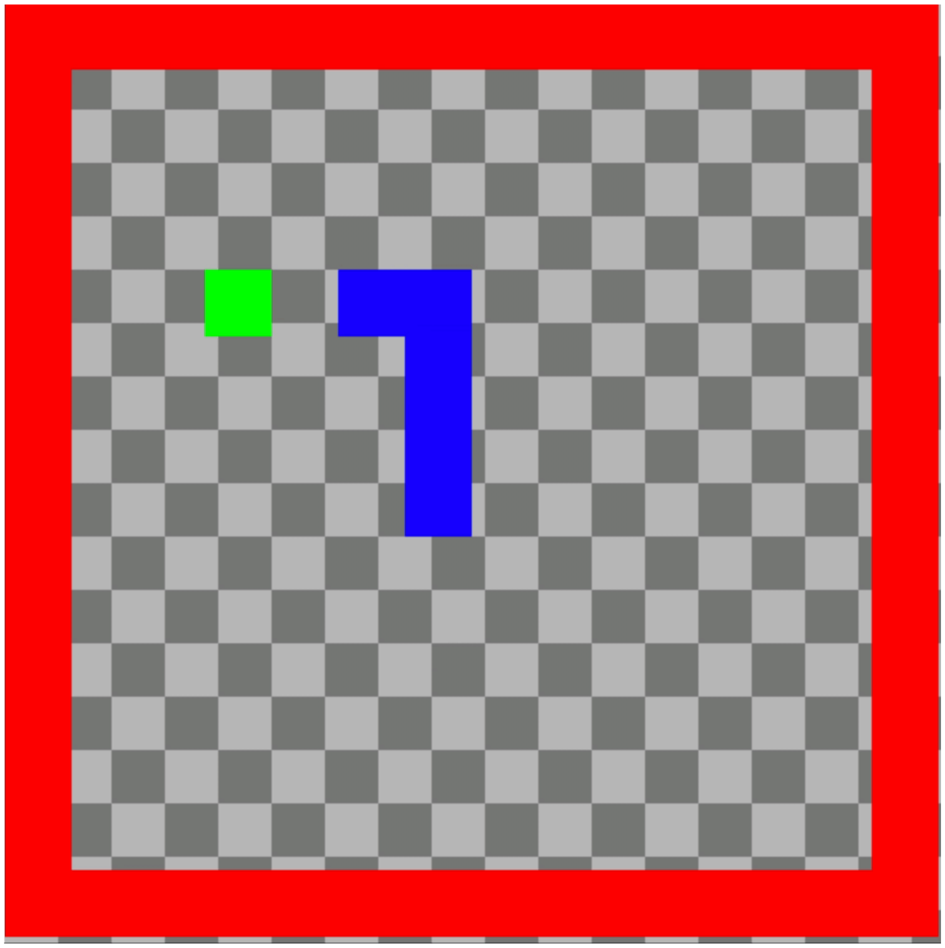
Let's start with detecting if the snake moves over a food block. Add the code below:

```
onCollide("snake", "food", (s, f) => {
    snake_length ++;
    respawn_food();
});
```

[78]https://kaboomjs.com/doc/#onCollide

We set up the `onCollide` function with tags for the snake, and the food object. Then, in the callback function, `snake_length` is increased by `1`, and we call `respawn_food` to replace the eaten food somewhere else on the map.

Running this, and eating the food, you should see the snake grow each time, and the food re-appear on another block:


Eating food

Click to open gif[79]

Now, we can add similar code to detect if the snake has hit the wall:

```
onCollide("snake", "wall", (s, w) => {
    run_action = false;
    shake(12);
    respawn_all();
});
```

[79]https://docs.replit.com/images/tutorials/21-snake-kaboom/eat-food.gif

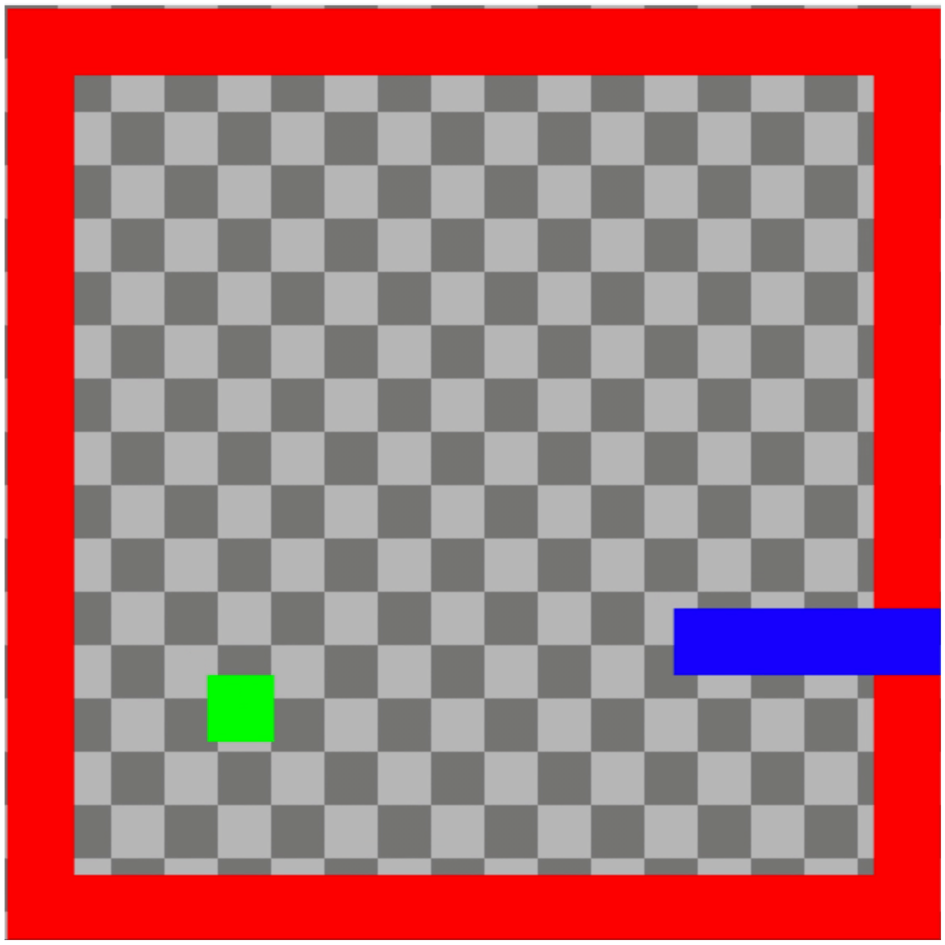
In the callback function, we immediately set the `run_action` flag to false. This is so that the code in the move loop does not run and create the appearance of the snake stuck in the wall. Then the code calls a cool Kaboom.js effect function `shake`[80], which "shakes" the screen in a way that makes it feel like the snake has crashed heavily, and communicates quite effectively that the game is over. Finally, we call `respawn_all` to reset all the game objects.

We can use the same code to detect if the snake has hit itself – we just replace the `wall` tag with another `snake` tag:

```
onCollide("snake", "snake", (s, t) => {
    run_action = false;
    shake(12);
    respawn_all();
});
```

Running the game now, and crashing into the wall should look something like this:

[80]https://kaboomjs.com/doc/#shake



Snake prang

Click to open gif[81]

Congratulations! You've finished creating Snake in Kaboom.js!

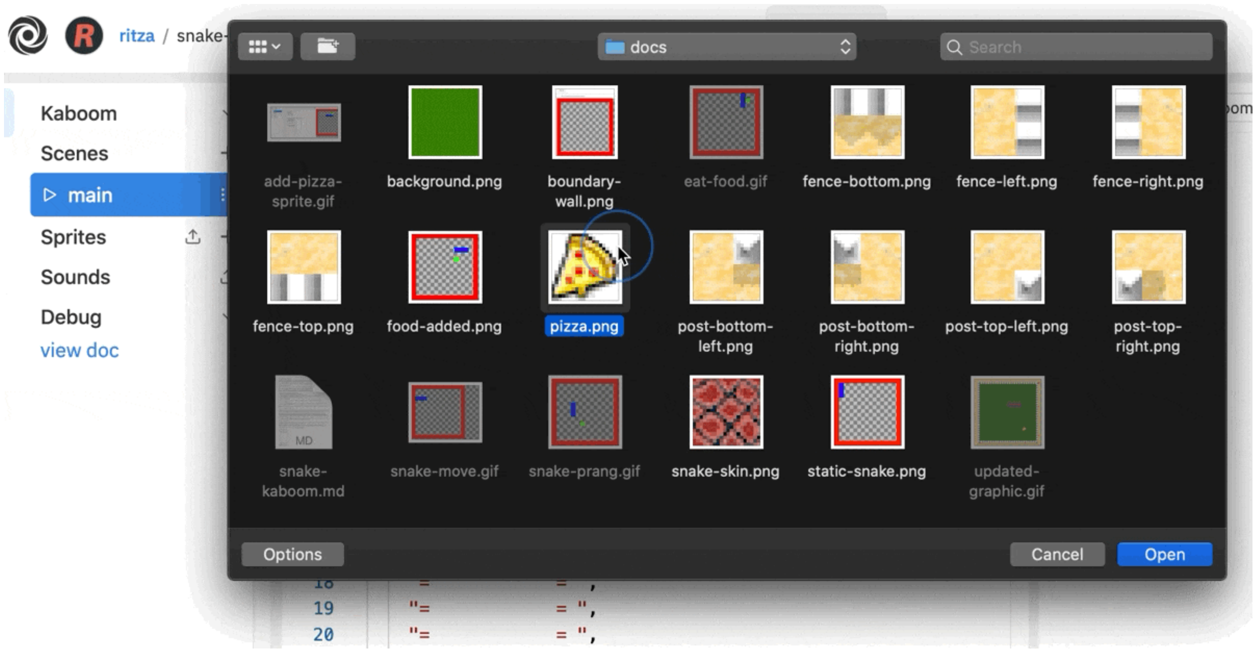
Improving the Graphics

We have a working snake game, but it does look a bit bland. Kaboom.js has good support for sprites[82], which are small pictures used to represent game objects and characters. Replit also has built-in management and loading of sprites for Kaboom.js to take care of the overhead in using sprites.

Using sprites, let's give the snake something nicer to eat than a green block. Right click and select "Save image as" on the pizza slice below, and save it to your computer. Then, in Replit, click the upload button next to "Sprites" and upload the pizza to your repl.

[81]https://docs.replit.com/images/tutorials/21-snake-kaboom/snake-prang.gif
[82]https://en.wikipedia.org/wiki/Sprite_%28computer_graphics%29

Pizza



Adding a pizza sprite

Click to open gif[83]

Now, we can update the `respawn_food` function to use this sprite, instead of drawing a green block. Remove the lines `rect` and `color`, and replace with a call to add the pizza sprite, like this:

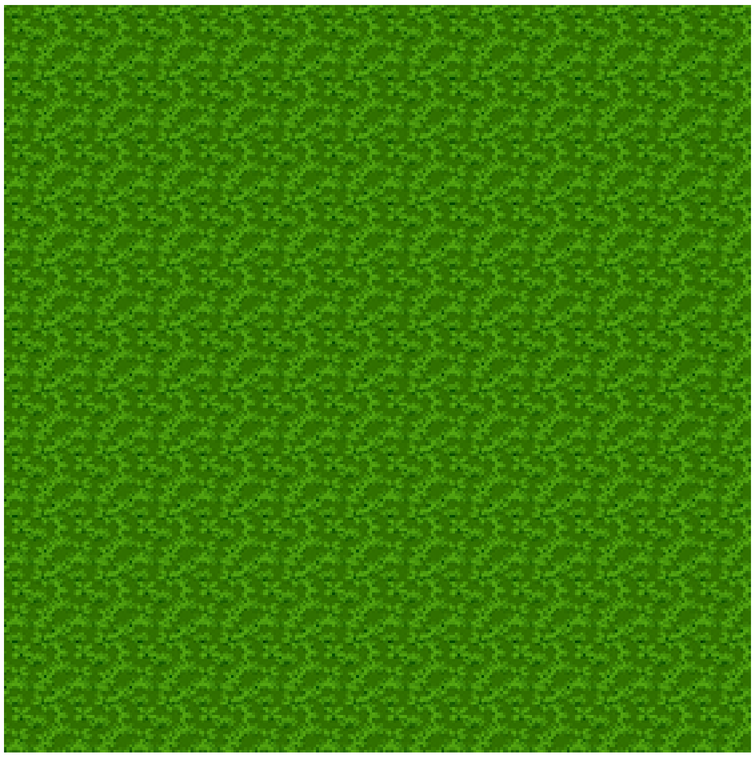
```
function respawn_food(){
    let new_pos = rand(vec2(1,1), vec2(13,13));
    new_pos.x = Math.floor(new_pos.x);
    new_pos.y = Math.floor(new_pos.y);
    new_pos = new_pos.scale(block_size);

    if (food){
        destroy(food);
    }
    food = add([
                sprite('pizza'),
                pos(new_pos),
                area(),
                "food"
            ]);
```

[83]https://docs.replit.com/images/tutorials/21-snake-kaboom/add-pizza-sprite.gif

```
}
```

We can also update the background to be more interesting. To do this, we can make use of Kaboom.js' layers[84] concept. This allows us to create different graphic layers, for example one for a static background image, another one for the active game objects over that, and another top layer for stats and scores etc.

We'll create `2` layers, background and game, to support a background. Download and add the background grass image below to your repl as you did for the pizza slice:



background

Now, we can set up the layers and add the background grass. Add the following code to the top of the `main` file:

[84]https://kaboomjs.com/doc/#layers

```
layers([
    "background",
    "game"
], "game");

add([
    sprite("background"),
    layer("background")
]);
```

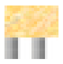
This sets up our 2 layers, and makes the `game` layer the default layer to draw on. Whenever we call `add`[85], we can optionally specify a layer to put the object on – if we don't specify a layer, it uses whatever we set as default in the call to `layers`[86]. Then we add our background sprite to the background layer.

Next, we can update the boundaries to look a bit better. Recall that in our map we add with `addLevel`[87], each different symbol we use can map to a different game object. Using this, we can create a good looking border fence, with different elements for each side and corners. Download the following 8 sprites as before, and upload them to your repl:

Fence bottom

Fence left

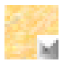
Fence right


Fence top

Post bottom left

Post bottom right

[85]https://kaboomjs.com/doc/#add
[86]https://kaboomjs.com/doc/#layers
[87]https://kaboomjs.com/doc/#addLevel


Post top left

Post top right

Now, we can update the level map to use these. Replace the previous `addLevel` code with the following code:

```
const map = addLevel([
    "1tttttttttttt2",
    "l            r ",
    "l            r ",
    "l            r ",
    "l            r ",
    "l            r ",
    "l            r ",
    "l            r ",
    "l            r ",
    "l            r ",
    "l            r ",
    "l            r ",
    "l            r ",
    "3bbbbbbbbbbbb4",
], {
    width: block_size,
    height: block_size,
    pos: vec2(0, 0),
    "t": ()=> [
        sprite("fence-top"),
        area(),
        "wall"
    ],
    "b": ()=> [
        sprite("fence-bottom"),
        area(),
        "wall"
    ],
    "l": ()=> [
        sprite("fence-left"),
        area(),
        "wall"
    ],
    "r": ()=> [
        sprite("fence-right"),
        area(),
        "wall"
    ],
    "1": ()=> [
        sprite("post-top-left"),
        area(),
        "wall"
    ],
    "2": ()=> [
        sprite("post-top-right"),
        area(),
        "wall"
    ],
    "3": ()=> [
        sprite("post-bottom-left"),
        area(),
        "wall"
    ],
    "4": ()=> [
        sprite("post-bottom-right"),
        area(),
        "wall"
    ],
});
```

The last thing is to upgrade the snake itself. Download the skin below, and upload to the repl as before.

Snake skin

We create snake pieces in 2 places: in the `respawn_snake` function, and in the draw loop. Update both to use the snake skin sprite instead of a blue block. The `respawn_snake` function should look like this:

```
function respawn_snake(){
  snake_body.forEach(segment => {
      destroy(segment);
    });
  snake_body = [];
  snake_length = 3;

  for (let i = 1; i <= snake_length; i++) {
      snake_body.push(add([
          sprite('snake-skin'),
          pos(block_size  ,block_size * i),
          area(),
          "snake"
      ]));
  }
  current_direction = directions.RIGHT;
}
```

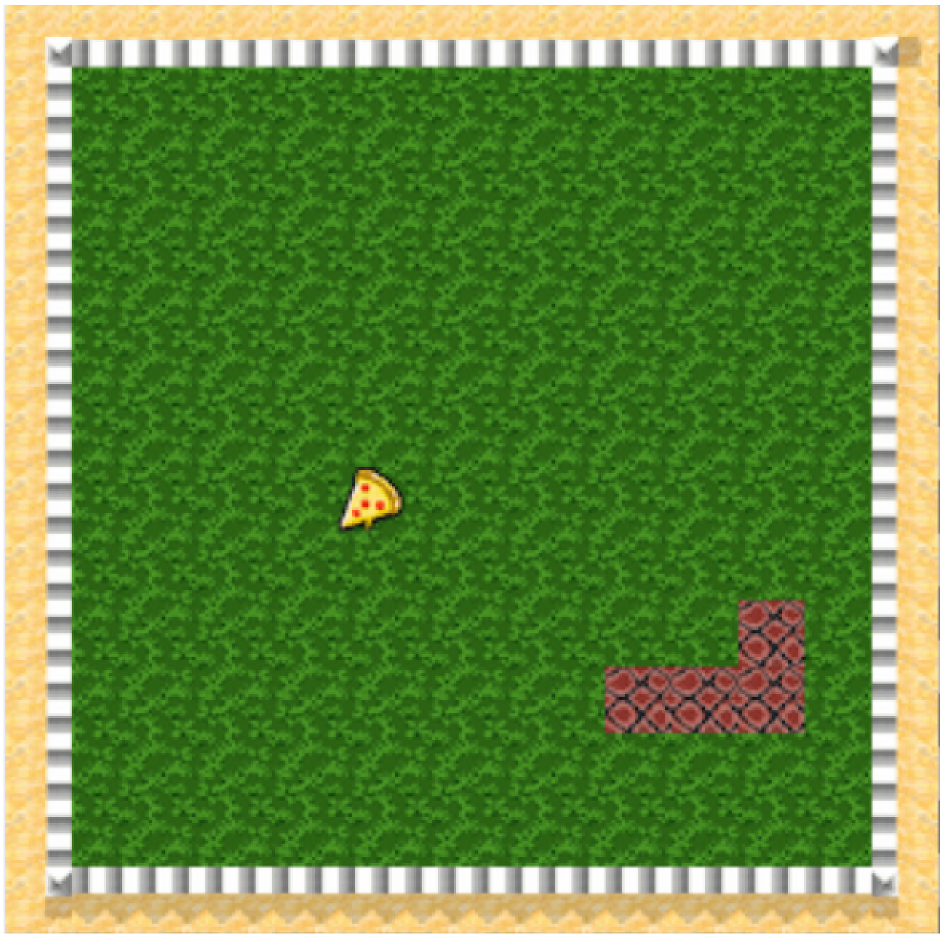
In the loop callback, the updated code for adding a new snake segment to the body should now look like this:

```
    snake_body.push(add([
        sprite('snake-skin'),
        pos(snake_head.pos.x + move_x, snake_head.pos.y + move_y),
        area(),
        "snake"
    ]));
```

Before we run the game we need to load the sprites that we made reference to in the code snippets above. Add the following code, below the line `kaboom();` to load the sprites and make them available in the game:

```
loadSprite("background", "sprites/background.png);
loadSprite("fence-top", "sprites/fence-top.png);
loadSprite("fence-bottom", "sprites/fence-bottom.png);
loadSprite("fence-left", "sprites/fence-left.png);
loadSprite("fence-right", "sprites/fence-right.png);
loadSprite("post-top-left", "sprites/post-top-left.png);
loadSprite("post-top-right", "sprites/post-top-right.png);
loadSprite("post-bottom-left", "sprites/post-bottom-left.png);
loadSprite("post-bottom-right", "sprites/post-bottom-right.png);
loadSprite("snake-skin", "sprites/snake-skin.png);
loadSprite("pizza", "sprites/pizza.png);
```

If you run the game now, you should see it looking much better!



Game graphics

Click to open gif[88]

Things to Try

There is a lot of good functionality in Kaboom.js[89] to try out, and make the game more entertaining. Here are some suggestions:

- Create a 2 player version.
- Add obstacles for the snake.
- Incrementally speed up the game as it goes on, to make it harder. You can do this by adjusting the delay parameter of the `loop` function as the game progresses.
- Add sound effects[90] and background music.

[88]https://docs.replit.com/images/tutorials/21-snake-kaboom/updated-graphic.gif
[89]https://kaboomjs.com/
[90]https://kaboomjs.com/doc/#play

Code

You can find the code for this tutorial on Replit[91]

[91]https://replit.com/@ritza/snake-kaboom

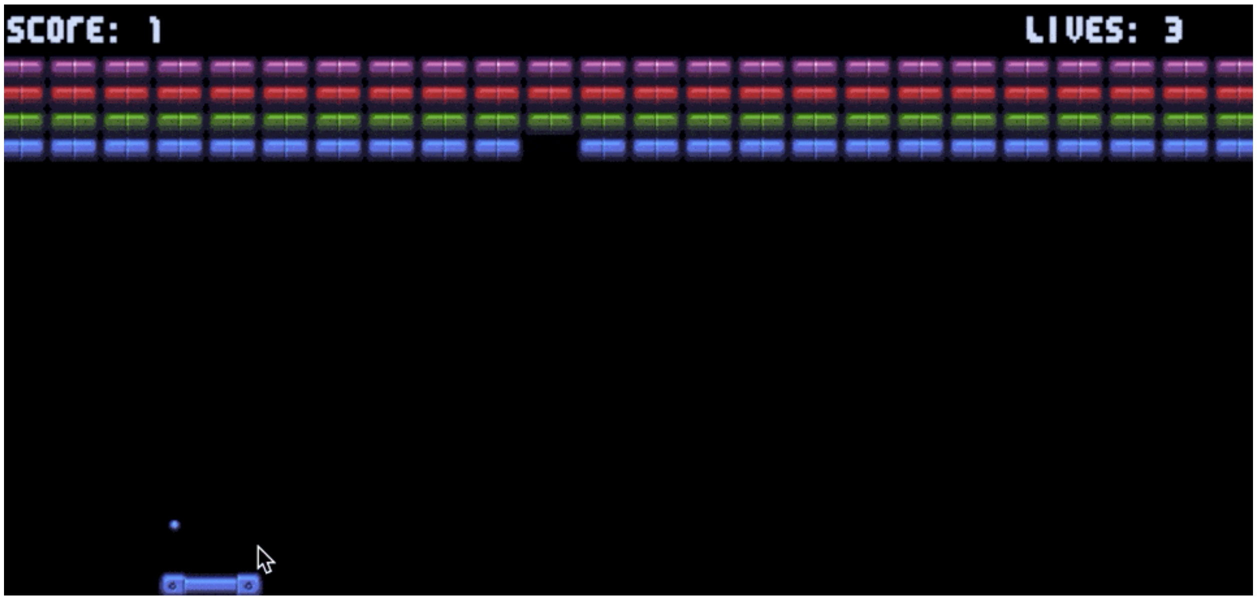
Building a block-breaking game with Kaboom.js

In this tutorial, we'll use the Kaboom framework to develop a simple block-breaking game, similar to classics like Atari's *Breakout* and Taito's *Arkanoid*.

By the end of this tutorial, you'll be able to:

- Use the Kaboom framework to develop a polished arcade game with multiple levels.
- Have a basic game to build on by adding your own powerups and block types.

Our finished game will look like this:



The finished game

Click to open gif[92]

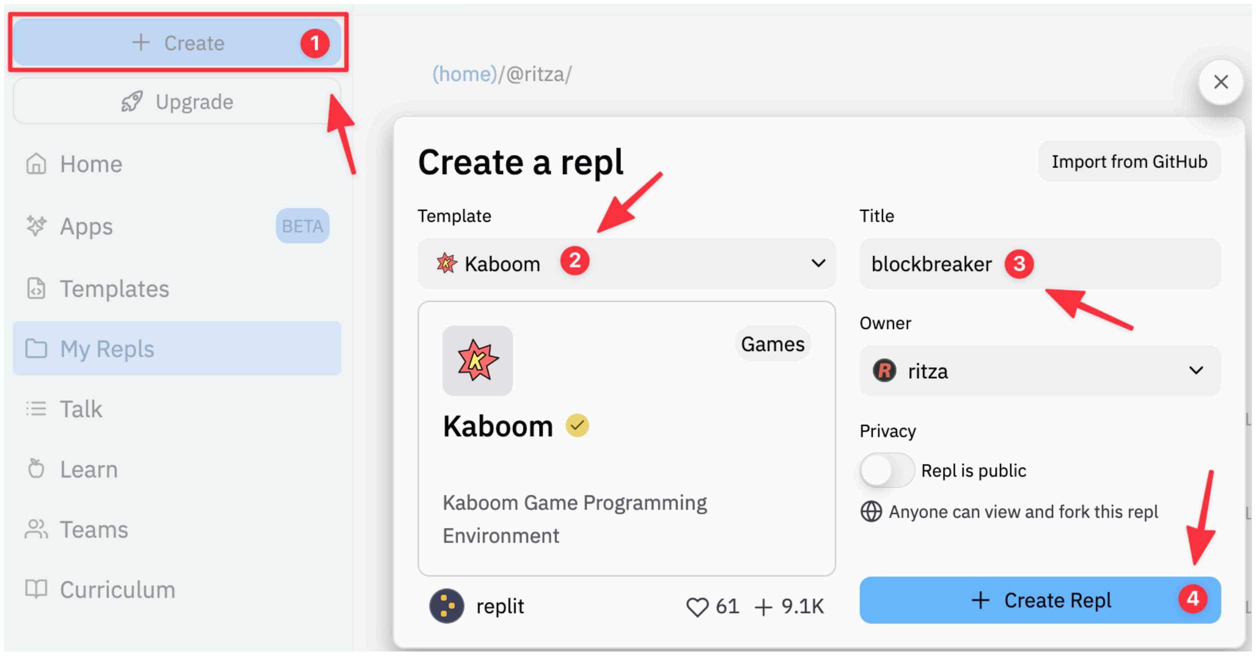
We will be using this set of sprites[93] by Michele Bucelli[94] and sound effects from this pack[95] by Jaymin Covy[96]. We'll also use music[97] by Eric Matyas[98] of Soundimage.org.

[92]https://docs.replit.com/images/tutorials/37-breakout-kaboom/gameplay.gif
[93]https://opengameart.org/content/breakout-set
[94]https://opengameart.org/users/buch
[95]https://opengameart.org/content/100-plus-game-sound-effects-wavoggm4a
[96]https://opengameart.org/users/damaged-panda
[97]https://soundimage.org/chiptunes-4/
[98]https://soundimage.org/chiptunes-4/

We've created a single ZIP file with the sprites and sounds you will need for this tutorial, which you can download here[99].

Getting started

Log into your Replit[100] account and create a new repl. Choose **Kaboom** as your project type. Give this repl a name, like "blockbreaker".

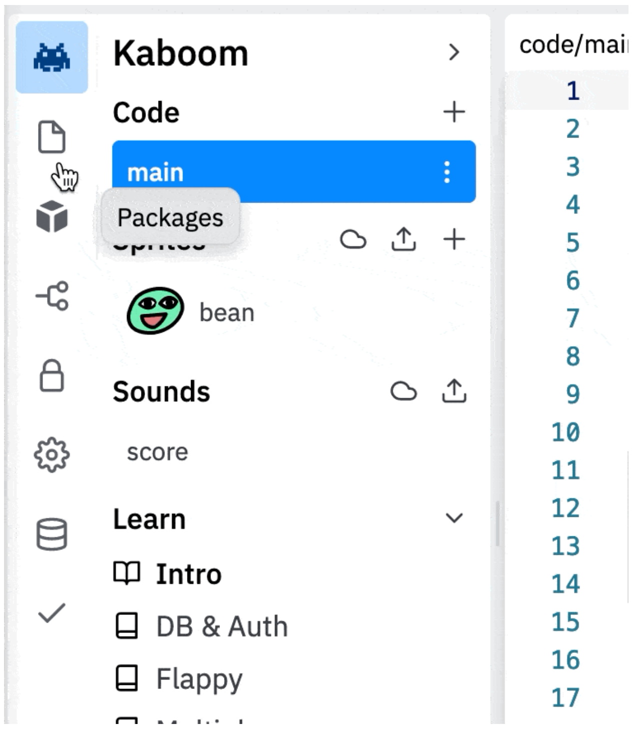


Create a new repl

Kaboom repls are quite different from other kinds of repls you may have seen before: instead of dealing directly with files in folders, you'll be dealing with code, sounds and sprites, the latter of which you can draw directly in Replit's image editor[101].

Before we start coding, we need to upload our sprites and sounds. Download this ZIP file[102] and extract it on your computer. Click the "Files" icon on the sidebar and upload everything in the extracted file's Sounds folder to the "sounds" section of your repl, and everything in the Sprites folder to the "sprites" section of your repl.

[99]https://docs.replit.com/tutorial-files/breakout-kaboom/breakout-resources.zip
[100]https://replit.com/login
[101]https://docs.replit.com/tutorials/kaboom-editor
[102]https://docs.replit.com/tutorial-files/breakout-kaboom/breakout-resources.zip



Uploading assets

Click to open gif[103]

Once you've uploaded the files, you can click on the "Kaboom" icon in the sidebar, and return to the "main" code file.

Loading assets

When you first open your new Kaboom repl, you'll be greeted by a file containing the sample code below.

```
import kaboom from "kaboom";

// initialize context
kaboom();

// load assets
loadSprite("bean", "sprites/bean.png");

// add a character to screen
add([
```

[103] https://docs.replit.com/images/tutorials/37-breakout-kaboom/upload-assets.gif

```
    // list of components
    sprite("bean"),
    pos(80, 40),
    area(),
]);

// add a kaboom on mouse click
onClick(() => {
        addKaboom(mousePos())
})

// burp on "b"
onKeyPress("b", burp)
```

Before we start developing our game, let's remove most of this code, leaving only the following lines:

```
import kaboom from "kaboom";

// initialize context
kaboom();
```

Now we can set the stage for our own game. First, we'll make the game background black and fix the game's screen size by altering the Kaboom context initialization[104]. Add the following to the line `kaboom();`:

```
// initialize context
kaboom({
    width: 768,
    height: 360,
    background: [0,0,0]
});
```

Next, we need to import sprites for our game's objects: the player's paddle, the ball, and the breakable blocks. As the OpenGameArt sprites we're using are all in a single image file, we'll load them using Kaboom's `loadSpriteAtlas()`[105] function. This saves us the hassle of splitting each sprite into its own image file. Add the following code to the bottom of your main code file:

[104]https://kaboomjs.com/#kaboom
[105]https://kaboomjs.com/#loadSpriteAtlas

```
loadSpriteAtlas("sprites/breakout_pieces.png", {
    "blocka": {
        x: 8,
        y: 8,
        width: 32,
        height: 16,
    },
    "blockb": {
        x: 8,
        y: 28,
        width: 32,
        height: 16,
    },
    "blockc": {
        x: 8,
        y: 48,
        width: 32,
        height: 16,
    },
    "blockd": {
        x: 8,
        y: 68,
        width: 32,
        height: 16,
    },
    "paddle": {
        x: 8,
        y: 152,
        width: 64,
        height: 16,
    },
    "ball": {
        x: 48,
        y: 136,
        width: 8,
        height: 8,
    },
    "heart": {
        x: 120,
        y: 136,
        width: 8,
        height: 8,
    }
});
```

Note that we've imported four different block sprites, named `block{a-d}`. Each sprite is a different color and will be worth a different number of points when broken. We've also left most of the sprite sheet's contents untouched – only a few sprites are needed for basic gameplay.

Next, we need to import a font, which we'll use to display the player's score and lives. As Kaboom comes with a number of default fonts we could use, this step is optional, but it will help to give our game a cohesive visual style.

```
loadFont("breakout", "sprites/breakout_font.png", 6, 8,  { chars: "ABCDEFGHIJKLMNOPQ\
RSTUVWXYZ  0123456789:!'" });
```

We've used Kaboom's `loadFont()`[106] function, specifying the name of the font, the image file to source it from, the width and height of individual characters, and the characters it contains. Take a look at the layout of `breakout_font.png` to see the format Kaboom expects. Also note that we will not be able to use any characters outside of the ones represented here – this includes lowercase letters.

Breakout font

Lastly, we need to load our sound effects and music. Add the following code at the bottom to do this:

```
// sounds
loadSound("blockbreak", "sounds/Explosion5.ogg");
loadSound("paddlehit", "sounds/Powerup20.ogg");
loadSound("powerup", "sounds/Powerup2.ogg");
loadSound("ArcadeOddities", "sounds/Arcade-Oddities.mp3");
```

[106]https://kaboomjs.com/#loadFont

Creating levels

We will create two initial levels for our game, using Kaboom's ASCII art level creation[107] functionality. Add the following level definitions to the bottom of your file:

```
// levels
const LEVELS = [
    [
        "                        ",
        "                        ",
        "dddddddddddddddddddddddd",
        "cccccccccccccccccccccccc",
        "bbbbbbbbbbbbbbbbbbbbbbbb",
        "aaaaaaaaaaaaaaaaaaaaaaaa",
        "                        ",
        "                        ",
        "                        ",
        "            .           ",
        "                        ",
        "                        ",
        "                        ",
        "                        ",
        "                        ",
        "                        ",
        "                        ",
        "                        ",
        "                        ",
        "                        ",
        "                        ",
        "                        ",
        "            @           ",
    ],
    [
        " aaaaaaaaaaaaaaaaaaaaaa ",
        " a                    a ",
        " a  bbbbbbbbbbbbbbbbb a ",
        " a  b               b a ",
        " a  b    ccccccc    b a ",
        " a  b  ccdddddddcc  b a ",
        " a  b    ccccccc    b a ",
        " a  b               b a ",
```

[107] https://kaboomjs.com/#addLevel

```
        " a  bbbbbbbbbbbbbbbbb a ",
        " a                    a ",
        " aaaaaaaaaaaaaaaaaaaaaa ",
        "                        ",
        "            .           ",
        "                        ",
        "                        ",
        "                        ",
        "                        ",
        "                        ",
        "                        ",
        "                        ",
        "                        ",
        "                        ",
        "            @           ",
    ],
]
```

In the above levels, `a-d` are blocks, `.` is the ball, and `@` is the player's paddle. We will make these definitions with a `LevelOpt` JSON object, which defines the width and height of individual blocks, and provides definitions for each game object. Add the following code to your file:

```
const LEVELOPT = {
    width: 32,
    height: 16,
    "a": () => [ // block
        sprite("blocka"),
        area(),
        "block",
        "bouncy",
        {
            points: 1
        }
    ],
    "b": () => [ // block
        sprite("blockb"),
        area(),
        "block",
        "bouncy",
        {
            points: 2
        }
    ],
    "c": () => [ // block
        sprite("blockc"),
        area(),
        "block",
        "bouncy",
        {
            points: 4
        }
    ],
    "d": () => [ // block
        sprite("blockd"),
        area(),
        "block",
        "bouncy",
        {
            points: 8
        }
    ],
    "@": () => [ // paddle
        sprite("paddle"),
        area(),
        origin("center"),
        "paddle",
        "bouncy",
        {
            speed: 400
        }
    ],
    ".": () => [ // ball
        sprite("ball"),
        color(WHITE),
        area(),
        origin("center"),
        "ball",
        {
            hspeed: 100,
            vspeed: 50
        }
    ]
}
```

Let’s take a closer look at each of these definitions, starting with the first block object.

```
    "a": () => [ // block
        sprite("blocka"),
        area(),
        "block",
        "bouncy",
        {
            points: 1
        }
    ],
```

A game object definition in Kaboom is a list of components, tags, and custom attributes.

Components are a core part of Kaboom – they provide different functionality to game objects, from an object's appearance to functionality such as collision detection. In this case, we have two components: `sprite()`[108], which tells the object which sprite to represent itself with and `area()`[109], which gives the object the ability to collide with other objects.

While components come with prepackaged behaviors, tags are just labels that we can use to define our own behavior. This object has the tag "block", which we'll use to give it block-specific behaviors, such as being destroyed by the ball and giving the player points. It also has the tag "bouncy", which we'll use to make the ball bounce off it.

Lastly, our block has a custom `points` attribute, which will determine how many points it will give the player when it is destroyed. Our four block objects use different sprites and have different point values, but are otherwise identical.

Next, let's look at the paddle object:

```
    "@": () => [ // paddle
        sprite("paddle"),
        area(),
        origin("center"),
        "paddle",
        "bouncy",
        {
            speed: 400
        }
    ],
```

Like our block objects, the paddle has both `sprite()` and `area()` components. We've also given it the "bouncy" tag, so that the ball will bounce off it. Using tags like this is a great way to avoid writing the same code multiple times. Unlike our block objects, the paddle has an `origin`[110] component, set to "center" – this will allow us to move the object from its center rather than its top-left corner.

[108] https://kaboomjs.com/#sprite
[109] https://kaboomjs.com/#area
[110] https://kaboomjs.com/#origin

The paddle object also has a `speed` attribute, which will determine how fast it moves across the screen.

Our last object is the ball.

```
    ".": () => [ // ball
        sprite("ball"),
        area(),
        origin("center"),
        "ball",
        {
            hspeed: 100,
            vspeed: 50
        }
    ]
```

As a moving object, this is largely similar to the paddle. The main difference is that we give it both horizontal and vertical speed attributes, as it will be moving in all directions, whereas the paddle only moves left and right.

Now that we've defined our level layouts and the objects that will populate them, we can create our game scene[111]. In Kaboom, a scene is a unique screen with its own objects and game rules. We can use scenes to separate gameplay from menus and information screens, or even to separate different kinds of minigames in the same project. As scenes can take arguments, we can use a single "game" scene to represent all of our levels. Add the following code below your `LEVELOPT` definition:

```
scene("game", ({levelIndex, score, lives}) => {

    addLevel(LEVELS[levelIndex], LEVELOPT);

});
```

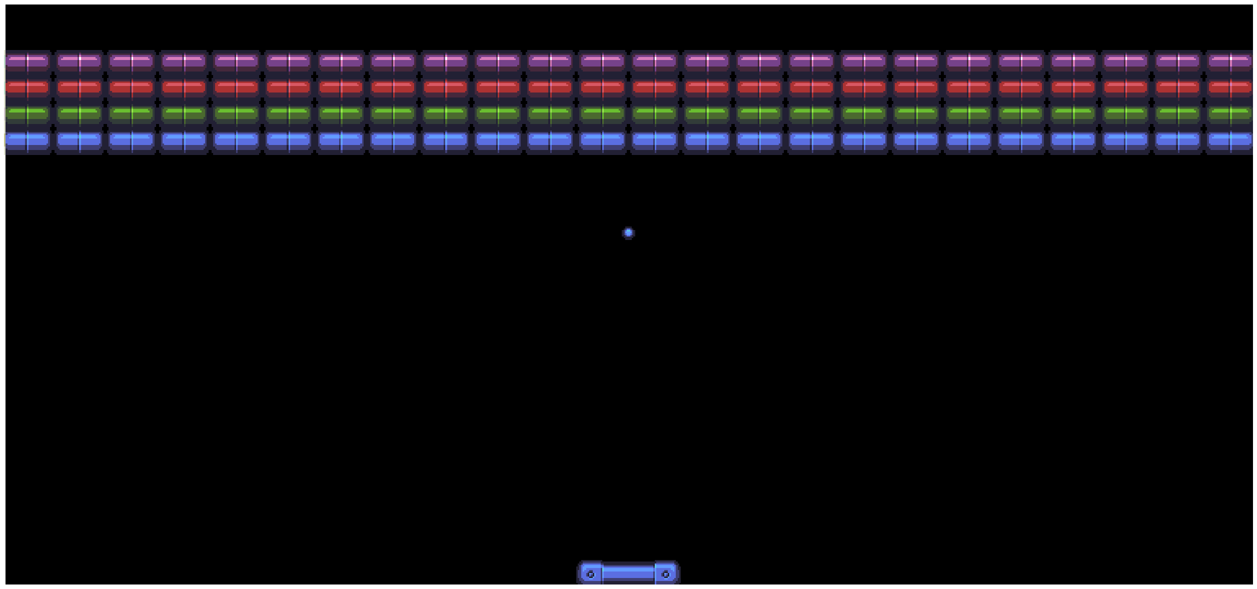
In addition to providing the level number as an argument (`levelIndex`), we provide both `score` and `lives`. This will allow us to preserve both values when the player defeats one level and moves to the next.

Then add this code to the bottom of the main code file to define and call game start function. This function will `go()`[112] to the first level, setting the player's score to `0` and their lives to `3`. In addition to calling it when the game first loads, we will call this function when we need to restart after a game over.

[111] https://kaboomjs.com/#scene
[112] https://kaboomjs.com/#go

```
// start game on first level
function start() {
    go("game", {
        levelIndex: 0,
        score: 0,
        lives: 3,
    });
}

start();
```

Run your repl now. You should see our first level, with its colorful blocks, paddle and ball, frozen in amber. In the next section, we'll add some motion.


First level

Moving the paddle

Let's write some code to control the player's paddle. First, we need to retrieve a reference to the paddle using `get()`[113]. We'll place this code inside the "game" scene, below `addLevel`:

```
    // player's paddle
    const paddle = get("paddle")[0];
```

Now we'll add code to move the paddle left and right. We could do this with the left and right arrow keys, which would give our game a retro feeling, but most modern browser-based block-breaking

[113] https://kaboomjs.com/#get

games have a mouse-controlled paddle. Moreover, as Kaboom automatically translates touch events to mouse events, implementing mouse controls will make our game playable on mobile devices without keyboards. So let's add some code to have our paddle follow the mouse cursor:

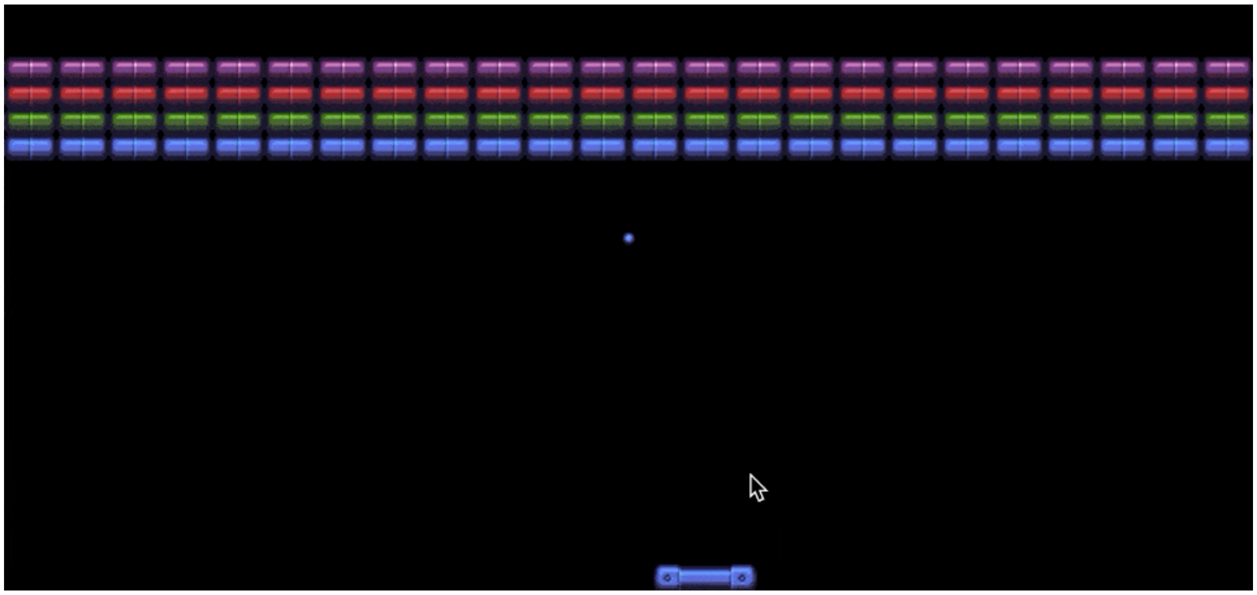
```
    // mouse controls
    onUpdate(() => {
        if (mousePos().x > 0 && mousePos().x < width() && mousePos().y > 0 && mouseP\
os().y < height()) {
            if (mousePos().x < paddle.worldArea().p1.x) { // left
                paddle.move(-paddle.speed, 0);
            }
            else if (mousePos().x > paddle.worldArea().p2.x) { // right
                paddle.move(paddle.speed, 0);
            }
        }
    });
```

This code will run every frame[114]. First, it checks whether the mouse cursor is inside the game area. Then it checks whether the cursor is to the paddle's left or right, and moves the paddle in that direction. The paddle won't move if it is vertically in line with the cursor.

Note the use of `worldArea()`. This method, provided by the `area()` component, returns an object containing two sets of X- and Y-coordinates, `p1` and `p2`. The first set, `p1`, is the top-left corner of the object's collision mask, and `p2` is its bottom-right corner. By default, an object's collision mask is a rectangle of the same size as its sprite – Kaboom does not support non-rectangular collision masks.

Rerun your repl now and try out the controls. If you have a touch screen on your device, you can also move the paddle by tapping or dragging your finger.

[114]https://kaboomjs.com/#onUpdate



Moving the paddle

Click to open gif[115]

Moving the ball

Now that we can move the paddle, we need the ball to move too. Add the following code to your file with the "game" scene:

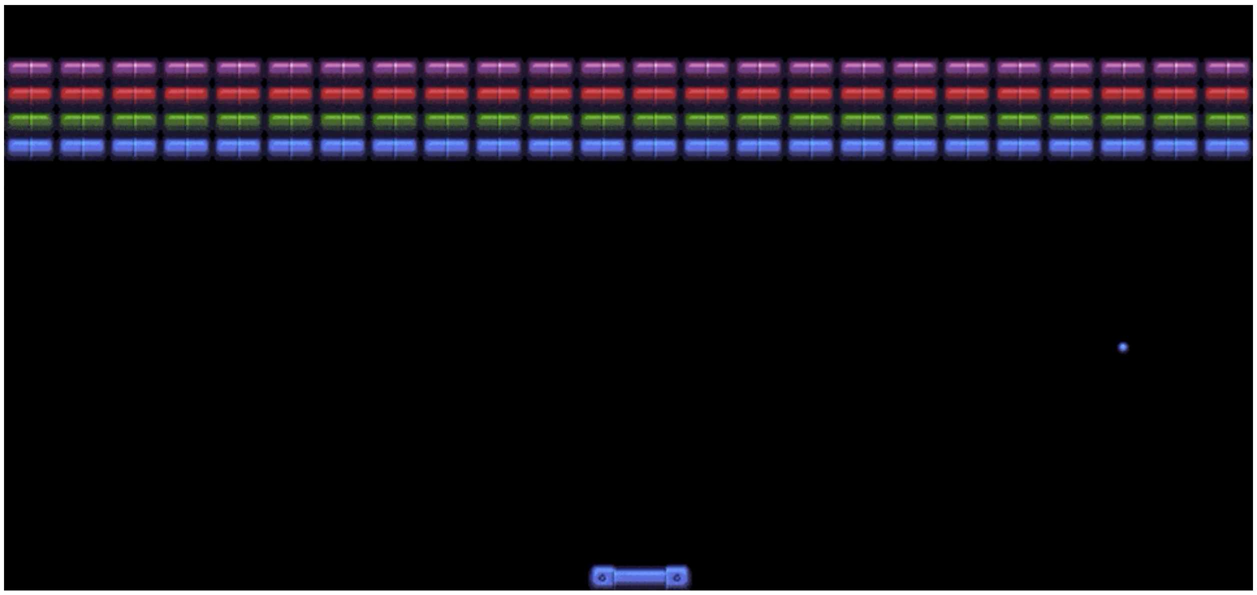
```
    // ball movement
    onUpdate("ball", (ball) => {
        ball.move(ball.hspeed, ball.vspeed);
    });
```

You'll recall that we set the ball's `hspeed` and `vspeed` in its object definition. Run your repl now, and watch as the ball flies off the screen. Our game won't last very long if the ball can leave the screen like this, so we need to add some code to have it bounce off the edges. Alter your `onUpdate("ball")` callback to match the following:

[115] https://docs.replit.com/images/tutorials/37-breakout-kaboom/move-paddle.gif

```
    onUpdate("ball", (ball) => {
        // bounce off screen edges
        if (ball.worldArea().p1.x < 0 || ball.worldArea().p2.x > width()) {
            ball.hspeed = -ball.hspeed;
        }

        if (ball.worldArea().p1.y < 0 || ball.worldArea().p2.y > height()) {
            ball.vspeed = -ball.vspeed;
        }

        // move
        ball.move(ball.hspeed, ball.vspeed);
    });
```

If the ball goes off the left or right edges of the screen, we reverse its horizontal direction, and if it goes off the top or bottom of the screen, we reverse its vertical direction. Run your repl now to see this effect in action.


Moving the ball

Click to open gif[116]

Collisions

Now that the ball can move and bounce off the screen's edges, we need it to also bounce off the paddle and the blocks. To achieve this, we'll write an `onCollide()`[117] event handler for the tags

[116]https://docs.replit.com/images/tutorials/37-breakout-kaboom/move-ball.gif
[117]https://kaboomjs.com/#onCollide

"ball" and "bouncy". Add the following code to the "game" scene, below your ball movement code:

```
    // collisions
    onCollide("ball", "bouncy", (ball, bouncy) => {
        ball.vspeed = -ball.vspeed;

        if (bouncy.is("paddle")) { // play sound
            play("paddlehit");
        }
    });
```

Note that we're only changing the vertical direction of the ball, because that's the important one for our gameplay. While we could implement more complex bounce physics by taking into account which sides the collision occurred on, changing the vertical direction alone gets us the type of ball movement players expect from a block-breaking game.

Now that the player can actually hit the ball with their paddle, we shouldn't have the ball bounce off the bottom of the screen anymore. Find your `onUpdate("ball")` callback and remove second condition from the second `if` statement. Your callback should now look like this:

```
    // ball movement
    onUpdate("ball", (ball) => {
        // bounce off screen edges
        if (ball.worldArea().p1.x < 0 || ball.worldArea.p2().x > width()) {
            ball.hspeed = -ball.hspeed;
        }

        if (ball.worldArea().p1.y < 0) { // <-- second condition removed
            ball.vspeed = -ball.vspeed;
        }

        // move
        ball.move(ball.hspeed, ball.vspeed);
    });
```

The other important collision event that we need to implement is having the ball destroy blocks it hits. Add the following code below the `onCollide("ball", "bouncy")` callback.

```
    onCollide("ball", "block", (ball, block) => {
        block.destroy();
        score += block.points;
        play("blockbreak"); // play sound
    });
```

Here we use the `destroy()`[118] function to remove the block object from play, and then increment our score by the block's points value.

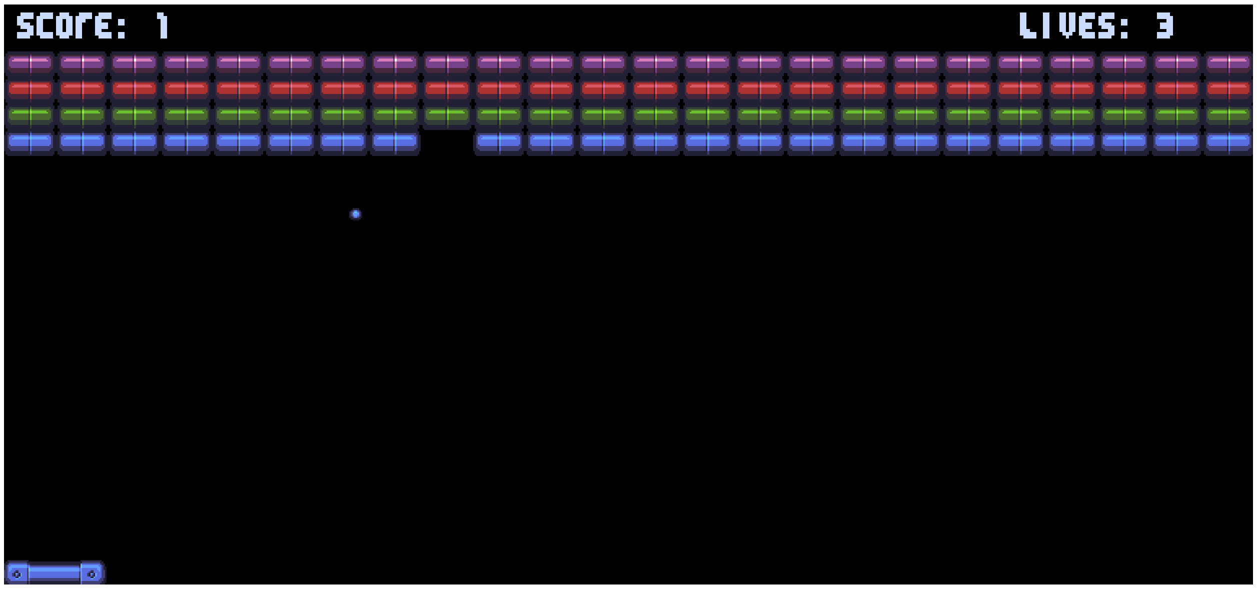
Now that we're changing the `score` variable, it's important that we display it on the screen, along with `lives`. Add the following code to the "game" scene, below your collision code:

```
    // ui
    onDraw(() => {
        drawText({
            text: `SCORE: ${score}`,
            size: 16,
            pos: vec2(8,8),
            font: "breakout",
            color: WHITE
        });
        drawText({
            text: `LIVES: ${lives}`,
            size: 16,
            pos: vec2(width()*13/16, 8),
            font: "breakout",
            color: WHITE
        });
    });
```

We've added an `onDraw()`[119] callback, which will run every frame, after all `onUpdate()` callbacks have run. The `onDraw()` callbacks are the only place we can use drawing functions such as `drawText()`[120]. Also note that we've used the font we defined at the start of this tutorial.

Run your repl now, and you should be able to hit the ball with your paddle, destroy blocks, and get points. Our core gameplay is implemented.

[118]https://kaboomjs.com/#destroy
[119]https://kaboomjs.com/#onDraw
[120]https://kaboomjs.com/#drawText



Game play

Winning and losing

As it stands, our game is both unforgiving and unrewarding. If you let the ball go off the bottom of the screen, it's permanently gone, and you have to refresh your browser to try again. If you manage to destroy all the blocks, the game continues without moving to the next level or acknowledging your victory.

Let's fix these deficiencies now by implementing lives, as well as win and lose conditions. We'll implement lives in the `onUpdate("ball")` callback that deals with ball movement. Find this callback and add the following new code just below the second `if` statement:

```
  onUpdate("ball", (ball) => {
    // bounce off screen edges
    if (ball.worldArea().p1.x < 0 || ball.worldArea().p2.x > width()) {
      ball.hspeed = -ball.hspeed;
    }

    if (ball.worldArea().p1.y < 0) {
      ball.vspeed = -ball.vspeed;
    }

    // fall off screen -- NEW CODE BELOW
    if (ball.pos.y > height()) {
      lives -= 1;
      if (lives <= 0) {
        go("lose", { score: score });
      }
      else {
        ball.pos.x = width()/2;
        ball.pos.y = height()/2;
      }
    }
    // END OF NEW CODE

    // move
    ball.move(ball.hspeed, ball.vspeed);
});
```

This code checks whether the ball has fallen of the screen, and if so, decrements `lives`. If there are lives remaining, it moves the ball back to the middle of the screen. Otherwise, it sends the player to the “lose” scene, which we’ll define soon. But first, we need to provide for the game’s win condition.

We’ll consider a level won once all the blocks have been destroyed. To determine this, we can check whether the number of blocks in the level is `0`. We’ll put this check in the ball and block collision callback, after the block is destroyed. Find this code and alter it to resemble the following:

```
    onCollide("ball", "block", (ball, block) => {
        block.destroy();
        score += block.points;
        play("blockbreak");

        // level end -- NEW CODE BELOW
        if (get("block").length === 0) { // next level
            if (levelIndex < LEVELS.length) {
                go("game", {
                    levelIndex: levelIndex+1,
                    score: score,
                    lives: lives
                    });
            }
            else { // win
                go("win", { score: score });
            }
        }
    });
```

Now we need to create our “win” and “lose” scenes. Add the following code for both scenes below the “game” scene and above the `start` function definition:

```
// gameover screens
scene("lose", ({ score }) => {

        add([
                text(`GAME OVER\n\nYOUR FINAL SCORE WAS ${score}`, {
            size: 32,
                        width: width(),
            font: "breakout"
                }),
                pos(12),
        ]);

        add([
                text(`PRESS ANY KEY TO RESTART`, {
            size: 16,
                        width: width(),
            font: "breakout"
                }),
                pos(width()/2, height()*(3/4)),
        ]);

        onKeyPress(start);
        onMousePress(start);
});

scene("win", ({ score }) => {

        add([
                text(`CONGRATULATIONS, YOU WIN!\n\nYOUR FINAL SCORE WAS ${score}`, {
            size: 32,
                        width: width(),
            font: "breakout"
                }),
                pos(width()/2, height()/2),
        ]);

        add([
                text(`PRESS ANY KEY TO RESTART`, {
            size: 16,
                        width: width(),
            font: "breakout"
                }),
                pos(width()/2, height()*(3/4)),
        ]);

        onKeyPress(start);
        onMousePress(start);
});
```

These scenes are quite similar to each other: each one displays some text, including the player's final score, and prompts the player to press any key. Both `onKeyPress(start)` and `onMousePress(start)` will call the `start` function if any keyboard key or mouse button is pressed, or if the screen is tapped on a touch device.

Run your repl now. You should now be able to play through both levels of our block-breaking game (or lose and reach the game over screen).

Game over screen

Powerups

There's one more sprite we loaded at the start of the tutorial that we haven't used yet – the heart. This will be a powerup. We'll have it randomly appear in place of destroyed blocks and start falling. If the player catches it with their paddle, they will gain an additional life.

Find your `onCollide("ball", "block")` code and add the new code specified below:

```
    // collisions
    onCollide("ball", "block", (ball, block) => {
        block.destroy();
        score += block.points;
        play("blockbreak");

        // level end
        if (get("block").length === 0) { // next level
            if (levelIndex < LEVELS.length) {
                go("game", {
                    levelIndex: levelIndex+1,
                    score: score,
                    lives: lives
                    });
            }
            else { // win
                go("win", { score: score });
            }
        }

        // powerups -- NEW CODE BELOW
        if (chance(0.05)) { // extra life
            add([
                sprite("heart"),
                pos(block.pos),
                area(),
                origin("center"),
                cleanup(),
                "powerup",
                {
                    speed: 80,
                    effect() { lives++; }
                }
            ]);
        }
    });
```

This code uses Kaboom's `chance()`[121] function to set our powerup to spawn after a block is destroyed 5% of the time. The powerup is a game object with similar components to other moving objects we've defined. The `cleanup()`[122] component will ensure it's automatically destroyed when it leaves the screen. Additionally, we give it a movement speed and an `effect()` function, which we'll call

[121]https://kaboomjs.com/#chance
[122]https://kaboomjs.com/#cleanup

when it hits the paddle.

Just below the `onCollide("ball", "block")` callback, add the following two callbacks to define our powerup's movement and collision behavior:

```
    // powerups
    onUpdate("powerup", (powerup) => {
        powerup.move(0, powerup.speed);
    });

    paddle.onCollide("powerup", (powerup) => {
        powerup.effect();
        powerup.destroy();
        play("powerup");
    });
```

Run your repl now and you should see the occasional extra life powerup as you play.

Music

As a final touch, we'll add some music to our game. Near the bottom of your file, just above the invocation of `start()`, add the following code:

```
// play music
const music = play("ArcadeOddities");
music.loop();
```

`loop()` will ensure that the music plays continuously.

Where next?

We've built a simple but polished block-breaking game. From here, you might want to make the following additions:

- Extra levels.
- New powerups, such as a longer paddle, multiple balls, or even a paddle-mounted laser gun.
- Bigger and smaller blocks, blocks that can take multiple hits, or moving blocks.

Code

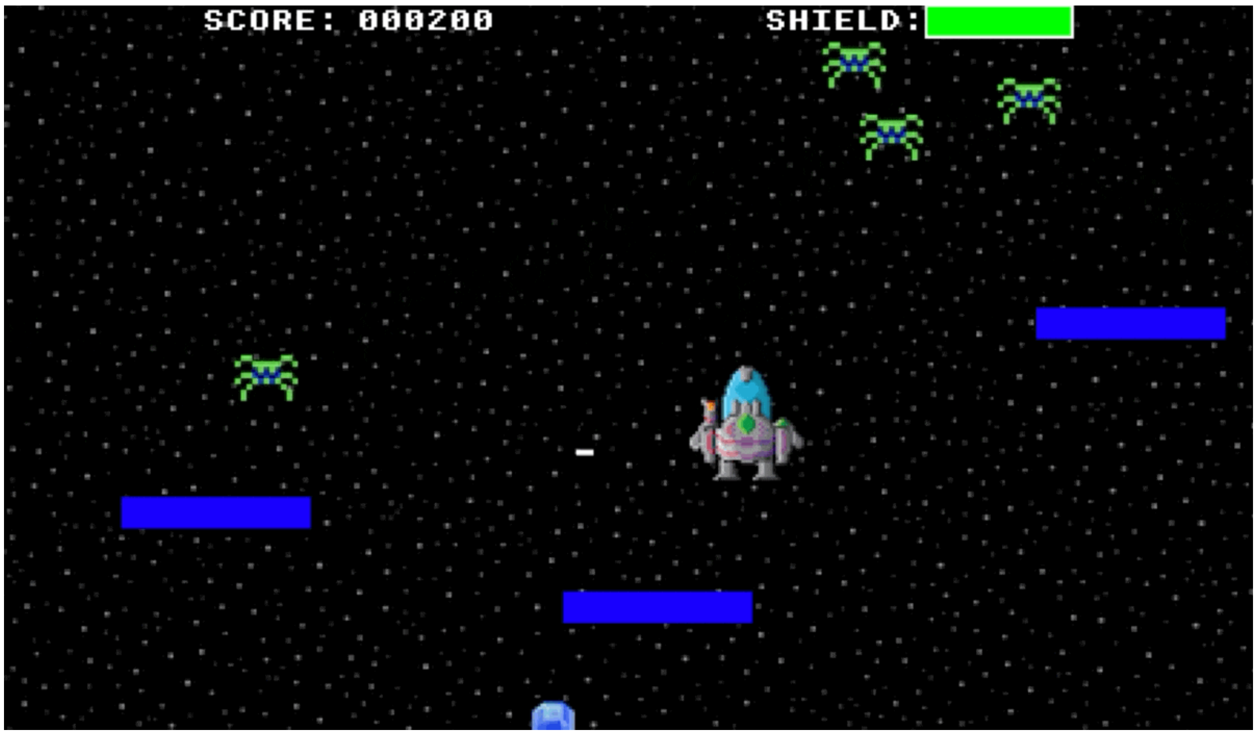
You can find the code for this tutorial on Replit[123]

[123]https://replit.com/@ritza/blockbreaker

Build a Space Shooter with Kaboom.js

In this tutorial, we'll build a space shooter game with a platformer feel. We'll use Kaboom.js[124] for the game engine, and we'll code it using Replit[125] online IDE (Integrated Development Environment).

Here's how the game will look when we're done:



The finished game

Click to open gif[126]

You can download this zip file[127] with all the sprites and sounds you'll need for this tutorial.

Game Design

Here's what we're aiming for in this game:

- Fast action: the player moves around a lot.

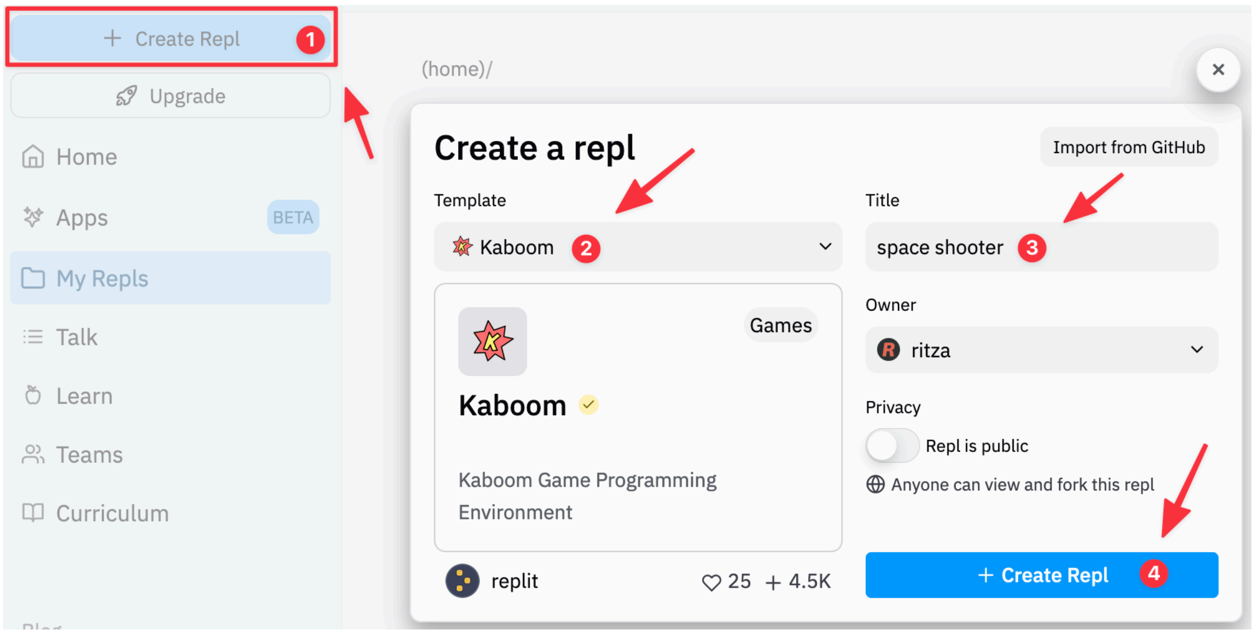
[124]https://kaboomjs.com
[125]https://replit.com
[126]https://docs.replit.com/images/tutorials/24-space-shooter-kaboom/gameplay.gif
[127]https://docs.replit.com/tutorial-files/space-shooter-kaboom/space-shooter-resources.zip

- Good sound effects: to immerse the player in the game and contribute to the overall game vibe.
- Lots of shooting opportunities.
- Increasing challenge: the game gets harder and faster as the player gets better.

In our game, a player flies a spaceship around a faraway planet, collecting gems and dodging or shooting alien bugs that explode on contact. The spaceship will lose shield strength each time an alien bug hits it. With every 1000 points the player earns, the game gets faster and more bugs appear.

Creating a New Project on Replit

Let's head over to Replit[128] and create a new repl. Choose **Kaboom** as your project type. Give this repl a name, like "Space Shooter".



Template: Kaboom

Title: space shooter

Owner: ritza

Privacy: Repl is public

Anyone can view and fork this repl

Creating an 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.

Getting Started with Kaboom.js

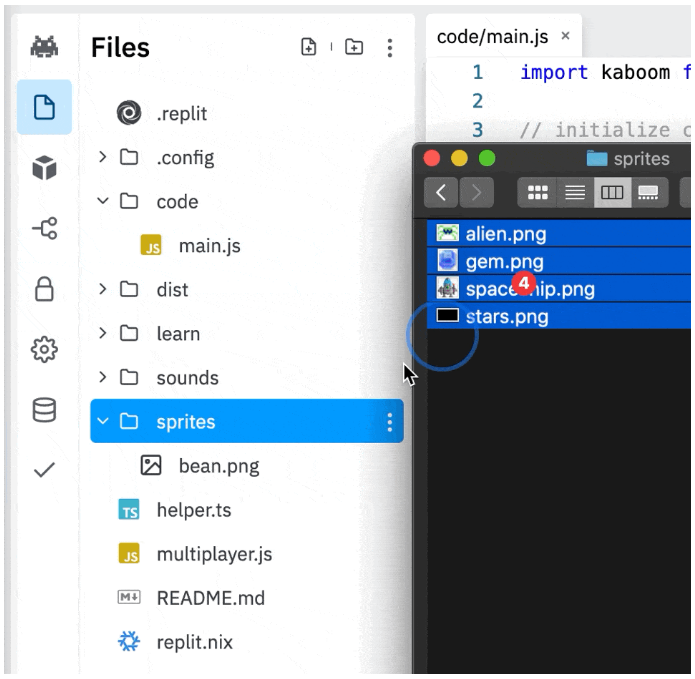
Kaboom.js[129] is a JavaScript library that contains a lot of useful features for making simple browser games. It has functionality to draw shapes and sprites (the images of characters and game elements)

[128] https://replit.com
[129] https://kaboomjs.com

to the screen, get user input, play sounds, and more. We'll use some of these features in our game to explore how Kaboom works.

The Replit Kaboom interface is specialised for game-making. Besides the Space Invader icon, you'll notice a few special folders in the file tray, like "Code", "Sprites", and "Sounds". These special folders take care of loading up assets, and all the necessary code to start scenes and direct the game. You can read up more about this interface here[130].

If you haven't already, download this zip file[131] which contains all the sprites and sounds for the game. Extract the file on your computer, then add the sprites to the "Sprites" section in the Replit editor, and the sounds to the "Sounds" section.



```
import kaboom f

// initialize c
```

Uploading sprites

Click to open gif[132]

Kaboom makes good use of JavaScript's support for callbacks[133]: instead of writing loops to read keyboard input and check if game objects have collided, Kaboom uses an event model that tells us when these events have happened. We can then write callback functions[134] to act on these events.

A Kaboom game is made up of "scenes", which are like levels, or different parts and stages of a game. Scenes have multiple "layers", allowing us to have game backgrounds, main game objects (like the player, bullets, enemies, etc), and UI elements (like the current score, health, etc).

[130]https://docs.replit.com/tutorials/kaboom
[131]https://docs.replit.com/tutorial-files/space-shooter-kaboom/space-shooter-resources.zip
[132]https://docs.replit.com/images/tutorials/24-space-shooter-kaboom/upload-sprites.gif
[133]https://developer.mozilla.org/en-US/docs/Glossary/Callback_function
[134]https://developer.mozilla.org/en-US/docs/Glossary/Callback_function

Setting up Kaboom

To start, we need to set up Kaboom with the screen size and colors we want for the game window. Replace the code in `main.js` with the code below:

```
import kaboom from "kaboom";

kaboom({
  background: [0, 0, 0],
  width: 440,
  height: 275,
  scale: 1.5
});
```

Here we import the kaboom library, and then initialize the context by calling `kaboom({ ... })`. We also set the size of the view to 440x275 pixels and `scale` the background by a factor of `1.5` on screen. Now, let's load up the sprites and sounds we will need in this game. This code loads each of the graphic elements we'll use, and gives them a name, so we can refer to them when we build the game characters:

```
loadRoot("sprites/");
loadSprite("stars", "stars.png");
loadSprite("gem", "gem.png");
loadSprite("spaceship", "spaceship.png");
loadSprite("alien", "alien.png");

loadRoot("sounds/");
loadSound("shoot","shoot.wav");
loadSound("score","score.wav");
loadSound("music","music.mp3");
loadSound("explosion","explosion.wav");
```

The first line, `loadRoot`[135], specifies which folder to load all the sprites and game elements from, so we don't have to keep typing it in for each sprite. Then each line loads a game sprite and gives it a name so that we can refer to it in code later. We use similar code to load the sounds we will need in this game, but instead of `loadSprite`[136] we use `loadSound`[137] to load sounds.

Adding the main game scene

Kaboom "scenes"[138] allow us to group logic and levels together. In this game we'll have 2 scenes:

[135]https://kaboomjs.com/#loadRoot
[136]https://kaboomjs.com/#loadSprite
[137]https://kaboomjs.com/#loadSound
[138]https://kaboomjs.com/#scene

- A "main" scene, which will contain the game levels and all the logic to move the spaceship.
- An "endGame" scene which will display when the game is over.

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

  layers([
    "bg",
    "obj",
    "ui",
  ], "obj");


  add([
    sprite("stars"),
    layer("bg")
  ]);

  // todo.. add main scene code here

});

go("main");
```

We define the scene using the `scene`[139] function. This function takes a string as the scene name – we're calling the scene "main".
Then we create 3 layers: "background" (`bg`), "object" (`obj`) and "user interface" (`ui`). The `obj` layer is set as the default layer. We then add the stars sprite to the background layer.

Finally, we use the `go`[140] function to go to the main scene when the game starts up.

Note The code snippets in the sections that follow have to be added within the body of the main scene unless specified otherwise.

Creating the Game Map

Let's get a scene layout, or map, drawn on the screen. This will define the ground and platforms in the game.

Kaboom has built-in support for defining game maps using text and the function `addLevel`[141]. This takes away a lot of the hassle normally involved in loading and rendering maps.

The code below creates the game map. Add it to the `main.js` file, within the main scene (below the code to add the stars sprite to the background layer).

[139]https://kaboomjs.com/#scene
[140]https://kaboomjs.com/#go
[141]https://kaboomjs.com/doc/#addLevel

```
// Game Parameters
const MAP_WIDTH = 440;
const MAP_HEIGHT = 275;
const BLOCK_SIZE = 11;

const map = addLevel([
  "--------------------------------------------",
  "-                                          -",
  "-                                          -",
  "-                                          -",
  "-                                          -",
  "-                                          -",
  "-                                          -",
  "-                                          -",
  "-                                          -",
  "-                                          -",
  "-                                pppppp    -",
  "-                                          -",
  "-                                          -",
  "-                                          -",
  "-                                          -",
  "-                                          -",
  "-   pppppp                                 -",
  "-                                          -",
  "-                                          -",
  "-                 pppppp                   -",
  "-                                          -",
  "-                                          -",
  "-                                          -",
  "-                                          -",
  "============================================",
  "                                            "
], {
  width: BLOCK_SIZE,
  height: BLOCK_SIZE,
  pos: vec2(0, 0),
  "=": ()=> [
    rect(BLOCK_SIZE, BLOCK_SIZE),
    color(150,75,0),
    "ground",
    area(),
    solid()
  ],
  "p": ()=> [
    rect(BLOCK_SIZE, BLOCK_SIZE),
    color(0,0,255),
    "platform",
    area(),
    solid()
  ],
  "-": ()=> [
    rect(BLOCK_SIZE/10, BLOCK_SIZE),
    color(0,0,0),
    "boundary",
    area(),
    solid()
  ],
});
```

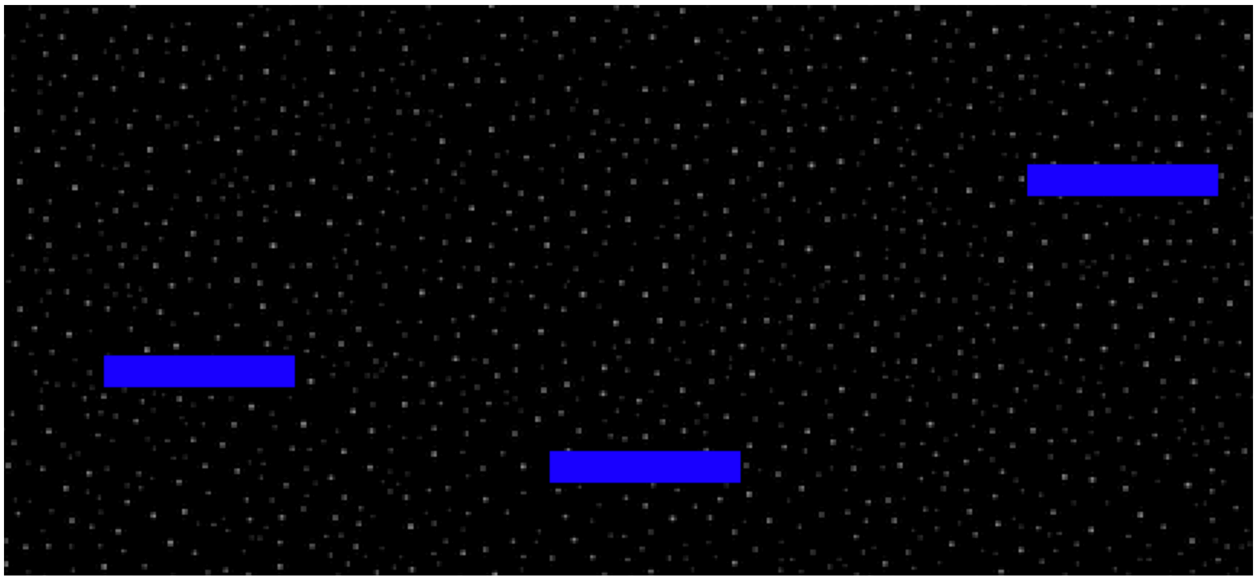
First, we add some game parameters, which we'll use when we define the size of the map, and the default block size for map elements.

Next, we create the game map. The map, or level design, is expressed in an array of strings. Each row in the array represents one row on the screen, so we can design visually in text what the map should look like. The width and height parameters specify the size of each of the elements in the map. The `pos` parameter specifies where on the screen the map should be placed – we chose `0,0`, which is the top left of the screen, as the starting point for the map.

Kaboom allows us to specify what to draw for each symbol in the text map. You can make maps out of different elements, e.g. a symbol for a wall, a symbol for ground, a symbol for a hump, and so on. To tell Kaboom what to draw for the symbol, we add the symbol as a key, for example `=`, and then specify parameters for it.

In this code, we have 3 different type of fixed map elements: `=` representing the ground, `p` representing platforms, and `-` representing the invisible boundaries of the screen. Each of the map elements has a tag (`ground`, `platform`, `boundary`) which is the string name grouping the individual pieces together. This allows us to refer to them collectively later.

If we run the code, we should see the game map, like this:



Game map

Adding the Spaceship

Let's add the spaceship using the `add`[142] function:

```javascript
const player = add([
  sprite("spaceship"),
  pos(100, 200),
  body(),
  area(),
  scale(1),
  rotate(0),
  origin("center"),
  "player",
  {
    score : 0,
    shield : 100
  }
]);
```

The `add`[143] function constructs a game object using different components, e.g. `pos`, `body`, `scale`, etc. Each of these components gives the object different features.

[142]https://kaboomjs.com/doc/#add
[143]https://kaboomjs.com/doc/#add

Notably, the `body`[144] component makes the object react to gravity: the spaceship falls if it's not on the ground or a platform. The `rotate`[145] component allows us to tilt the spaceship in the direction the player wants to go, providing good visual feedback. By default, all operations are calculated around the top left corner of game objects. To make the tilt work correctly, we add the `origin`[146] component and set it to `center`, so that the tilt adjusts the angle from the center of the object.

Kaboom also allows us to attach custom data to a game object. We've added `score` to hold the player's latest score, and `shield` to hold the percentage of the ship's protection shield still available. We can adjust these as the player picks up items or crashes into aliens.

When we created the `map` earlier, we added the `solid`[147] component to map objects. This component marks objects as solid, meaning other objects can't move past them.

Moving the Spaceship

We'll allow a few different moves for the spaceship: change direction left or right and fly up. We also need to keep track of which way the spaceship is facing, so that we'll know which side to shoot lasers from later.

To handle the changing and tracking of direction, add the following code:

```javascript
const directions = {
  LEFT: "left",
  RIGHT: "right"
}

let current_direction = directions.RIGHT;

onKeyDown("left", () => {
    player.flipX(-1);
    player.angle = -11;
    current_direction = directions.LEFT;
    player.move(-100,0);
});

onKeyDown("right", () => {
    player.flipX(1);
    player.angle = 11;
    current_direction = directions.RIGHT;
    player.move(100,0);
```

[144] https://kaboomjs.com/doc/#body
[145] https://kaboomjs.com/doc/#rotate
[146] https://kaboomjs.com/doc/#origin
[147] https://kaboomjs.com/doc/#solid

```
});


onKeyRelease("left", ()=>{
    player.angle = 0;
});

onKeyRelease("right", ()=>{
    player.angle = 0;
});
```

First, we create a constant object defining the directions our game allows. Then we create a variable to track the `current_direction` the spaceship is facing.

Then we add the key-handling code. The key names `left` and `right` refer to the left and right arrow keys on the keyboard. Kaboom provides the `onKeyDown`[148] event, which lets us know if a certain key is being pressed. We create `onKeyDown` event handlers for each of the arrow keys. As long as the given key is held down, `onKeyDown` calls the event handler repeatedly.

The code inside each `onKeyDown` event does the following:

- The `flipX` function mirrors the player's spaceship image so that it looks different depending on the direction it is facing. We use `-1` to flip it to appear facing the left, `1` the right.
- The function `player.angle` slightly tilts the spaceship while the key is being held down. This is so the spaceship looks like it is about to move in the given direction.
- The `current_direction` tracking variable is updated. We'll use this variable when we add shooting.
- The `move` function moves the spaceship in the given direction.

We also have `onKeyRelease` event handlers for the left and right keys. These reset the spaceship's tilt angle to `0` (i.e. straight up) when the ship is no longer moving in that direction.

Now we want to have the spaceship fly up when we press the `up arrow` key. To do this, we'll take advantage of Kaboom's `jump`[149] attribute (which is part of the `body`[150] component) and repurpose it for flying up. Add the following code to the main scene:

```
onKeyDown("up", () => {
      player.jump(100);
});
```

[148]https://kaboomjs.com/#onKeyDown
[149]https://kaboomjs.com/doc/#body
[150]https://kaboomjs.com/doc/#body

Adding Laser Guns

Because the game takes place in outer space, the weapon of choice is a laser gun. We’ll need to add functions to create the bullet when the player fires, and to control the direction of the bullets. We’ll also need to add another key handler to check when the player presses a key to “fire”, which is the space key in this game.

```
const BULLET_SPEED = 400;

onKeyPress("space", () => {
        spawnBullet(player.pos);
});

function spawnBullet(bulletpos) {
    if (current_direction == directions.LEFT){
        bulletpos = bulletpos.sub(10,0);
    } else if (current_direction == directions.RIGHT){
        bulletpos = bulletpos.add(10,0);
    }
        add([
                rect(6, 2),
                pos(bulletpos),
                origin("center"),
        color(255, 255, 255),
        area(),
                "bullet",
        {
            bulletSpeed : current_direction == directions.LEFT?-1*BULLET_SPEED: BULL\
ET_SPEED
        }
        ]);

    play("shoot", {
                volume: 0.2,
                detune: rand(-1200, 1200),
        });
};
```

First, we add a constant `BULLET_SPEED` to define the speed at which the laser “bullets” fly across the screen. Then we use the `onKeyPress`[151] event to trigger the shooting. Notice `onKeyPress` only calls the event handler once as the key is pressed, unlike the `onKeyDown` event we used for moving. This

[151]https://kaboomjs.com/#onKeyPress

is because it's more fun if the player needs to bash the "fire" button as fast as possible to take down an enemy, rather than just having automatic weapons.

The `onKeyPress` handler calls the `spawnBullet` function with the player's current position. This function handles creating a new laser shot in the correct direction. The first few lines of the method adjust the bullet's starting position a little to the left or right of the spaceship's position. This is because the position of the spaceship that gets passed to the function is the center of the spaceship (remember the `origin` component we added to it earlier). We adjust it a little so that the bullet looks like it is coming from the edge of the spaceship.

Then we add a new bullet object to the game using the `add`[152] function. We don't use a sprite for the bullet, but draw a `rect`[153], or rectangle, with our given color. We tag it `bullet` so we can refer to it later when detecting if it hit something. We also give it a custom property, `bulletSpeed`, which is the distance and direction we want the bullet to move on each frame.

Finally, we add sound effects when the player shoots. The `play`[154] function plays our "shoot.wav" file. We adjust the volume down a bit, so it fits in better with the overall sound mix. We use the `detune` parameter along with a random number generator, `rand`[155], to change the pitch of the sound each time it's played. This is so the sound doesn't become too repetitive and also because it sounds weird and "spacey".

Now that we've set up the bullet, we need to make it move on each frame. To do this we can use the `onUpdate`[156] event, using the `bullet` tag to identify the objects we want to update:

```
onUpdate("bullet", (b) => {
        b.move(b.bulletSpeed,0);
        if ((b.pos.x < 0) ||(b.pos.x > MAP_WIDTH)) {
                destroy(b);
        }
});
```

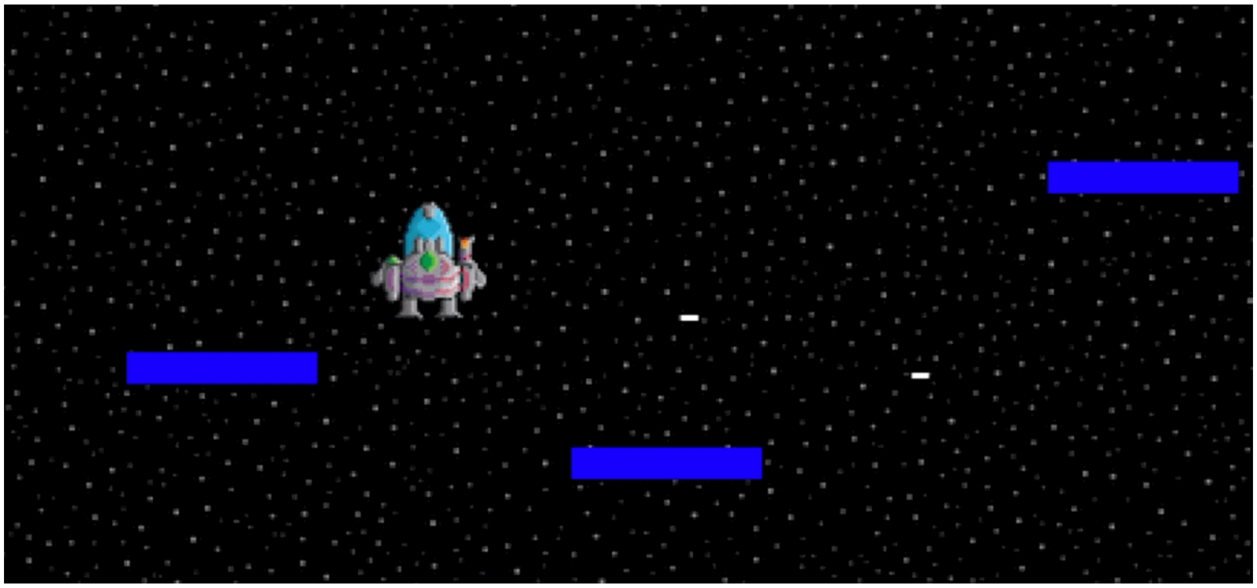
With each frame, the action event updates the objects with the matching tag, in this case `bullet`. We call `move`[157] on the bullet, using the custom value for `bulletSpeed` that we assigned to it on creation. We also check to see if the bullet has gone off the screen, and if it has, we `destroy`[158] it.

We also need to destroy the bullet if it hits a platform. We can do this using the Kaboom `onCollide`[159] event. Add the following code:

[152]https://kaboomjs.com/doc/#add
[153]https://kaboomjs.com/doc/#rect
[154]https://kaboomjs.com/doc/#play
[155]'https://kaboomjs.com/doc/#rand'
[156]https://kaboomjs.com/doc/#onUpdate
[157]https://kaboomjs.com/doc/#pos
[158]https://kaboomjs.com/doc/#destroy
[159]https://kaboomjs.com/doc/#onCollide

```
onCollide("bullet","platform", (bullet, platform) =>{
    destroy(bullet);
});
```

Run the code now, and you should be able to shoot.



Laser firing

Click to open gif[160]

Adding Alien Space Bugs

Now that we have a spacecraft, and it can shoot, we need something to shoot at. Let's add some hostile exploding alien space bugs. We'll want to have them coming in a relatively constant stream to keep the game challenging. We also want them coming in from different sides and angles to keep the player on their toes. We'll add a new function to control the creation of alien space bugs:

[160]https://docs.replit.com/images/tutorials/24-space-shooter-kaboom/laser-firing.gif

```
const ALIEN__BASE_SPEED = 100;
const ALIEN_SPEED_INC = 20;

function spawnAlien() {
    let alienDirection = choose([directions.LEFT, directions.RIGHT]);
    let xpos = (alienDirection == directions.LEFT ? 0:MAP_WIDTH);

    const points_speed_up = Math.floor(player.score / 1000);
    const alien_speed = ALIEN__BASE_SPEED + (points_speed_up * ALIEN_SPEED_INC);
    const new_alien_interval = 0.8 - (points_speed_up/20);

        add([
                sprite("alien"),
                pos(xpos, rand(0, MAP_HEIGHT-20)),
        area(),
                "alien",
                 {
                         speedX: rand(alien_speed * 0.5, alien_speed * 1.5) * (alienDirection == di
ns.LEFT ? 1: -1),
            speedY: rand(alien_speed * 0.1, alien_speed * 0.5) * choose([-1,1])
                 },
        ]);

        wait(new_alien_interval, spawnAlien);
}

spawnAlien();
```

We create 2 parameters for the alien's speed: a base rate and an incremental rate. Each time the player gains another 1000 points, we'll add to the incremental rate.

Tip: You can put these parameters and all the others we have defined at the top of the file, so that they are easy to find and adjust if you want to tweak the game parameters later.

Then we define the `spawnAlien` function. To randomly choose the side of the screen the alien will fly in from, we use the Kaboom `choose`[161] function, which picks an element at random from an array. From the chosen direction, we can determine the alien's starting position on the `x axis` (horizontal plane).

Then we go into the calculation to figure out the speed that the alien should move at. First, we check if we need to increase the alien's speed based on the player's score. We divide the player's score by 1000 (since the aliens' speed increases with every 1000 points the player earns). We get rid of decimals by using the `Math.floor`[162] function, which is built into JavaScript. The result is our

[161]https://kaboomjs.com/doc/#choose
[162]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/floor

`points_speed_up` value.

Next we take the `ALIEN_BASE_SPEED` and add the incremental rate multiplied by our `points_speed_up` value.

We also calculate a new rate at which aliens are spawned, making the aliens not only faster at moving, but also faster at respawning.

Now that we've calculated our basic parameters, we create a new alien using the `add`[163] function again:

- `sprite('alien')` creates the alien with the image `alien`.
- `pos(xpos, rand(0, MAP_HEIGHT-20))` sets the starting position of the alien. We calculated the `x pos` from the randomly chosen direction. We also add a random `y` (vertical) position for the alien, between the top (position `0`) of the map, and the bottom (`MAP_HEIGHT`) of the map (screen co-ordinates start from the top left of the screen). We remove `20` pixels from the bottom bounds, to account for the ground.
- We add the `"alien"` tag to the object, so we can identify and call it in other parts of the code.
- We also add a custom object with the speed of this particular alien, broken into it's speed along the `x` and `y` axis. For the speed along the x-axis `speedX`, we add a random component so that not all aliens move at exactly the same speed. Then we multiply the speed by -1 or 1 depending on whether the alien is meant to be moving left or right across the screen.

Finally, we use Kaboom's `wait`[164] function to wait a short amount of time before calling `spawnAlien` again to create a new alien. We also have a call to `spawnAlien` to get it started when the game starts.

Moving the Aliens

To move the aliens, we'll create a handler to attach to the `onUpdate` event, which fires for each alien object on every frame, like we did for the bullets:

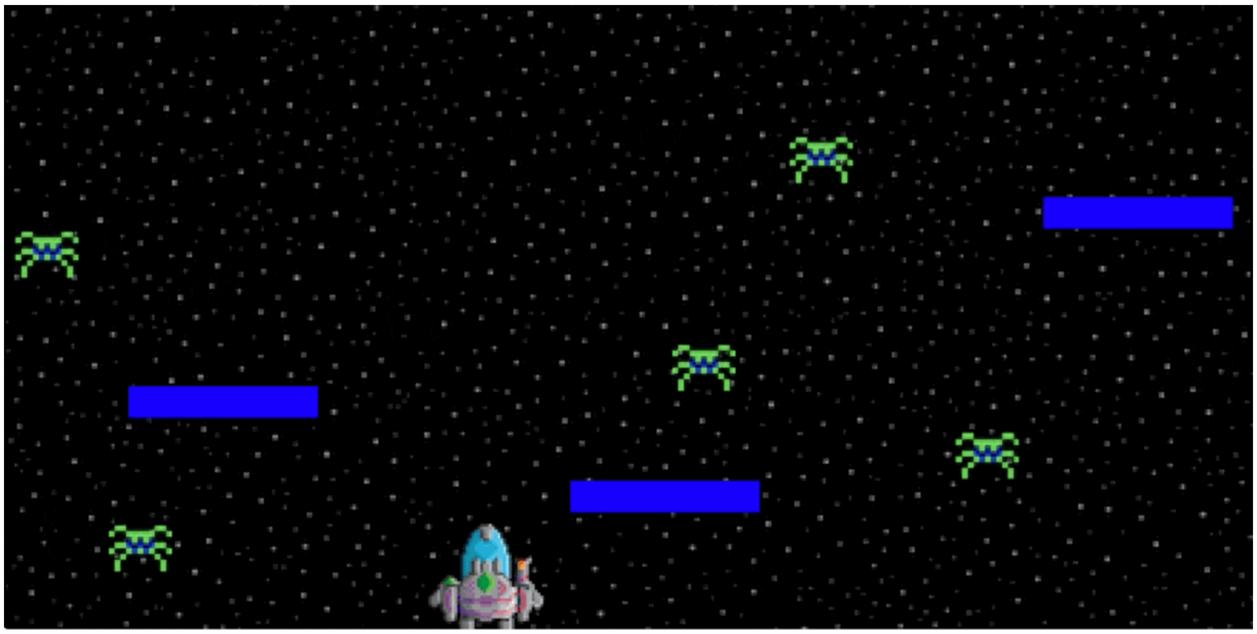
```
onUpdate("alien", (alien) => {
	alien.move(alien.speedX, alien.speedY);

	if ((alien.pos.y - alien.height > MAP_HEIGHT) || (alien.pos.y < 0)) {
		destroy(alien);
	}
    if ((alien.pos.x < -1 * alien.width) ||(alien.pos.x > MAP_WIDTH)) {
		destroy(alien);
	}
});
```

[163]https://kaboomjs.com/doc/#add
[164]https://kaboomjs.com/doc/#wait

First, the function moves the alien by the amount we calculated earlier and saved to the alien's custom data.

Then the function checks to see if the alien has moved out of bounds of the map area. If it has, we destroy it, as it is no longer visible. Having too many active objects can decrease performance, so this step is important.

Run the code now, you should see moving aliens.



Aliens

Click to open gif[165]

Shooting the Aliens

Now that we have moving aliens, a moving spaceship, and laser bullets, let's add the code to deal with a laser bullet hitting an alien. Of course, we want this to have a cool explosion and sound effect to give good feedback to the player.

[165]https://docs.replit.com/images/tutorials/24-space-shooter-kaboom/aliens.gif

```
onCollide("alien","bullet", (alien, bullet) =>{
    makeExplosion(alien.pos, 5, 5, 5);
    destroy(alien);
    destroy(bullet);
    play("explosion", {
                volume: 0.2,
                detune: rand(0, 1200),
        });
});
```

This is similar to the code used before to check if a bullet has hit a platform. We `destroy`[166] both the bullet and alien to remove them from the scene. Then we use the `play`[167] function to play the explosion sound effect. We set the volume so it fits in the mix, and we also put a random detune (pitch adjust) on the sound, to vary it and make it more interesting when a lot of aliens are being shot at.

We also call out to a function to create an explosion around the area where the alien bug used to be. This code is from the "shooter" example on the Kaboom examples page[168] (which is a great game). It makes a series of bright white flashes around the explosion site, giving a cool cartoon or comic-book-like feel to the explosions. Add this code:

```
function makeExplosion(p, n, rad, size) {
                for (let i = 0; i < n; i++) {
                        wait(rand(n * 0.1), () => {
                                for (let i = 0; i < 2; i++) {
                                        add([
                                                pos(p.add(rand(vec2(-rad), vec2(rad)))),
                                                rect(1, 1),
                                                color(255,255,255),
                                                origin("center"),
                                                scale(1 * size, 1 * size),
                                                grow(rand(48, 72) * size),
                                                lifespan(0.1),
                                        ]);
                                }
                        });
                }
}

function lifespan(time) {
                let timer = 0;
```

[166]https://kaboomjs.com/doc/#destroy
[167]https://kaboomjs.com/doc#playhttps://kaboomjs.com/doc/#play
[168]https://kaboomjs.com/demo/#shooter

```
                return {
                        update() {
                                timer += dt();
                                if (timer >= time) {
                                        destroy(this);
                                }
                        },
                }
}

function grow(rate) {
    return {
        update() {
            const n = rate * dt();
            this.scale.x += n;
            this.scale.y += n;
        },
    };
}
```

The `makeExplosion` function has four arguments (inputs to the function). These are:

- `p`, the center position to base the explosions around
- `n`, the number of main flashes to make
- `rad`, the radius or distance from `p` to make the flashes in
- `size`, the size of each of the flashes

The function creates a `for loop`[169] to loop for `n` times (the number of main flashes we want to make). It uses the Kaboom `wait`[170] function to leave a little bit of time (0.1) seconds between each main flash.

Another `for loop` loops twice to create 2 sub flashes, using the Kaboom `add`[171] function to add a `rectangle`[172] shape for each flash, and setting the color to bright white (color components in Kaboom go from 0-1). This rectangle starts out at 1 pixel in each dimension. Then the `scale`[173] component is added to increase the size of the flash to the `size` passed in to the function - we'll use this later when we "grow" the explosion. The `origin`[174] component is used to set the origin of the rectangle to it's center - this will be used when we "grow" the flash to give the impression that it is starting from a small point and exploding. We set the origin as the center so that scale is calculated from this position, giving it a more natural feel.

[169] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for
[170] https://kaboomjs.com/doc/#wait
[171] https://kaboomjs.com/doc/#add
[172] https://kaboomjs.com/doc/#rect
[173] https://kaboomjs.com/doc/#scale
[174] https://kaboomjs.com/doc/#origin

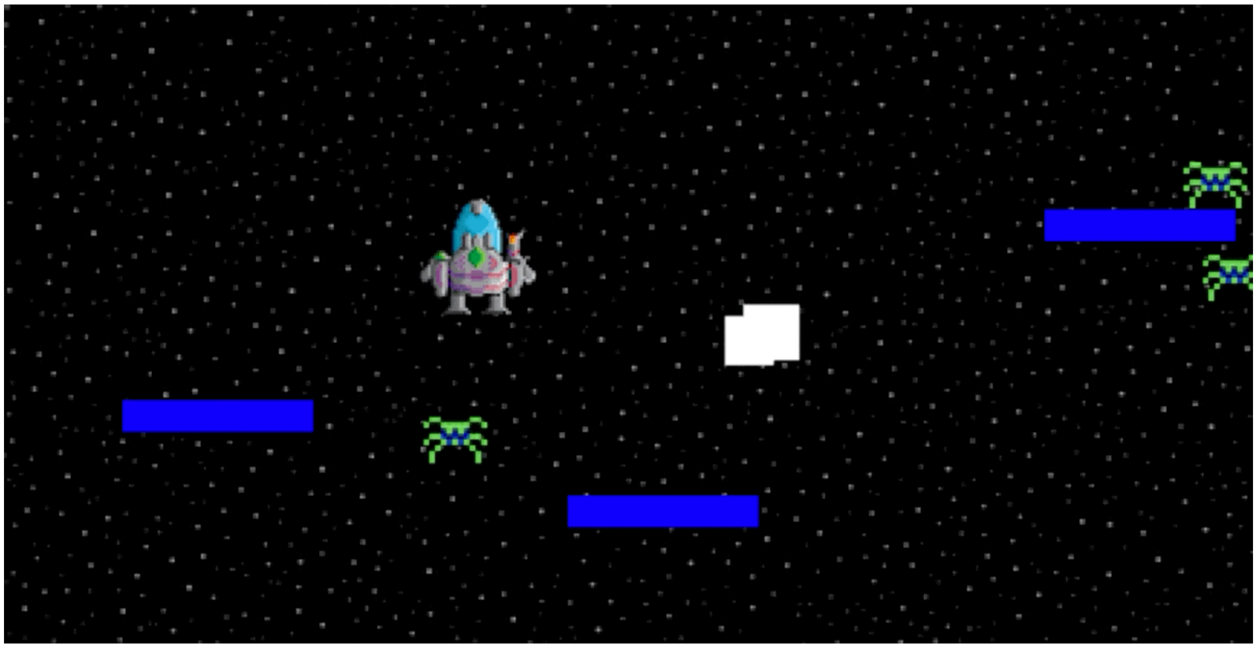
To make the flashes appear around the position `p` that we specified, the `pos`[175] component is adjusted by a random amount, ranging from `-rad` to `rad`, the radius we specified (in other words, the blast area).

Then there are references to two custom components - `lifespan` and `grow`. Kaboom allows us to define our own components to give objects any behaviour or attributes we want. All we need to do is create a function that returns an object with a method called `update`, which is then called for each frame of the object the component is added to.

Let's first look at the custom component `grow`. This is used to create the effect of the flash expanding outwards, like a firework explosion starting at a small point and getting larger until it disappears. In `grow`'s `update` function, the object is scaled up (available because we used the `scale`[176] component on the object) on each frame. This is calculated from the `rate` passed in - which is the size the object should grow per second, multiplied by the time difference from the last frame, using the Kaboom `dt`[177] function, which provides that time difference in seconds for us. The explosion flash will keep on growing in each frame, so we need a way to end the explosion before it covers the entire screen.

This brings us to the `lifespan` component. This is implemented to automatically `destroy`[178] the object after a short time, to solve the ever-growing explosion problem. It works by having a `timer` variable, which is updated each frame with the difference in time from the last frame, using the Kaboom `dt`[179] function again. When the `timer` count is more than the `time` parameter passed into the component, the object is automatically `destroyed`[180]. This creates the impression of a quick explosion blast.

[175] https://kaboomjs.com/doc/#pos
[176] https://kaboomjs.com/doc/#scale
[177] https://kaboomjs.com/doc/#dt
[178] https://kaboomjs.com/doc/#destroy
[179] https://kaboomjs.com/doc/#dt
[180] https://kaboomjs.com/doc/#destroy



Shooting Aliens

Click to open gif[181]

Exploding the Alien Bugs on Contact

When the alien bugs hit something solid, they should explode. To do this, we'll add the following code:

```
onCollide("alien","platform", (alien, platform) =>{
    makeExplosion(alien.pos, 5, 3, 3);
    destroy(alien);
    play("explosion", {
                volume: 0.1,
                detune: rand(-1200, 1200),
        });
});

onCollide("alien","ground", (alien, ground) =>{
    makeExplosion(alien.pos, 5, 3, 3);
    destroy(alien);
    play("explosion", {
                volume: 0.1,
                detune: rand(-1200, 1200),
```

[181] https://docs.replit.com/images/tutorials/24-space-shooter-kaboom/shooting-aliens.gif

```
        });
});
```

Here we have 2 collision handlers: one for aliens hitting a platform, and one for aliens hitting the ground. They both do the same thing. First, since we have a great explosion creating function, we use it gratuitously. Then we `destroy`[182] the alien object to remove it from the scene. Finally, we play an explosion sound effect at a lower volume, as this explosion is not caused by the player and doesn't directly affect them. We also add the usual random `detune`[183] function to modify the sound each time and keep it interesting.

Adding Score and Shield UI

Let's add the UI to show the ship's shield health and the player's overall score.

First, add text for the player's score:

```
add([
        text("SCORE: ", { size: 8, font: "sink"}),
        pos(100, 10),
        origin("center"),
        layer("ui"),
]);

const scoreText = add ([
        text("000000", { size: 8, font: "sink"}),
        pos(150, 10),
        origin("center"),
        layer("ui"),
]);
```

Here we add two new objects, rendered with the `text`[184] component. The first is just the static label for the score. The second is the text placeholder for the actual score. Note that the `layer`[185] component is used in both cases to place the text on the UI layer we created at the start of the tutorial. We haven't had to specify the layer for all our other game objects, because we set the `obj` layer as the default to use when we defined the layers.

Now that we have the UI components for showing the score, we need a function to update the score when it changes, and reflect it on the UI.

[182]https://kaboomjs.com/doc/#destroy
[183]https://kaboomjs.com/doc/#play
[184]https://kaboomjs.com/doc/#text
[185]https://kaboomjs.com/doc/#layer

```
function updateScore(points){
    player.score += points;
    scoreText.text = player.score.toString().padStart(6,0);
    play("score", {
                volume: 0.5,
                detune: rand(-1200, 1200),
        });
}
```

This `updateScore` function takes as its argument the number of points to add to the score and adds them to the player's current score - remember we added `score` as a custom property when we created the player (spaceship) object.

Next we update the `scoreText` UI element we created previously. The player's score is converted to a string using JavaScript's `toString`[186] method, which is part of every object in JavaScript. It is also modified with `padStart`[187], which makes sure the resulting score string is exactly `6` digits long, using 0s to put in front of the string (`start`) if the number is smaller than `6` digits long. This makes nice placeholders for the score, and gives a cue to the users as to the maximum score they could reach. Finally, we play a little sound to indicate that points have been earned. As before, we vary the pitch each time using `detune`[188] to keep the sound fresh.

To increment the score when the alien bugs get hit by a bullet, update the `onCollide` event we added earlier for a bullet and alien bug as follows:

```
onCollide("alien","bullet", (alien, bullet) =>{
    makeExplosion(alien.pos, 5, 5, 5);
    destroy(alien);
    destroy(bullet);
    play("explosion", {
                volume: 0.2,
                detune: rand(0, 1200),
        });
    updateScore(10);  // new line
});
```

The next UI element to add is the ship's shield health. This would be great as a kind of health-bar-style display, that starts out green and turns red when the shield is low. The game should end when the shield is fully depleted, as the spaceship is then totally destroyed.

[186]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/toString
[187]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/padStart
[188]https://kaboomjs.com/doc/#play

```
add([
        text("SHIELD: ", { size: 8, font: "sink"}),
        pos(300, 10),
        origin("center"),
        layer("ui"),
]);

const shieldHolder = add ([
    rect(52,12),
        pos(350, 10),
    color(100,100,100),
        origin("center"),
        layer("ui"),
]);

const shieldHolderInside = add ([
    rect(50,10),
        pos(350, 10),
    color(0,0,0),
        origin("center"),
        layer("ui"),
]);

const shieldBar = add ([
    rect(50,10),
        pos(325, 5),
    color(0,255,0),
        layer("ui"),
]);
```

First, we add a text label so that players know what the bar represents. To create the shield bar UI, we use 3 elements :

- A border, or `shieldHolder`, to outline the bar.
- A black inner block to make the holder look like a thin line, `shieldHolderInside`.
- The `shieldBar` itself, which will get shorter as the shield is damaged.

Now we need a function to call when we want to update the shield's health:

```
function updatePlayerShield(shieldPoints){
    player.shield += shieldPoints;
    player.shield = Math.max(player.shield, 0);
    player.shield = Math.min(player.shield, 100);

    shieldBar.width = 50 * (player.shield / 100);

    if (player.shield < 20) shieldBar.color = rgb(255,0,0);
    else if (player.shield < 50) shieldBar.color = rgb(255,127,0);
    else shieldBar.color = rgb(0,255,0);

    if (player.shield <=0){
        destroy(player);
        for (let i = 0; i < 500; i++) {
            wait(0.01 *i, ()=>{
                makeExplosion(vec2(rand(0,MAP_WIDTH,), rand(0, MAP_HEIGHT)), 5, 10, \
10);
                play("explosion", {
                    detune: rand(-1200, 1200)
                });
            });
        }
        wait(2, ()=>{
            go("endGame");
        });
        }
}
```

This function has an argument for the number of `shieldPoints` to update the shield by and adjusts the custom `shield` property on the UI layer. It also clamps the minimum and maximum amount the shield can be to between 0 and 100.

The function sets the width of the `shieldBar` (its dimension along the x axis) to the percentage of the shield available (`player.shield / 100`), multiplied by the full width of the bar, `50`.

Then the function updates the color of the bar depending on the health of the shield:

- Less than 20% health, shield bar is red;
- Less than 50% but more than 20% health, shield bar is orange;
- The shield bar is set to green for all other health values, in other words, when health is over 50%.

The final step in the shield health function is to check if the shield health is depleted, and end the game if it is.

When the game ends, we destroy the spaceship to remove it from the scene. Now we have another opportunity to create some more explosions using the makeExplosion function we added earlier. This time we can go really big! To make a big impact, we create a for loop to set off 500 explosions all over the screen for seriously dramatic effect. We use the Kaboom wait[189] function to have a small delay between each explosion so that they don't all go off at once. Then we make each explosion happen at random positions on the map, passing in other parameters to the makeExplosion function to set the blast radius, number of sub-explosions and general size. We also play the explosion sound effect using Kaboom's play[190] function. This time we don't adjust the volume down, as we want the sound to be as dramatic as possible. We detune it randomly again to create a true cacophony and sense of mayhem.

After setting off all those sound effects and visual fireworks, we wait[191] for 2 seconds for everything to settle down, and then use the Kaboom function go[192] to switch to a new scene, endGame, and wait for the player to play again. Add the code below to create the endGame scene, at the bottom of main.js below the line go("main"):

```
scene("endGame", ()=> {
  const MAP_WIDTH = 440;
  const MAP_HEIGHT = 275;

  add([
    text("GAME OVER ", {size: 40, font: "sink"}),
    pos(MAP_WIDTH / 2, MAP_HEIGHT / 3),
    origin("center"),
    layer("ui"),
  ]);


  onKeyRelease("enter", ()=>{
    go("main");
  });
});
```

This scene adds[193] a large "GAME OVER" text over the screen until the player presses and releases the enter key. Then the onKeyRelease[194] event returns the player to the main scene, and uses go[195] to switch scenes and restart the game. Because this is a new scene, in a new scope, we need to add the MAP_WIDTH and MAP_HEIGHT constants again.

[189]https://kaboomjs.com/doc/#wait
[190]https://kaboomjs.com/doc#play
[191]https://kaboomjs.com/doc/#wait
[192]https://kaboomjs.com/doc/#go
[193]https://kaboomjs.com/doc/#add
[194]https://kaboomjs.com/#onKeyRelease
[195]https://kaboomjs.com/doc/#go

Allowing the Alien Bugs to Attack

Now that we have mechanisms for updating points and shield health, we can add the code dealing with alien bugs hitting the spaceship to the main scene:

```
const ALIEN_SHIELD_DAMAGE = -15;

onCollide("alien", "player", (alien, player) =>{
    shake(20);
    makeExplosion(alien.pos, 8, 8, 8);
    destroy(alien);
    play("explosion",
    {
      detune: -1200,
      volume : 0.5
    });
    updatePlayerShield(ALIEN_SHIELD_DAMAGE);
});
```

This is a big event - it's the way the ship shield gets damaged and it can be fatal - so we want to add a bit more dramatic effect. Kaboom can create a cool screen-shaking effect, as if the player has been hit, which we can invoke by calling `shake`[196] with a number representing how dramatic the shake should be. Then we add some visual effect with the `makeExplosion` function. We also destroy the alien and `play`[197] the `explosion` effect again, this time a bit louder as the alien exploding has directly affected the player. We also detune the effect to the lowest pitch we can, to make it "feel" more direct, particularly if the player has a sub-woofer.

Then we call the `updatePlayerShield` function we defined previously, with a constant that defines by how much a shield is damaged per hit. You can move the constant to the top of the main scene file to keep it neat if you want.

Raining Gems

It's time to add the element that gives the game its purpose: gems the player can collect to earn points. Add this function to the main scene to create a gem:

[196]https://kaboomjs.com/doc/#shake
[197]https://kaboomjs.com/doc/#play

```
function spawnGem(){
    let xpos = rand(BLOCK_SIZE, MAP_WIDTH - BLOCK_SIZE);
    add([
                sprite("gem"),
                pos(xpos, BLOCK_SIZE),
        area(),
        body(),
                "gem"
        ]);
}

onUpdate("gem", (gem)=>{
    if (gem.pos.y > MAP_HEIGHT) {
        destroy(gem);
        spawnGem();
    }

});

spawnGem();
```

On this weird planet in outer space, the gems rain from the sky, which is the top of the map for our purposes. We calculate a random position, `xpos`, along the `x` axis for the gem to appear on by calling the Kaboom `rand`[198] function. We don't want the gems to fall right at the edge of the screen, as they will be cut off and the spaceship won't be able to get to them because of the `boundary` elements we added all around the screen. So we limit the random `xpos` to one `BLOCK_SIZE` from each edge.

Now we `add`[199] the gem sprite to the scene. The `pos` component is set to the `xpos` we calculated, with the `y` component set to one `BLOCK_SIZE` from the top of the screen. This is to avoid the gem getting stuck on our upper `boundary`. We also give the gem the `body`[200] component, which makes it subject to Kaboom gravity so that it falls down towards the ground. We give it the label `gem` so that we can refer to it later.

Then we add the `onUpdate`[201] event handler for the gem - we need to do this for all objects with a `body` component so that interactions with `solid`[202] objects are taken care of. Sometimes, if the frame rate gets too low (if there's a lot of action, or the computer's slow), some `body` and `solid` interaction maybe missed, and the `object falls through the solid`[203]. This could cause gems to fall through the ground, out of reach of the player's spaceship. To account for this possibility, we check if the gem's `y` position is beyond the bounds of the map, and destroy it and spawn a new gem if it is.

[198]https://kaboomjs.com/doc/#rand
[199]https://kaboomjs.com/doc/#add
[200]https://kaboomjs.com/doc/#body
[201]https://kaboomjs.com/doc/#onUpdate
[202]https://kaboomjs.com/doc/#solid
[203]https://github.com/replit/kaboom/issues/86

Finally, we call `spawnGem()` to start the gem raining process.

Collecting Gems

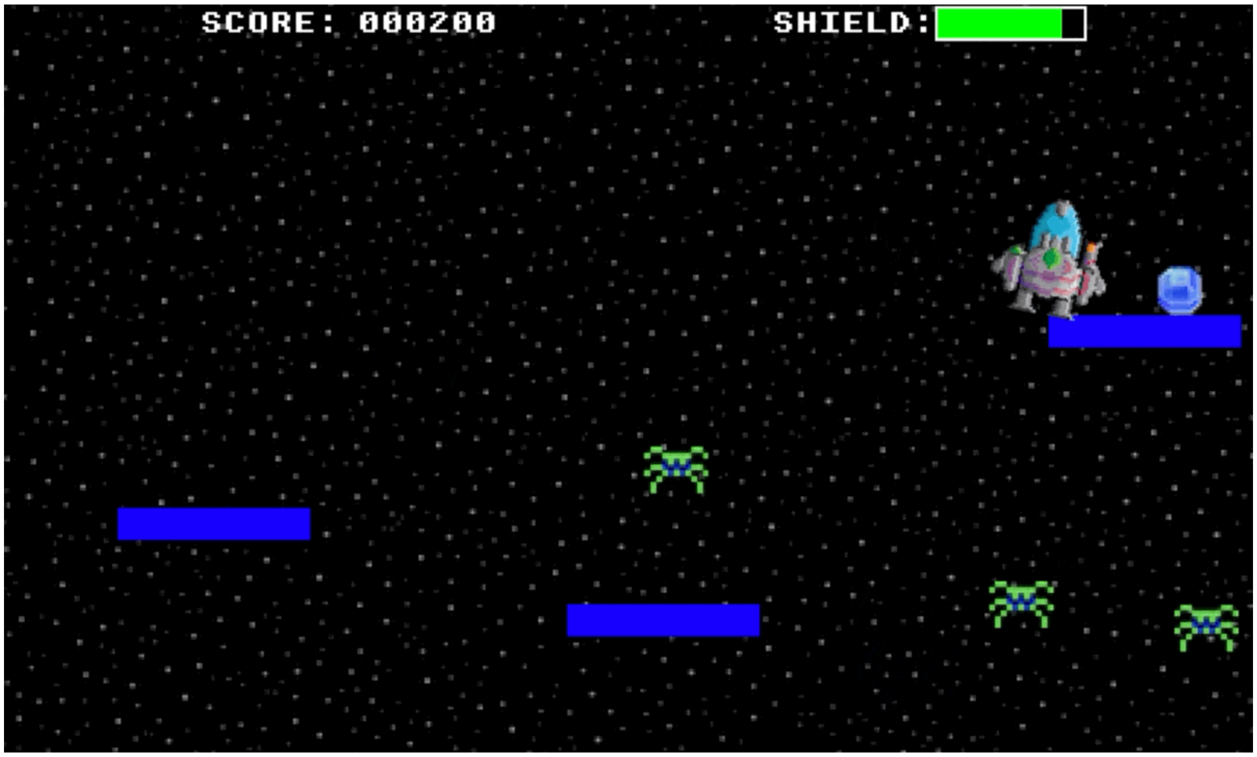
Now that gems are raining down, we can add a handler to pick up when the player's spaceship moves over a gem. This is how the spaceship "collects" gems, and will earn the player points. Add the following `onCollide`[204] event handler:

```
const POINTS_PER_GEM = 100;

player.onCollide("gem", (gem) => {
  destroy(gem);
  updateScore(POINTS_PER_GEM);
  wait(1, spawnGem);
});
```

This fires whenever the spaceship and a gem collides. We `destroy`[205] the gem to remove it from the scene, and call the `updateScore` function we added earlier to update the player's points by the amount declared in the `POINTS_PER_GEM` constant. Then we `wait`[206] one second before another gem is spawned for the player to collect.

Run the code now and start collecting gems.

[204]https://kaboomjs.com/doc/#onCollide
[205]https://kaboomjs.com/doc/#destroy
[206]https://kaboomjs.com/doc/#wait



Collecting gems

Click to open gif[207]

Adding Background Music

Having sound effects is cool, but games generally need a soundtrack to tie all the sounds together. Kaboom allows us to play a sound file on loop as constant background music. Add this code to the bottom of `main.js` file to play the track:

```
const music = play("music");
music.loop();
```

The music is a track called “Battle of Pogs” by “Komiku” from “Free music archive”[208], a good resource for music that you can legally use in your games.

Playing the Game

Congratulations, you’ve finished making this Kaboom game! Try running and playing the game to see what score you can get. You can also experiment with adjusting the parameters to see how they change the gameplay.

[207]https://docs.replit.com/images/tutorials/24-space-shooter-kaboom/collecting-gems.gif
[208]https://freemusicarchive.org/music/Komiku/Captain_Glouglous_Incredible_Week_Soundtrack/pog

Credits

The game art and sounds used in this tutorial are from the following sources:

- Music : https://freemusicarchive.org/music/Komiku/Captain_Glouglous_Incredible_Week_-Soundtrack/pog[209]
- Laser : https://freesound.org/people/sunnyflower/sounds/361471/[210]
- Explosion: https://freesound.org/people/tommccann/sounds/235968/[211]
- Point Beep : https://freesound.org/people/LittleRobotSoundFactory/sounds/270303/[212]
- Gem: https://opengameart.org/content/planetcute-gem-bluepng[213]
- Space Background: https://opengameart.org/content/space-background-8[214]
- Alien Bug: https://opengameart.org/content/8-bit-alien-assets[215]

The spaceship was made by Ritza[216].

Thank you to all the creators for putting their assets up with a Creative Commons license and allowing us to use them.

Things to Try Next

Here are a few things you can try to add to the game and polish it up:

- Self healing on the shield. Perhaps add back 1 or 2 shield points every 10 seconds, so that players can go further if they dodge the aliens.
- A better ending screen, with the player's score.
- An intro scene, explaining the game and the controls.
- Different types of alien bugs. Perhaps a large "boss" bug that can also shoot back.

Code

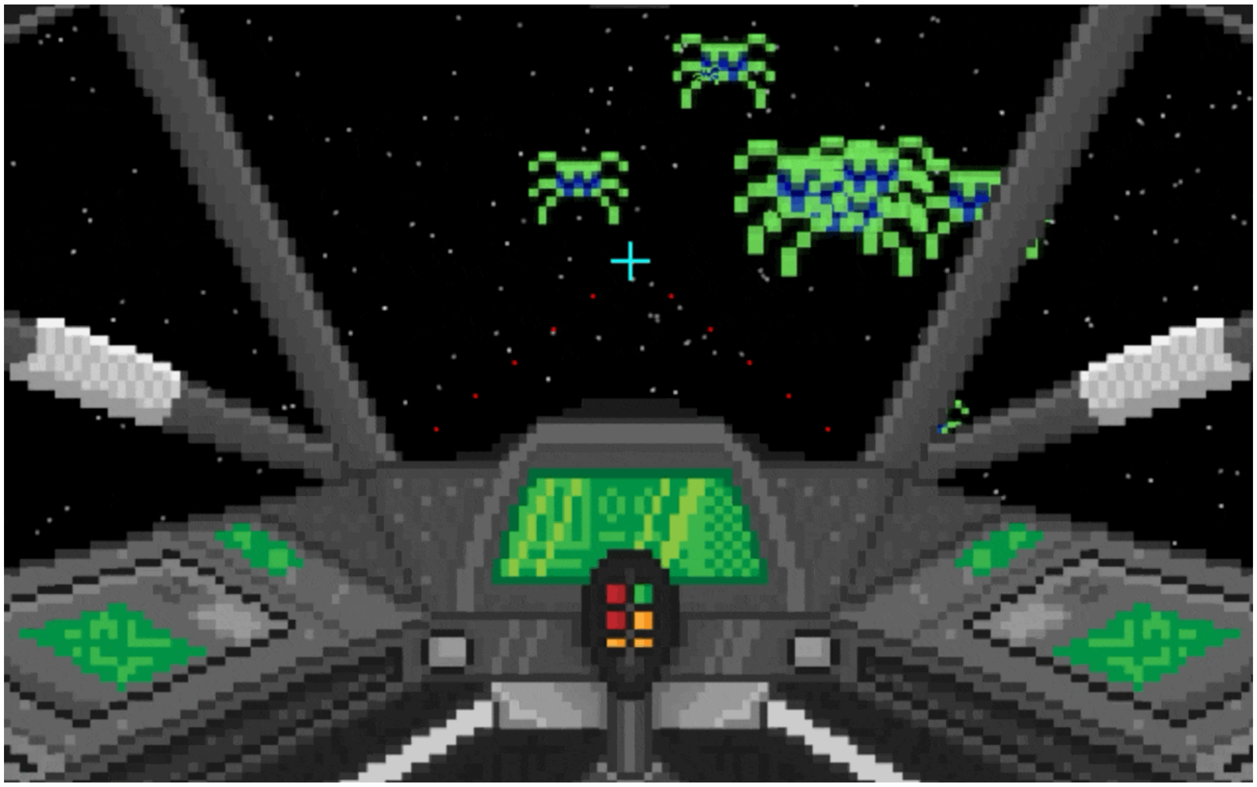
You can find the code for this tutorial on Replit[217]

[209]https://freemusicarchive.org/music/Komiku/Captain_Glouglous_Incredible_Week_Soundtrack/pog
[210]https://freesound.org/people/sunnyflower/sounds/361471/
[211]https://freesound.org/people/tommccann/sounds/235968/
[212]https://freesound.org/people/LittleRobotSoundFactory/sounds/270303/
[213]https://opengameart.org/content/planetcute-gem-bluepng
[214]https://opengameart.org/content/space-background-8
[215]https://opengameart.org/content/8-bit-alien-assets
[216]https://ritza.co
[217]https://replit.com/@ritza/Space-Shooter-new

Building a pseudo-3D game with Kaboom.js

Three-dimensional games became popular in the late 80's and early 90's with games like the early Flight Simulator and Wolfenstein 3D. But these early games were really 2.5D, or pseudo-3D[218]: the action takes place in 2 dimensions, and the world only appears to be 3D.

Kaboom.js[219] is a 2D game engine, but we can use some of those early game designers' techniques to create a pseudo-3D game. This game is roughly based on our 2D space shooter game tutorial[220], but we'll use a view from the cockpit of the spaceship instead of the side-scrolling view.



The finished game

Click to open gif[221]

You can download this zip file[222] with all the sprites and sounds you'll need for this tutorial.

[218]https://en.wikipedia.org/wiki/2.5D
[219]https://kaboomjs.com
[220]https://docs.replit.com/tutorials/24-build-space-shooter-with-kaboom
[221]https://docs.replit.com/images/tutorials/25-3d-game-kaboom/gameplay.gif
[222]https://docs.replit.com/tutorial-files/3d-game-kaboom/3d-game-resources.zip

Game design

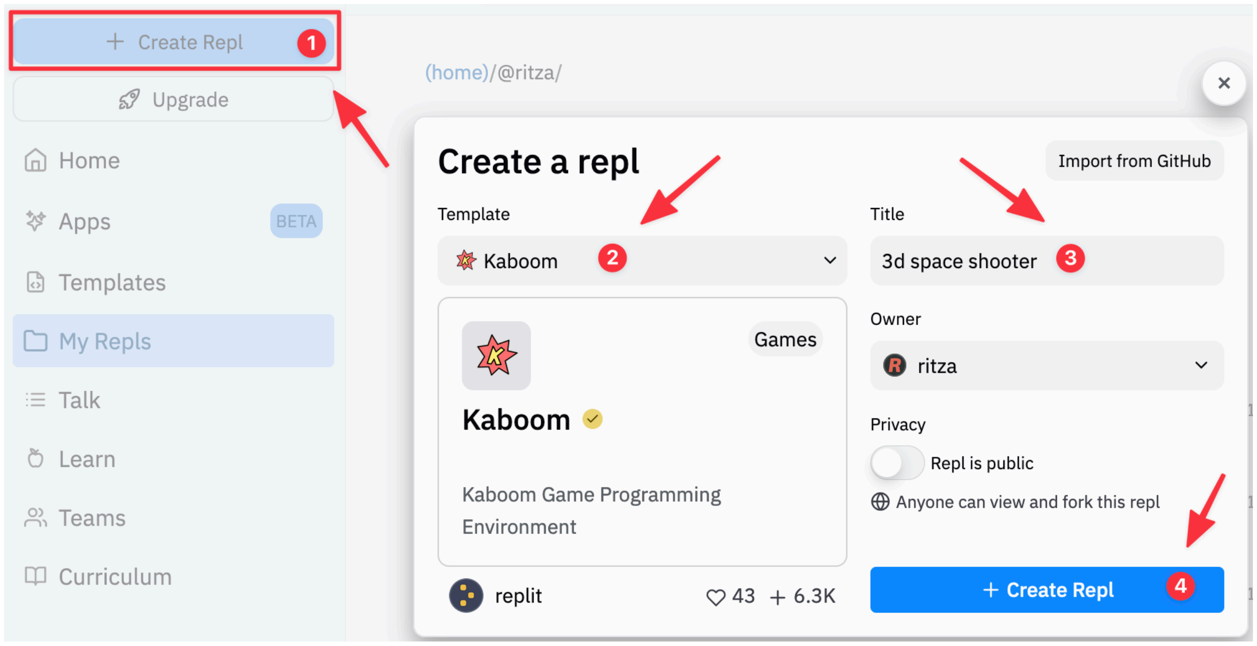
Here's what we want from this game:

- A sense of depth to give the illusion of 3D
- The feeling of freedom of movement throughout space

We'll make use of Kaboom's scale component[223] to achieve the sense of depth, representing our sprites as smaller if they are meant to be further away, and larger when they are closer. We'll create a feeling of moving through space using an algorithm to generate a star field, like the early Windows screensavers.

Creating a new project

Head over to Replit[224] and create a new repl. Choose **Kaboom** as your project type. Give this repl a name, like "3D Space Shooter".



Template: Kaboom

Kaboom Game Programming Environment

Title: 3d space shooter

Owner: ritza

Privacy: Repl is public

Anyone can view and fork this repl

Creating a new repl

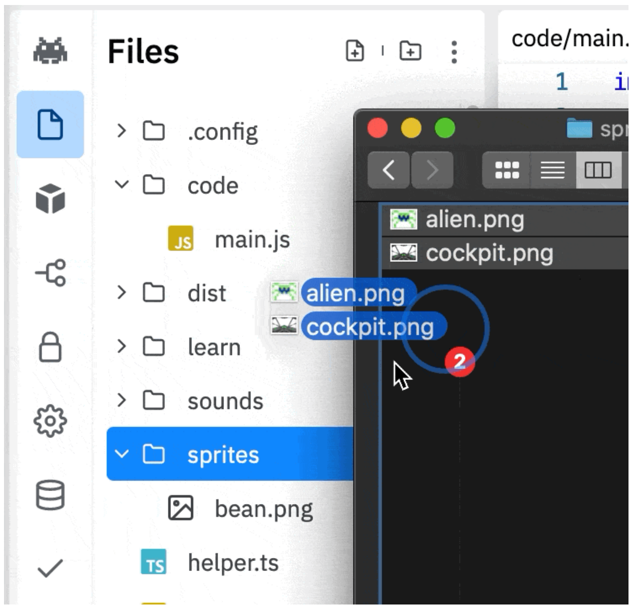
After the repl has booted up, you should see a `main.js` file under the "Code" section. This is where we'll start coding.

[223] https://kaboomjs.com/doc/#scale
[224] https://replit.com

Setting up the Kaboom environment

The Kaboom interface on Replit is specialised for game-making. Besides the Space Invader icon, you'll notice a few special folders in the file try, like "Code", '"Sprites", and "Sounds". These special folders take care of loading up assets, and all the necessary code to start scenes and direct the game. You can read up more about the Kaboom interface here[225].

If you haven't already, download this zip file[226] containing all the sprites and sounds for the game. Extract the file on your computer, then add the sprites to the "Sprites" folder, and the sounds to the "Sounds" folder.



Uploading sprites

Click to open gif[227]

To set up the game play environment, we need to set up Kaboom with the screen size and colors we want for the game window. Replace the code in `main.js` with the code below:

[225]https://docs.replit.com/tutorials/kaboom
[226]https://docs.replit.com/tutorial-files/3d-game-kaboom/3d-game-resources.zip
[227]https://docs.replit.com/images/tutorials/25-3d-game-kaboom/upload-sprites.gif

```
import kaboom from "kaboom";

kaboom({
  background: [0, 0, 0],
  width: 320,
  height: 200,
  scale: 2
});
```

Here we import the kaboom library, and then initialize the context by calling `kaboom({ ... })`. We set the size of the view to 320x200 pixels and `scale` to make the background twice the size on screen. Now, let's load up the sprites and sounds we will need in this game. The code below loads each of the graphic elements we'll use, and gives them a name, so we can refer to them when we build the game characters:

```
loadRoot("sprites/")
loadSprite("cockpit", "cockpit.png")
loadSprite("alien", "alien.png")

loadRoot("sounds/")
loadSound("shoot","shoot.wav")
loadSound("explosion","explosion.wav")
```

The first line, `loadRoot`[228], specifies which folder to load all the sprites and game elements from, so we don't have to keep typing it in for each sprite. Then each line loads a game sprite and gives it a name so that we can refer to it in code later. We use similar code to load the sounds we will need in this game, but instead of `loadSprite`[229] we use `loadSound`[230] to load sounds.

Creating the interface layers

Kaboom games are made up of "Scenes", which are like levels, or different parts and stages of a game. The IDE has a default "main" scene already, which we can use for our main game code. Each scene has multiple "Layers", allowing us to have backgrounds that don't affect the game, main game objects (like the player, bullets, enemies, and so on), and UI elements (like the current score and health).

Add the following code to the `main.js` file to create the 3 layers "Background (`bg`)", "Object (`obj`)", "User Interface (`ui`)":

[228]https://kaboomjs.com/#loadRoot
[229]https://kaboomjs.com/#loadSprite
[230]https://kaboomjs.com/#loadSound

```
layers([
    "bg",
    "obj",
    "ui",
], "obj");
```

The `obj` layer is set as the default layer and that's where the game action will take place. We'll use the `bg` layer to draw the star field, as we don't interact with the objects on that layer. Then we'll use the `ui` layer to draw fixed foreground objects, like the cockpit of the spaceship the player is travelling in.

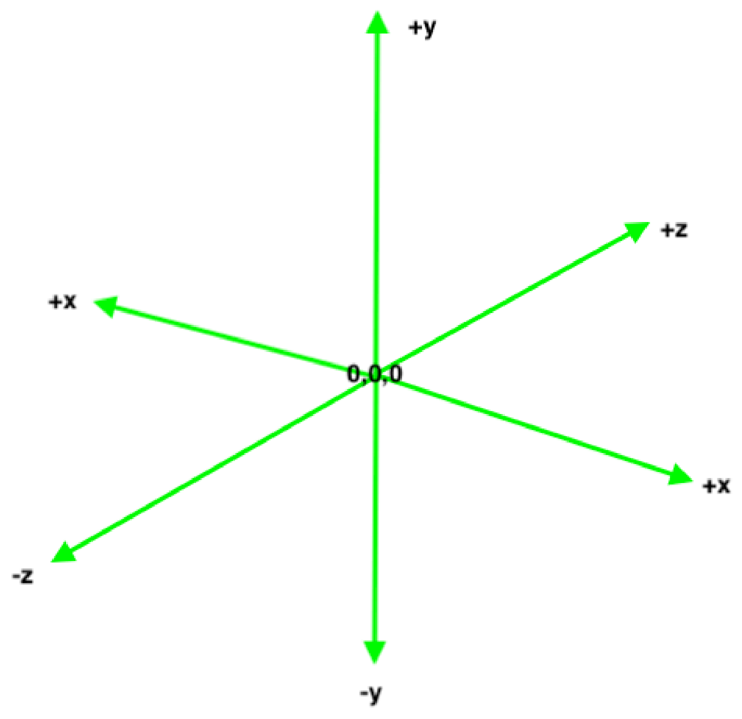
Creating alien bugs

As in the 2D version of this game[231], the point of our game is to avoid and shoot down exploding alien bugs. This time, instead of the bugs coming from the left and right of the screen, we'll make it appear as though they are coming toward the player from "inside" the screen.

To create this effect, we'll start by making the alien bugs small and spread out over the screen, and have them get bigger and loom toward the center of the screen as they get closer.

We need a 3D coordinate system to work out how our elements should move. We'll create a system like the one in the image below, with `0` for all three dimension axes in the center. This is how we'll track the movements of the aliens in code. When we draw them to the screen, we'll convert these coordinates into the 2D screen coordinate system.

[231]https://docs.replit.com/tutorials/24-build-space-shooter-with-kaboom



3D co-ordinate system

Let's add the following code to the `main` scene file to achieve this:

```
const SCREEN_WIDTH = 320;
const SCREEN_HEIGHT = 200;
const ALIEN_SPEED = 200;


let aliens = [];

function spawnAlien(){
    const x = rand(0 , SCREEN_WIDTH);
    const y = rand(0 , SCREEN_HEIGHT);

    var newAlien = add([
        sprite("alien"),
        pos(x,y),
        scale(0.2),
        area(),
        rotate(0),
        {
            xpos: rand(-1*SCREEN_WIDTH/2, SCREEN_WIDTH/2),
            ypos: rand(-1*SCREEN_HEIGHT/2, SCREEN_HEIGHT/2),
            zpos: 1000,
            speed: ALIEN_SPEED + rand(-0.5* ALIEN_SPEED, 0.5*ALIEN_SPEED)
        },
        "alien"
    ]);

    aliens.push(newAlien);
}

loop(0.8, spawnAlien);
```

First, we define some general constants for the size of the screen and the speed at which aliens will move. This way, we don't have to keep remembering and typing numbers, and it's easier to change these aspects later if we need to. We also create an array to hold each alien object we create so that we can keep track of all of them. This will be especially important when we add movement to the aliens.

The function `spawnAlien` creates a new alien at a random location on the screen. The first lines calculate a random x and y position to place the alien on the screen initially. This isn't logically needed, as we'll calculate the alien's actual position later from our 3D coordinate system and calculate the projected screen position on each frame. But we need to pass a position `pos`[232] component to the `add`[233] method when we create a new object, so any random position will do.

There are two more components we include when constructing the alien object:

- `scale`[234], allowing us to adjust the size of the alien over time as if it's getting closer, and
- `rotate`[235], allowing us to rotate the aliens so we can simulate 'rolling' when changing the spaceship's direction.

We also add the coordinates of the alien's position in the 3D system to the alien object as custom properties. We start with a fixed `zpos`, or position on the Z axis, far from the screen.

Then we set the alien's speed, varied by a random amount of up to half the base speed faster or slower so that there's some variety in the way aliens approach the ship. We'll use these custom values when we calculate the alien's position on each frame.

Finally, we add the new alien to the `aliens` array we created earlier to keep track of it.

[232]https://kaboomjs.com/doc/#pos
[233]https://kaboomjs.com/doc/#add
[234]https://kaboomjs.com/doc/#scale
[235]https://kaboomjs.com/doc/#rotate

Outside the function, we make use of the Kaboom `loop`[236] functionality to call the `spawnAlien` function to create new aliens at regular intervals.

Moving the alien bugs

Now we need to have the aliens we've generated move with each frame. Here's the code:

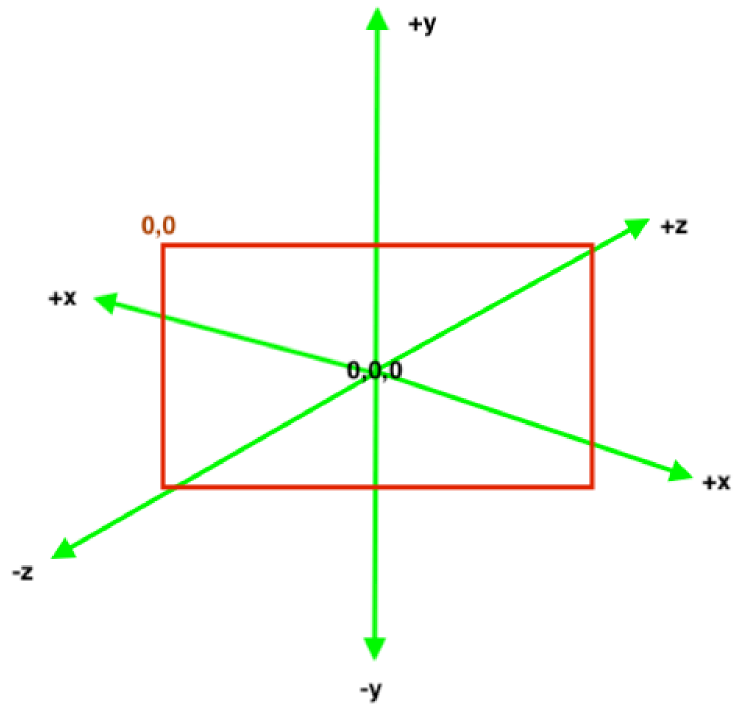
```
onUpdate("alien", (alien)=>{
    alien.zpos -= alien.speed * dt();

    alien.scale = 2 - (alien.zpos * 0.002);

    const centerX = SCREEN_WIDTH * 0.5;
    const centerY = SCREEN_HEIGHT * 0.25;

    alien.pos.x  = centerX + alien.xpos * (alien.zpos * 0.001);
    alien.pos.y = centerY + alien.ypos * (alien.zpos * 0.001);

    if (alien.zpos <= 1 ){
        destroyAlien(alien);
    }
});

function destroyAlien(alien){
    aliens = aliens.filter(a => a != alien);
    destroy(alien);
}
```

First we add a new event handler onto the `onUpdate`[237] event, and filter for any objects tagged `alien`. The `onUpdate` event handler is fired for each frame. In this event handler function, we adjust the `zpos` of the alien to make it 'move' a little closer to the screen. We use the `dt()`[238] function to get the time from the last frame, together with the speed per second we assigned to the alien when we constructed it, to calculate the alien's new `zpos` in our 3D coordinate system. We then translate that value to screen coordinates, and mimic the z-axis position by adjusting the size, or `scale`, of the alien sprite.

Remember that screen coordinates start with (0,0) in the top left corner of the screen, and our 3D coordinate system starts with (0,0,0) in the 'center' of the system. To translate between the 2 systems, we need to find the center of the screen so that we can center the 3D system over it. We do this by

[236]https://kaboomjs.com/doc/#loop
[237]https://kaboomjs.com/doc/#onUpdate
[238]https://kaboomjs.com/doc/#dt

by halving the screen `WIDTH` and `HEIGHT` by `2`. The screen is the red rectangle in the image below, showing how the 3D system will be centered on it.

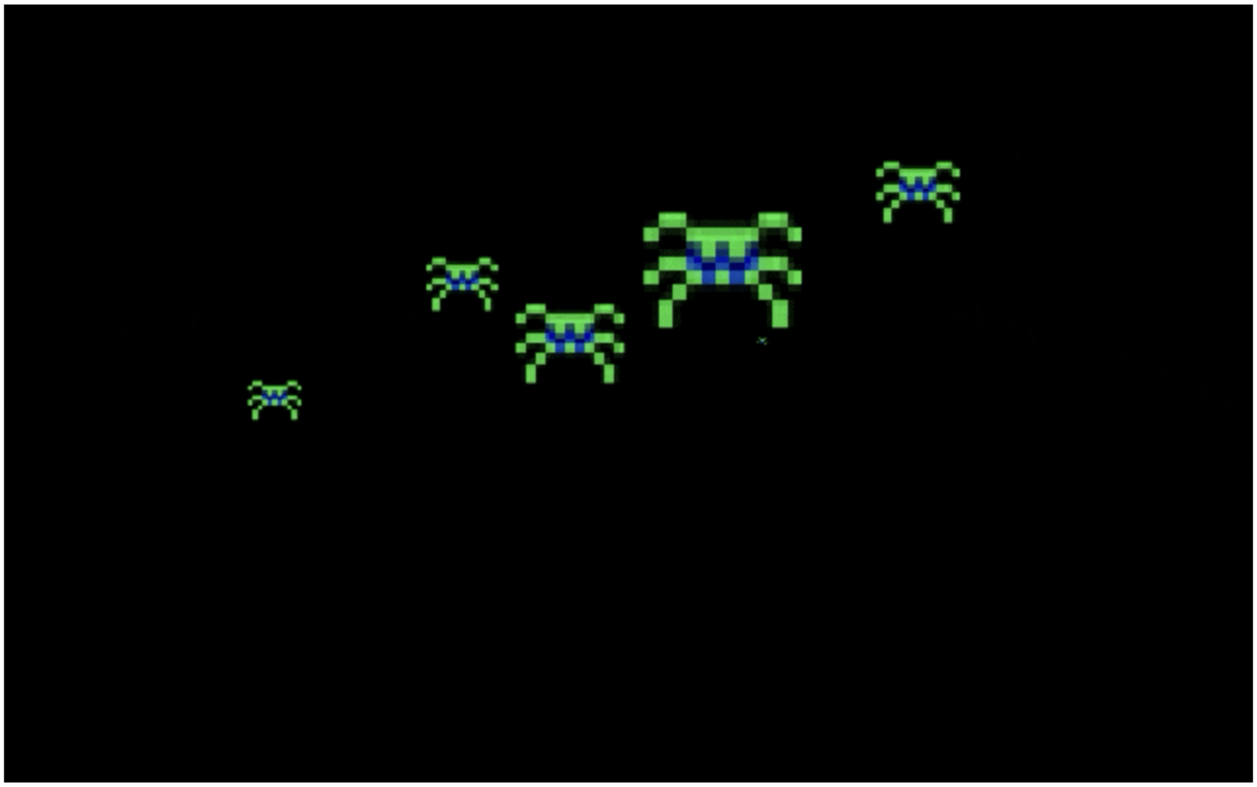


Overlay 3d system over 2d system

Now we can add the alien's `x` and `y` positions in 3D coordinate space relative to the center point of the screen. We bias the center point "up" a bit, as this will seem to be the center of the spaceship's view when we add the cockpit later. We also modify each of these `x` and `y` positions by a factor relating to the alien's `z` position: As the alien approaches, its `zpos` value decreases, and our factor uses this value to draw the alien nearer to the center of the screen. This enhances the depth illusion and makes it feel to the player that the aliens are coming at them.

Finally, we see if the alien is very close by seeing if the `zpos < 1`. If it is, we destroy the alien to remove it from the scene, as it's either gone past our spaceship or crashed into it. We create a small helper function, `destroyAlien`, to manage this, so that we also remove the alien from the tracking array.

If you run the code now, you should see the aliens start to move toward you.



Aliens coming at you

Click to open gif[239]

Adding a star field

Now that we have the aliens moving and coming at us, let's add another element to give a further sense of depth and show that we are in outer space: the star field generator. We can implement the star field in a similar way as we did for the aliens' movement. One difference will be that we will use color, or more specifically intensity, instead of scaling to proxy for the `z-axis`. Another difference is that we'll have the stars spread away from the center rather than towards it, as if the ship is going past them. This also makes it seem like we're travelling at warp speed, which is cool.

[239]https://docs.replit.com/images/tutorials/25-3d-game-kaboom/alieans-coming.gif

```
const STAR_COUNT = 1000;
const STAR_SPEED = 5;
var stars = [];

function spawnStars(){
    for (let i = 0; i < STAR_COUNT; i++) {
        const newStar = {
          xpos: rand(-0.5*SCREEN_WIDTH, 0.5*SCREEN_WIDTH),
          ypos: rand(-0.5*SCREEN_HEIGHT, 0.5*SCREEN_HEIGHT),
          zpos: rand(1000)
        };
        stars.push(newStar);
    }
}

spawnStars();

onUpdate(()=>{
  const centerX = SCREEN_WIDTH * 0.5;
  const centerY = SCREEN_HEIGHT * 0.5;

  stars.forEach((star)=>{
    star.zpos -= STAR_SPEED;
    if (star.zpos <=1)
    {
      star.zpos = 1000;
    }
    const x = centerX + star.xpos / (star.zpos * 0.001);
    const y = centerY + star.ypos / (star.zpos * 0.001);

    if (x>= 0 && x<= SCREEN_WIDTH && y>=0 && y<= SCREEN_HEIGHT) {
      const scaled_z = star.zpos * 0.0005;
      const intensity = (1 - scaled_z) * 255;

      drawRect({
        width: 1,
        height: 1,
        pos: vec2(x,y),
        color: rgb(intensity, intensity, intensity)
      });
    }
  })
});
```

While this is very similar to the code we added above for the alien bugs, you'll notice the `spawnStars` function has a few differences to the `spawnAlien` function, such as:

- We create all the stars at once. This is because we need a significant star field to start with, not just a few stars every second.
- We don't create a Kaboom object for each star. This is because we don't need the collision handling and other overhead that comes with a Kaboom object, especially since we are generating a lot of stars (`const STAR_COUNT = 1000;`). Instead, we store the stars' info in custom object literals[240], and add each of these to the `stars` array.
- We set the initial `z-pos` of the stars to a random value from 0 to 1000, using the Kaboom `rand`[241] function. We do this because we create all the stars at once, so we seed the stars at random positions on the z-axis to give the feeling of depth to the star field. If the stars were all initialised to the same `z-pos`, they would move in unison, and it would look like a mass of pixels were coming at us - a bit weird!

Now take a look at the `onUpdate`[242] event handler for our stars. It differs from the event handler for our alien bugs in a few ways:

- We don't use an object filter to look for the stars, as we didn't create them as Kaboom objects. Instead, we just cycle through each star in the `stars` array.
- Instead of destroying the star and removing it from the array when it reaches the 'front' of the screen, we recycle it by resetting its `z-pos` back to 1000.
- We also check if the star is out of the screen view. If it is, we don't draw it, to save a bit of overhead.
- Instead of using the `z-pos` to calculate a value by which to scale the star, we use it to calculate the star's intensity, or brightness. Kaboom uses color values in the range 0-255. So we first scale the `z-pos` down to below 1. Then we subtract it from 1 to create an inverse relationship between `z-pos` and intensity. We then multiply the intensity value by 255 to scale it to a value that is within the range 0-255. In other words, stars further away from us have higher `zpos` values, giving us lower color intensity. This makes stars far away glow dimly, while those closer to our view look brighter.
- Finally, we use the Kaboom's drawRect[243] method to directly draw the star to the screen. As there is no pixel level drawing function in Kaboom, we create a rectangle of size 1 to draw just one pixel.

Adding the spaceship cockpit

Now that we have a star field to fly through, let's add the player's spaceship. Our game uses the spaceship pilot's point of view. Add the following code to add a view from the spaceship cockpit.

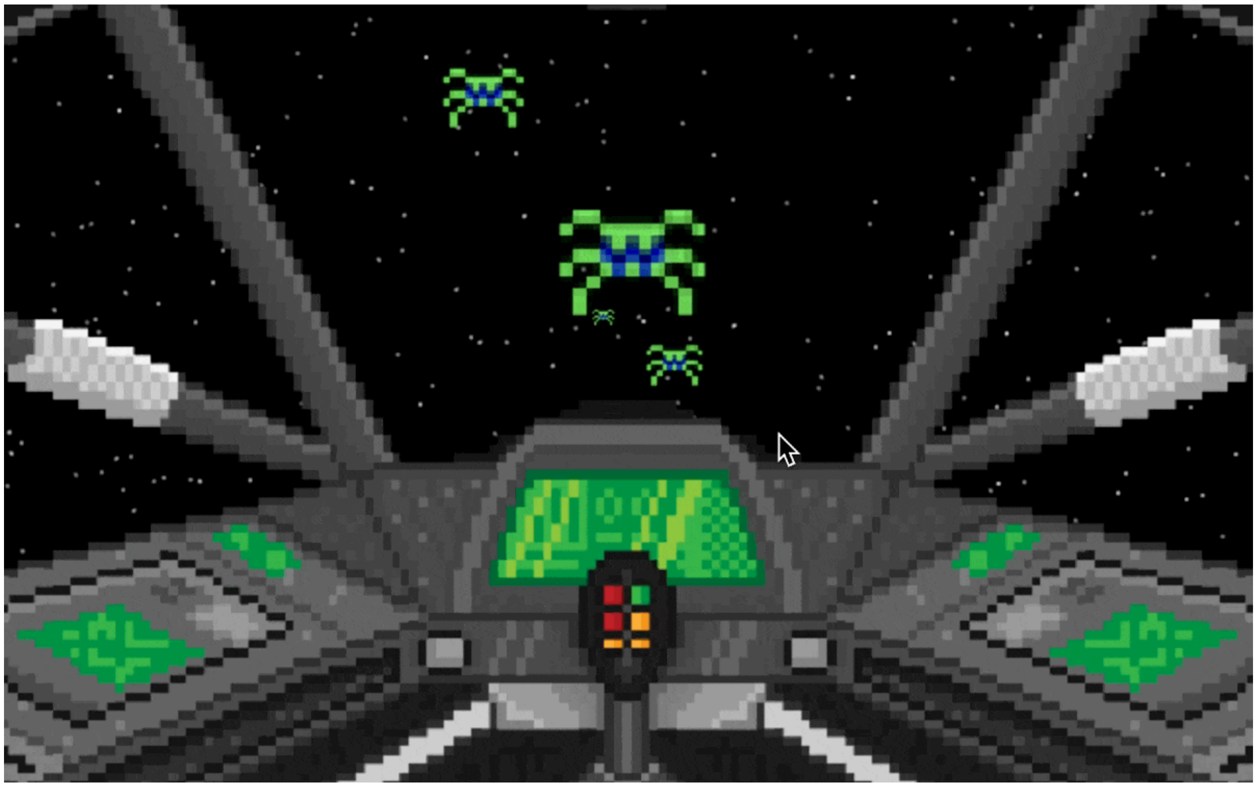
[240]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types#object_literals
[241]https://kaboomjs.com/doc#rand
[242]https://kaboomjs.com/doc/#onUpdate
[243]https://kaboomjs.com/doc/#drawRect

```
    const cockpit = add([
        sprite("cockpit"),
        layer("ui"),
        rotate(0),
        pos(SCREEN_WIDTH/2, SCREEN_HEIGHT/2 ),
        origin("center"),
        scale(0.275)
    ]);
```

This adds the `cockpit` sprite (image) to the `ui` layer. We also add the `rotate`[244] component to it, so that we can add some rotation effects when the spaceship is flying. We use the `origin`[245] component to center the image in the middle of the screen, which also provides the axis to rotate the sprite around when banking (turning) the spaceship. Then we use a scaling factor to `scale`[246] the image down to the size of the screen. We scale the image as it's much larger (1334×834) than the size of the game screen (320x200). We could resize the image in an image editing programme, but we would lose some detail and sharpness. Note that the factor of the scale means that the image is still a little larger than the screen size. This gives us a bit of overlap available for when we rotate the image when banking the spaceship.

Run the game now and you should see the view from inside the spaceship.

[244]https://kaboomjs.com/doc/#rotate
[245]https://kaboomjs.com/doc/#origin
[246]https://kaboomjs.com/doc#scale



Spaceship view

Click top open gif[247]

Creating the spaceship's movement controls

Our basic game world is up and running, now let's add some controls so we can move around in it. We'll allow a few different moves for the spaceship: bank left or right, and fly up or down.

Consider for a moment how the spaceship moves through the game world. We can't move the cockpit left or right, up or down - it would just disappear off the screen. One way of simulating movement from the point of view of the cockpit is to keep it stationary and move all the other game elements.

To achieve this, let's add some helper functions to move the game objects. Here's the code:

[247]https://docs.replit.com/images/tutorials/25-3d-game-kaboom/spaceship-view.gif

```
function shiftAliens(x, y){
    aliens.forEach((alien) =>{
        alien.xpos += x / (alien.zpos*0.01);
        alien.ypos += y / (alien.zpos*0.01);
    });
}

function shiftStars(x, y){
    stars.forEach(star =>{
        star.xpos += x *0.01;
        star.ypos += y *0.01;
    });
}
```

These 2 functions take `x` and `y` values for the amount we want to "move" by, and uses these to move the aliens and the stars. In each case, we loop through the arrays holding the alien or star game objects. We make some adjustments to the values supplied to the functions to account for the perception that, when we move, objects further away appear to move "less" than objects close to us. In the case of the stars, we assume they are all in the far distance, so we scale down the amounts to move by a constant factor. In the case of the aliens, some are far away, while others are right up against the spaceship. To account for this variance, we adjust the amount to move an alien by a factor related to its distance from us, or `zpos`. Aliens close by will appear to move more than those far away.

Now we can add some event handlers for keyboard input:

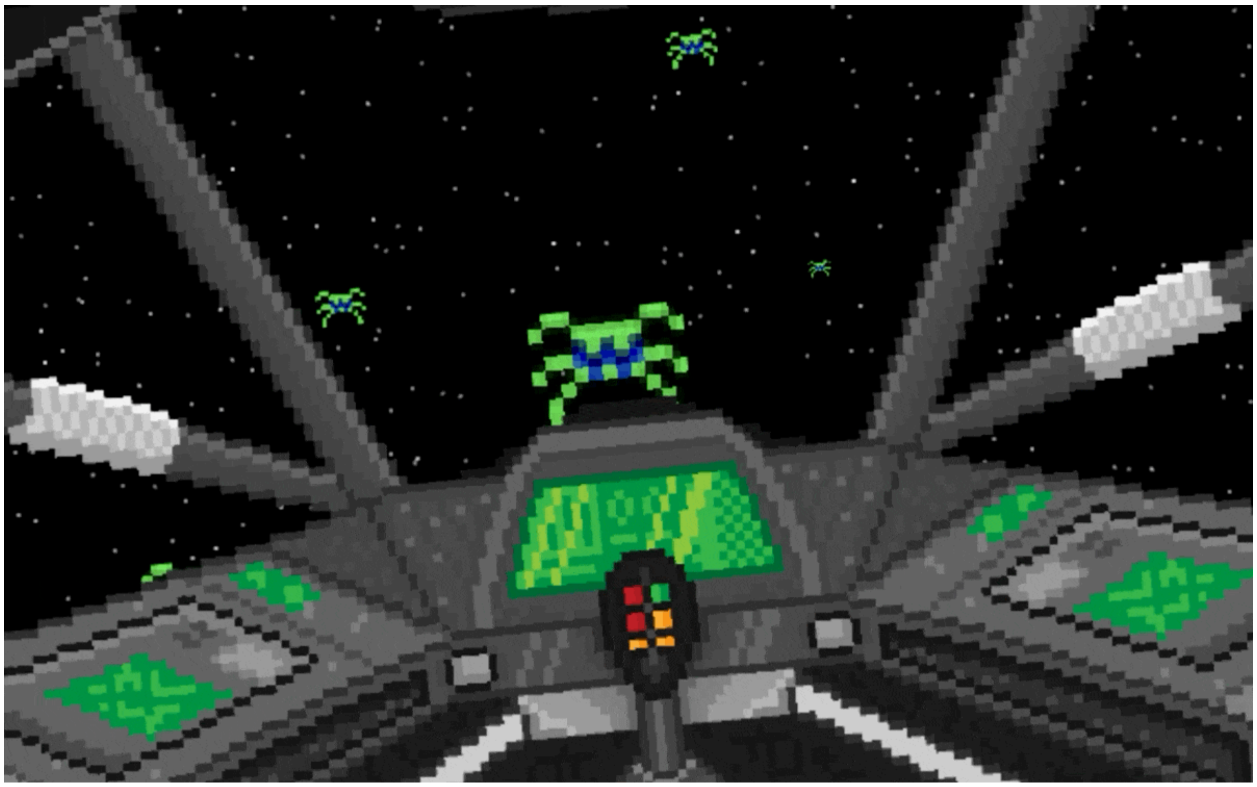
```
    const MOVE_DELTA = 2000;

    onKeyDown("left", () => {
        const delta =  MOVE_DELTA * dt();
        shiftAliens(delta, 0);
        shiftStars(delta*0.01, 0);
        camRot(5.7);
    });

    onKeyDown("right", () => {
        const delta =  -1 * MOVE_DELTA * dt();
        shiftAliens(delta, 0);
        shiftStars(delta*0.01, 0);
        camRot(-5.7);
    });


    onKeyDown("up", () => {
        const delta = MOVE_DELTA * dt();
        shiftAliens(0, delta);
        shiftStars(0,delta*0.01);
    });

    onKeyDown("down", () => {
        const delta = -1 * MOVE_DELTA * dt();
        shiftAliens(0, delta);
        shiftStars(0, delta*0.01);
    });

    onKeyRelease("left", ()=>{
        camRot(0);
    });

    onKeyRelease("right", ()=>{
        camRot(0);
    });
```

Because we're moving the aliens and stars to make it appear as if the spaceship is moving, these elements must move in the opposite direction to that of the arrow key being used. When we use the left key to move the spaceship, our scene moves to the right. We use the Kaboom events `onKeyDown`[248] and `onKeyRelease`[249] to attach event handlers for direction controls to the arrow keys on the keyboard. In each of the `onKeyDown` event handlers, we get the time elapsed from the last frame by calling the `dt()`[250] function, and multiply it by a constant `MOVE_DELTA`, representing the amount to move by each second. For the `right` and `up` keys, our elements move left and down respectively, so we make the amount to move negative - recall that we are moving the objects in our 3D coordinate system. Then we call the 2 helper functions we defined above with the amount to move the objects in the `x` and `y` dimensions.

In the event handlers for `left` and `right` keys, we also make use of the Kaboom `camRot`[251] effect. This effect rotates all objects by the amount we specify, giving the perception of the spaceship banking hard while turning. We add in two additional event handlers on `onKeyRelease`[252] for the `left` and `right` keys to reset the rotation when the player stops turning.

Give the game a run, and you should be able to control the spaceship.

[248]https://kaboomjs.com/doc/#onKeyDown
[249]https://kaboomjs.com/doc/#onKeyRelease
[250]https://kaboomjs.com/doc/#dt
[251]https://kaboomjs.com/doc/#camRot
[252]https://kaboomjs.com/doc/#onKeyRelease



Flying controls

Clcik to open gif[253]

Adding weapons

Now we're flying through the alien bug field, but if an alien makes contact with our spaceship, it will explode and damage us. We need some weapons to shoot the aliens with and protect ourselves. For this, we need to implement some lasers. First, let's add cross hairs to aim with:

```
const vertical_crosshair = add([
    rect(1, 10),
    origin('center'),
    pos(SCREEN_WIDTH*0.5, SCREEN_HEIGHT*0.33),
    color(0, 255, 255),
    layer("ui")
]);

const horizontal_crosshair = add([
    rect(10, 1),
```

[253] https://docs.replit.com/images/tutorials/25-3d-game-kaboom/fly-controls.gif

```
    origin('center'),
    pos(SCREEN_WIDTH*0.5, SCREEN_HEIGHT*0.33),
    color(0, 255, 255),
    layer("ui")
]);
```

This adds 2 lines at a point halfway across the screen, and about 1/3 down the screen, which is roughly the center of the view out of the spaceship window. Since Kaboom doesn't have a line component, we use `rect`[254] to draw rectangles with a width of 1 pixel, effectively a line. We add the cross hairs to the UI layer, so they are always on top of the aliens and stars.

Now we have a point to aim at, let's add the lasers. Our player will shoot using the spacebar, and we want a classic laser effect: 2 lasers, one shooting from each side of the ship towards the same point to give the effect of shooting into the distance, towards a vanishing point.

```
const BULLET_SPEED = 10;
function spawnBullet() {

    const BULLET_ORIGIN_LEFT = vec2(SCREEN_WIDTH *0.25, (SCREEN_HEIGHT - SCREEN_HEIG\
HT*0.33));
    const BULLET_ORIGIN_RIGHT = vec2(SCREEN_WIDTH - (SCREEN_WIDTH*0.25), (SCREEN_HEI\
GHT - SCREEN_HEIGHT*0.33));

    const BULLET_VANISHING_POINT = vec2(SCREEN_WIDTH * 0.5, SCREEN_HEIGHT *0.33);

    add([
        rect(1, 1),
        pos(BULLET_ORIGIN_LEFT),
        area(),
        color(255, 0, 0),
        "bullet",
        {
            bulletSpeed:  BULLET_SPEED ,
            targetPos: BULLET_VANISHING_POINT
        }
    ]);

    add([
        rect(1, 1),
        pos(BULLET_ORIGIN_RIGHT),
        color(255, 0, 0),
        "bullet",
```

[254] https://kaboomjs.com/doc/#rect

```
        {
            bulletSpeed:  -1*BULLET_SPEED,
            targetPos: BULLET_VANISHING_POINT
        }
    ]);

    play("shoot", {
        volume: 0.2,
        detune: rand(-1200, 1200),
    });
}
```

When the player fires, we call the `spawnBullet` function to create a new set of laser bullets. First, we calculate the position the bullets will be coming from. To make it seem as though they're coming from under the spaceship either side, we calculate our bullets' starting positions a quarter way from each side of the screen and about a third of the way from the bottom using the multipliers `0.25` and `0.33` respectively.

Then we calculate where we want the bullets to end up. This is the same position as the cross hairs.

Using these values, we create 2 bullet objects - simple 1 pixel objects with the tag `bullet` and color set to red `(255,0,0)` so they look menacing. We also add custom properties to the object: A speed for the bullet to move at, and the vanishing point where its course ends.

As a final detail, we set our "shoot" sound to play as each bullet is created, adjusting the volume and applying a randomly generated detune value so that the pitch of the sound is slightly different each time it's played.

We've got our bullets, their sounds and their trajectories, so let's make them go from their origin point to the vanishing point at the cross hairs by moving them with each frame:

```
onUpdate("bullet", (b) => {

    const m = (b.pos.y - b.targetPos.y) / (b.pos.x - b.targetPos.x);
    const c = b.targetPos.y - m*(b.targetPos.x);

    let newX = b.pos.x + b.bulletSpeed;
    let newY = m * newX + c;
    b.pos.x = newX
    b.pos.y = newY;
    // Remove the bullet once it has hit the vanishing point y line
    if ((b.pos.y < SCREEN_HEIGHT*0.33)) {
        destroy(b);
    }
});
```

```

onKeyDown("space", () => {
    spawnBullet();
});
```

Here we use the `onUpdate`[255] event handler, filtered to `bullet` objects.

To calculate the bullet's next position on its trajectory for each frame, we need to find the values for the slope (`m`) and y-intercept (in this case, `c`) of the straight line between the bullet's current position and its end position. Our first `2` lines of the function express those variable parameters as formulas. Let's take a moment to see how we came to those formulas.

We began with the equation for a straight line[256]:

```
y = m*x + c
```

Since we have the `x` and `y` coordinates for the start and end of the trajectory, we can use them to find the values of the unknowns `m` and `c` by solving simultaneous equations:

```
y_start = m*x_start + c         (1)
y_target = m*x_target + c       (2)

re-arranging (2):
c = y_target - m*x_target

Substitute (2) into (1) for c:
y_start = m*x_start + (y_target - m*x_target)
y_start - y_target  = m*x_start - m*x_target
                    = m*(x_start - x_target)
so m = (y_start - y_target) / (x_start - x_target)

Now we can solve for c:
c = y_target - m*x_target
```

Now that we can express `m` and `c` as formulas, we use them in our code to calculate the parameters.

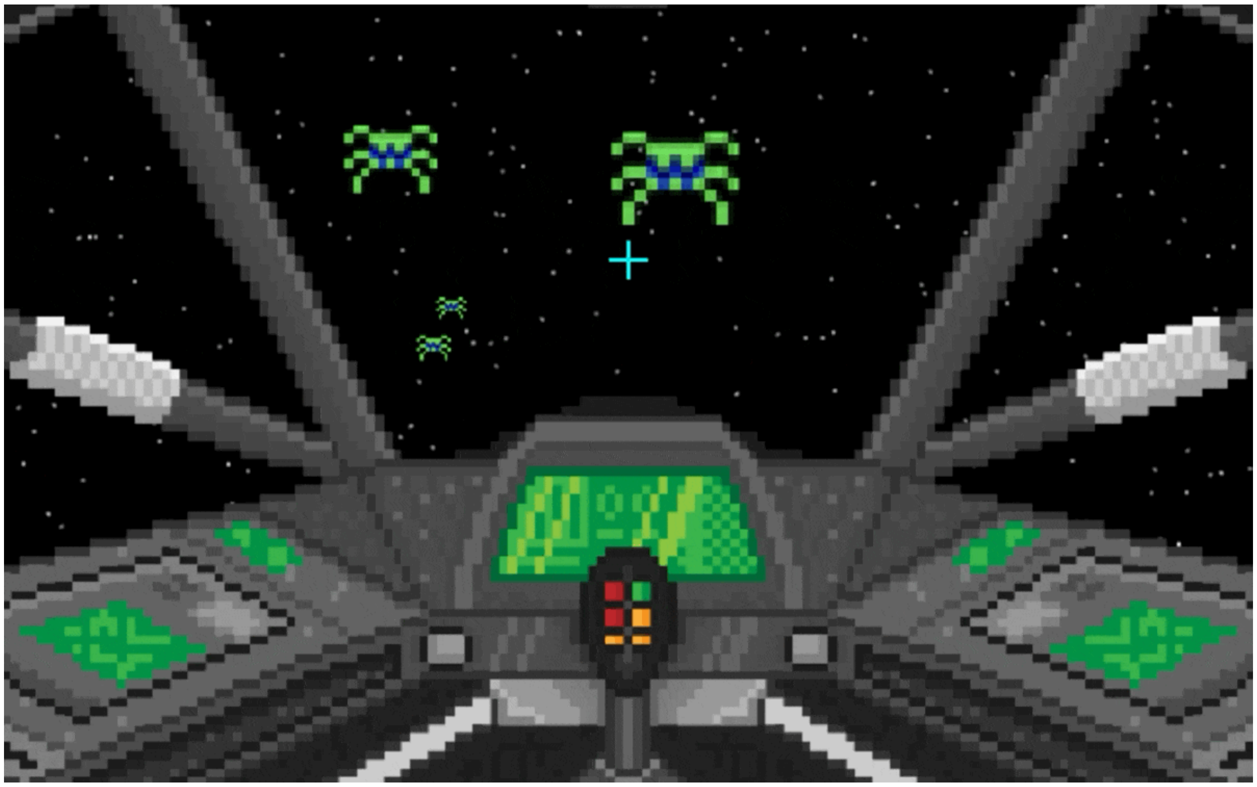
Our code goes on to advance the bullet's current `x` position by the bullet speed amount, and we can figure out the corresponding new `y` position using the `m` and `c` values calculated above with our new `x` position. We then update the bullet's new position (`pos`) with these new values.

We want the bullets to disappear once they hit the target at the vanishing point, so we go on to check if the bullet has crossed the horizontal cross hairs. If it has, we remove the bullet from the scene using the `destroy`[257] function.

[255]https://kaboomjs.com/doc/#onUpdate
[256]https://www.mathsisfun.com/equation_of_line.html
[257]https://kaboomjs.com/doc/#destroy

Finally, we have an event handler for the `space` key, which calls the `spawnBullet` function whenever it is pressed.

Try this out now, and you should be able to shoot some laser bullets into space.



Shooting

Click to open gif[258]

Checking for collisions with bullets

Now that we can shoot bullets, we need to check if they hit an alien bug. If they did, we explode the alien.

[258]https://docs.replit.com/images/tutorials/25-3d-game-kaboom/shooting.gif

```
const BULLET_SLACK = 10;
onCollide("alien","bullet", (alien, bullet) =>{
    if (bullet.pos.y > SCREEN_HEIGHT*0.33 + BULLET_SLACK) return;
    makeExplosion(bullet.pos, 5, 5, 5);
    destroy(alien);
    destroy(bullet);
});
```

We make use of the Kaboom event `onCollide`[259] which is fired when 2 game objects are overlapping or touching each other. We pass in the tags for the aliens and bullets, so we know when they collide.

We want to limit bullet hits to only be around the target area, so that the 3D perspective is kept. But because they could collide at any point along the path the bullet takes, we check if the collision has taken place at around the cross hairs area. Then, if is in the target zone, we remove both the bullet and the alien from the scene, and call a function to create an explosion effect. This is the same code used in the 2D Space Shooter tutorial[260].

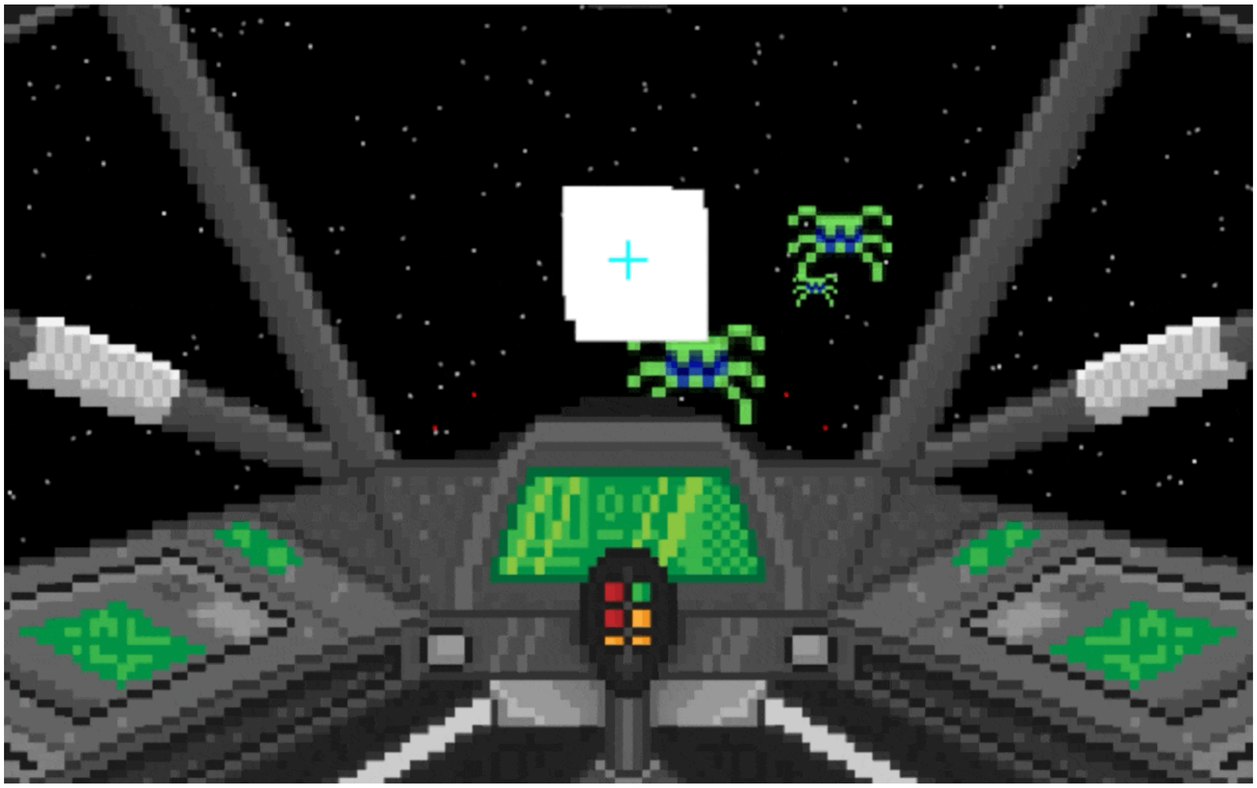
```
function makeExplosion(p, n, rad, size) {
        for (let i = 0; i < n; i++) {
            wait(rand(n * 0.1), () => {
                for (let i = 0; i < 2; i++) {
                    add([
                        pos(p.add(rand(vec2(-rad), vec2(rad)))),
                        rect(1, 1),
                        scale(1 * size, 1 * size),
                        lifespan(0.1),
                        grow(rand(48, 72) * size),
                        origin("center"),
                    ]);
                }
            });
        }
}

function lifespan(time) {
        let timer = 0;
        return {
            update() {
                timer += dt();
                if (timer >= time) {
                    destroy(this);
```

[259]https://kaboomjs.com/doc/#onCollide
[260]https://docs.replit.com/tutorials/24-build-space-shooter-with-kaboom

```
                }
            },
        }
}

function grow(rate) {
    return {
        update() {
            const n = rate * dt();
            this.scale.x += n;
            this.scale.y += n;
        },
    };
}
```

We won't explain this code here, but if you'd like to know how it works, visit the 2D Space Shooter tutorial[261] to learn more.

Run this now, and you should be able to shoot the alien bugs down.



Shooting explosions

[261] https://docs.replit.com/tutorials/24-build-space-shooter-with-kaboom

Click to open gif[262]

Checking if alien bugs hit the spaceship

Now we can add functionality to check if an alien bug makes it past our laser and explodes into the spaceship. Since the cockpit covers the entire screen, we can't make use of the `onCollide`[263] function to check if an alien has hit the cockpit, as it would always be colliding. Instead, we can check the `z` value of the alien, plus if it is within an area of the spacecraft that would cause damage. We'll use a "strike zone" in the center of the cockpit view as the area that aliens can do damage to the craft. Outside that area, we'll assume that the aliens go around or up and over the spacecraft.

To implement this scheme, add a definition for the strike zone:

```
const STRIKE_ZONE = {x1:80, x2:240, y1:20, y2:100};
```

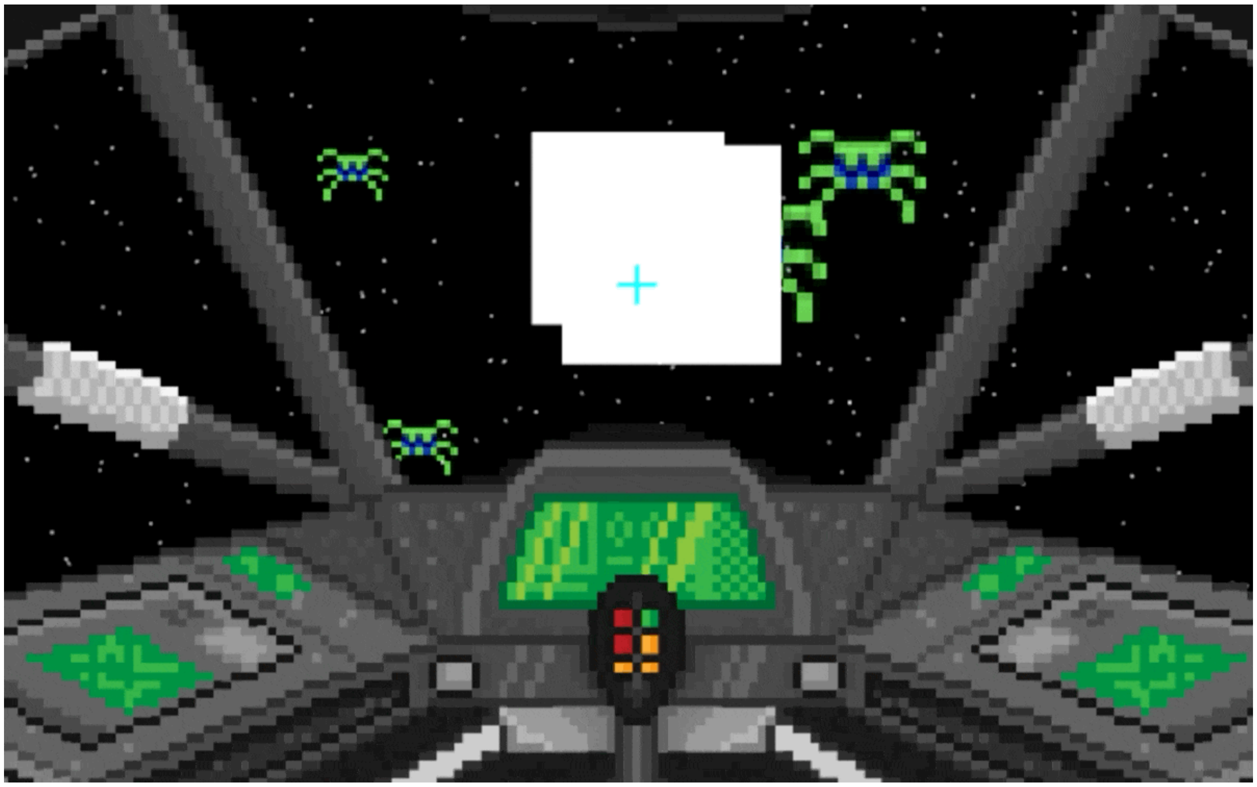
Then we can modify the `onUpdate("alien",....)` event handler that we added earlier in "**Moving the Alien Bugs**" section. In the part of the function where we check if the alien is close to us (`if (alien.zpos <= 1 )`), update the code as follows:

```
  if (alien.zpos < 1 ){
      //check if the alien has hit the craft
      if (alien.pos.x >= STRIKE_ZONE.x1 &&
          alien.pos.x <= STRIKE_ZONE.x2 &&
          alien.pos.y >= STRIKE_ZONE.y1 &&
          alien.pos.y <= STRIKE_ZONE.y2){
              shake(20);
              makeExplosion(alien.pos, 10, 10, 10);
      }
      destroyAlien(alien);
   }
```

We've modified the code to check if the alien is really close to us (`alien.zpos < 1`), and if it is, we check if it is within the bounds of the `STRIKE_ZONE` area. The strike zone is a rectangle - you could implement more complex shapes if you wanted to be more accurate about where the alien can hit. However, a rectangle approximation is OK for this game.

If the alien is close enough, and within our strike zone, we use the `shake`[264] effect to make it "feel" like we've been hit. Then we create an explosion at the point of impact for some visual effects.

[262]https://docs.replit.com/images/tutorials/25-3d-game-kaboom/shooting-explosion.gif
[263]https://kaboomjs.com/doc/#onCollide
[264]https://kaboomjs.com/#shake


Colliding

Click to open gif[265]

Finishing up the game

Congratulations, we've got all the main elements of flying and shooting and damage in the game. The next thing to do would be to add a scoring system, and a way to reduce the spaceship's health or shield when it gets hit. You can look at the tutorial for the 2D version of this game[266], and copy the scoring and health code from there into this game. You can also copy the code for background music and more sound effects.

Happy coding and have fun!

Credits

The game art and sounds used in this tutorial are from the following sources:

Laser : https://freesound.org/people/sunnyflower/sounds/361471/[267]

[265]https://docs.replit.com/images/tutorials/25-3d-game-kaboom/colliding.gif
[266]https://docs.replit.com/tutorials/24-build-space-shooter-with-kaboom
[267]https://freesound.org/people/sunnyflower/sounds/361471/

Explosion: https://freesound.org/people/tommccann/sounds/235968/[268]

Alien Bug: https://opengameart.org/content/8-bit-alien-assets[269]

The spaceship cockpit was made by Ritza[270].

Thank you to all the creators for putting their assets up with a Creative Commons license and allowing us to use them.

Code

You can find the code for this tutorial on Replit[271]

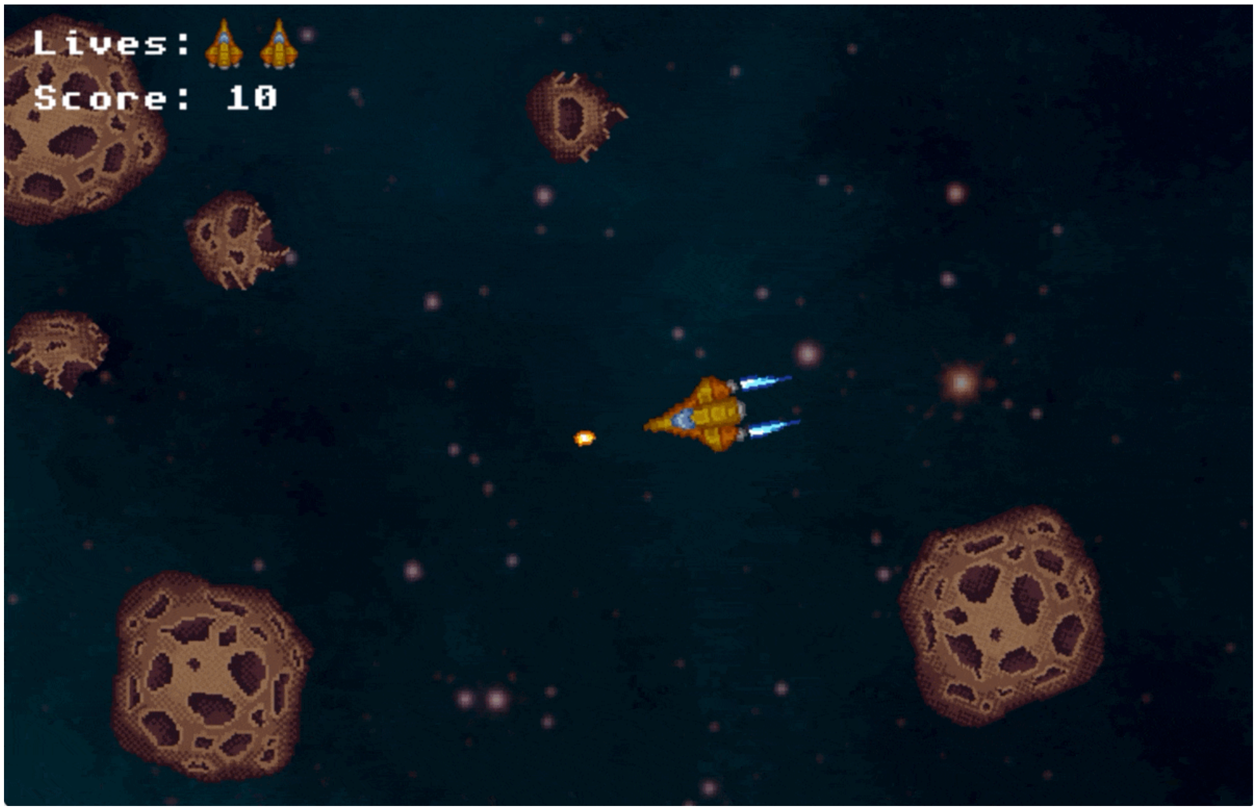
[268]https://freesound.org/people/tommccann/sounds/235968/
[269]https://opengameart.org/content/8-bit-alien-assets
[270]https://ritza.co
[271]https://replit.com/@ritza/3d-space-shooter-new

Building Asteroids with Kaboom.js

Following our previous tutorial on building Snake[272], in this tutorial we'll implement Asteroids, one of the first major arcade game hits, developed by Atari in 1979. We'll use Replit's built-in game development platform, Kaboom.js[273], and cover the following topics:

- Getting set up in Kaboom.js.
- Calculating movement angles for our spaceship and bullet mechanics.
- Random generation of asteroid placement.
- Defining the effects of object collisions.
- Developing a polished game with animation and sound effects.

Our finished game will look like this:



The finished game

[272]https://docs.replit.com/tutorials/21-build-snake-with-kaboom
[273]https://kaboomjs.com/

Click to open gif[274]

We will use these Asteroids sprites[275] and this space background[276] from OpenGameArt.org, and the following sounds from FreeSound.org:

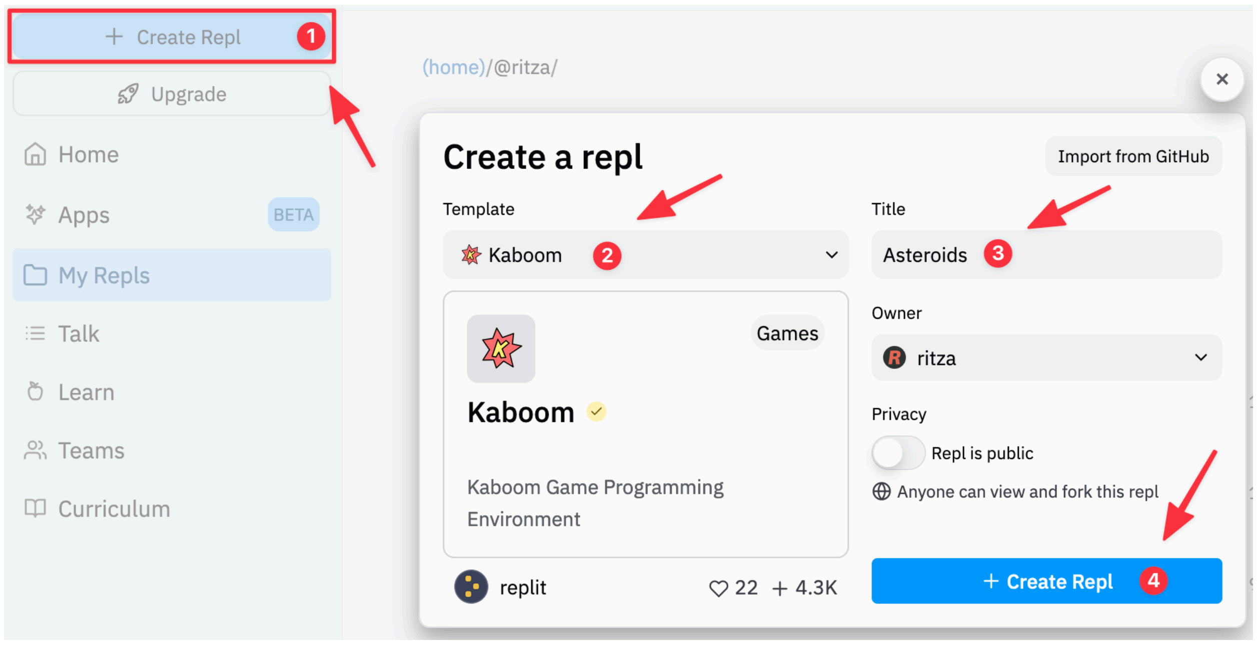
- Laser[277]
- Explosion[278]
- Rocket thrust[279]

We will also be using music by Eric Matyas of Soundimage.org[280].

We've created a single ZIP file with the sprites and sounds you will need for this tutorial, which you can download here[281].

Creating a new project and loading assets

Log into your Replit[282] account and create a new repl. Choose **Kaboom** as your project type. Give this repl a name, like "asteroids".

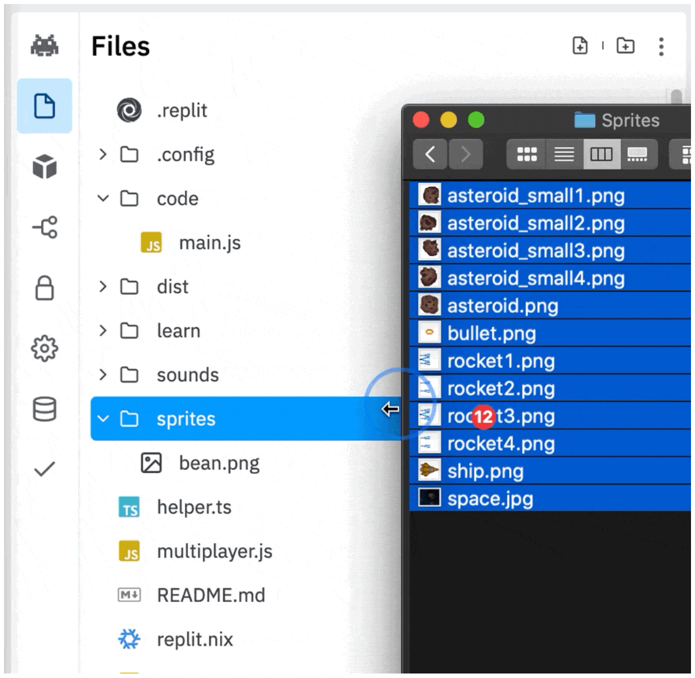


Creating an REPL

[274] https://docs.replit.com/images/tutorials/23-asteroids-kaboom/asteroids-game.gif
[275] https://opengameart.org/content/asteroids-game-sprites-atlas
[276] https://opengameart.org/content/space-background
[277] https://freesound.org/people/Daleonfire/sounds/376694/
[278] https://freesound.org/people/deleted_user_5405837/sounds/399303/
[279] https://freesound.org/people/MATRIXXX_/sounds/515122/
[280] https://soundimage.org/sci-fi/
[281] https://docs.replit.com/tutorial-files/asteroids-kaboom/asteroids-resources.zip
[282] https://replit.com

Kaboom repls are quite different from other kinds of repls you may have seen before: instead of dealing directly with files in folders, you'll be dealing with scenes, sounds and sprites.

Before we start coding, we need to upload our sprites and sounds. Download this ZIP file[283] and extract it on your computer. Click the "Files" icon on the sidebar then, upload everything in Sounds folder to the "sounds" section of your repl, and everything in the Sprites folder to the "sprites" section of your repl.



Uploading sprites

Click to open gif[284]

Once you've uploaded the files, you can click on the "Kaboom" icon in the sidebar, and return to the "main" code file.

Setting up Kaboom

To start, we need to initialise and set up Kaboom with the scale we want for the game window. Replace the code in `main.js` with the code below:

[283] https://docs.replit.com/tutorial-files/asteroids-kaboom/asteroids-resources.zip
[284] "https://docs.replit.com/images/tutorials/23-asteroids-kaboom/upload-sprites.gif

```
import kaboom from "kaboom";

kaboom({
  scale: 1.5
});
```

This will give us a blank canvas with a nice checkerboard pattern.

Now, let's load up the sprites and sound files to make them available in the game. This code loads each of the graphic elements we'll use, and gives them a name so we can refer to them when we build the game objects:

```
loadRoot("sprites/");
loadSprite("space", "space.jpg");
loadSprite("rocket1", "rocket1.png");
loadSprite("rocket2", "rocket2.png");
loadSprite("rocket3", "rocket3.png");
loadSprite("rocket4", "rocket4.png");
loadSprite("ship", "ship.png");
loadSprite("bullet", "bullet.png");
loadSprite("asteroid", "asteroid.png");
loadSprite("asteroid_small1", "asteroid_small1.png");
loadSprite("asteroid_small2", "asteroid_small2.png");
loadSprite("asteroid_small3", "asteroid_small3.png");
loadSprite("asteroid_small4", "asteroid_small4.png");

loadRoot("sounds/");
loadSound("rocket_thrust", "rocket_thrust.wav");
loadSound("laser", "laser.wav");
loadSound("explosion", "explosion.mp3");
loadSound("Steamtech-Mayhem_Looping","Steamtech-Mayhem_Looping.mp3");
```

The first line, `loadRoot`[285], specifies which folder to load all the sprites and game elements from, so we don't have to keep typing it in for each sprite. Then each line loads a game sprite and gives it a name so that we can refer to it in code later. We also use similar code to load sound elements for our game using the `loadSound`[286] function.

Setting the scene

A Kaboom.js game is made up of scenes, which you can think of as different screens, levels or stages. You can use scenes for game levels, menus, cut-scenes and any other screens your game might contain. In this tutorial, we'll just use one scene, which will contain the entire game logic.

[285]https://kaboomjs.com/#loadRoot
[286]https://kaboomjs.com/#loadSound

Scenes are further divided into layers, which are populated by game objects (also called sprites). The layer an object is on will determine when it gets drawn and which other objects it can collide with. In this game, we'll use three layers: the background layer (`bg`), the object layer (`obj`), and UI layer (`ui`). To initialize these layers, add the code below to the bottom of the `main.js` file:

```
scene("main", ()=> {
  layers([
    "bg",
    "obj",
    "ui",
  ], "obj");

  // add more scene code here
});

go("main");
```

On the first line we define the `main` scene, and then we declare the game layers.
These layers will be drawn in the order declared. The majority of gameplay will happen in the `obj` layer, so we've set that as the default layer. Any objects we create will be placed in this layer, unless we specify a different layer when we create the object.

Note: The code snippets in the sections that follow should be added within the body of the main scene unless specified otherwise.

The `bg` layer will be drawn first, beneath everything else, and we'll use that to specify a background image for our game. Do that now by adding the following code to within the body of the main scene.

```
// Background
add([
    sprite("space"),
    layer("bg")
]);
```

Here we're adding a very simple game object: the space sprite we uploaded earlier on the background layer. Later game objects, such as the player's ship and the asteroids, will be more complicated.

The final layer, `ui`, is where we will display information such as the player's remaining lives and total score. Let's draw the score now. First, we have to declare a global variable named `score`, with the following line:

```
let score = 0;
```

Now we'll create an empty object on the UI layer, as follows:

```
// UI
ui = add([
    layer("ui")
]);
```

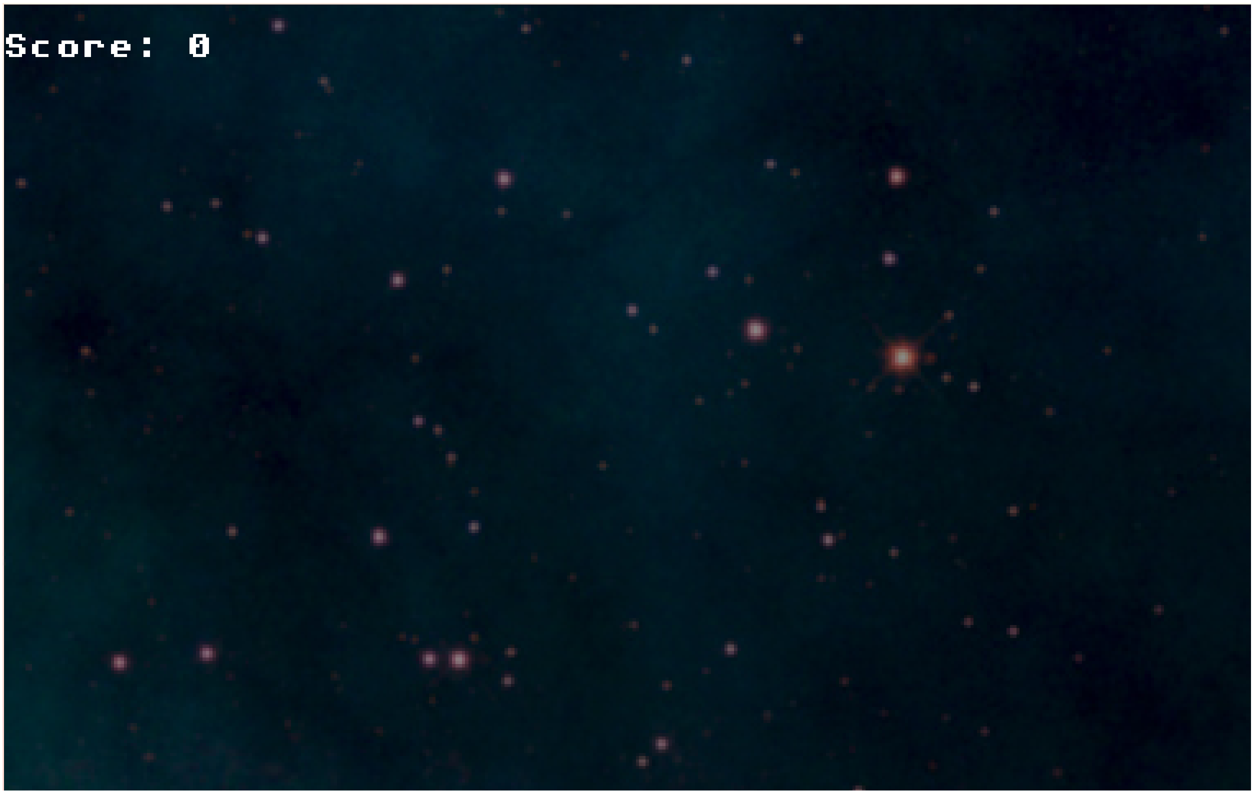
Although Kaboom allows us to create objects that display text[287], this text is set once at object creation and has to be updated manually, which doesn't really make sense for a real-time UI. Instead of doing that, we'll use our `ui` object's draw event[288] callback[289] to draw text containing the current score. Add the following code:

```
ui.on("draw", () => {
  drawText({
    text: "Score: " + score,
    size: 14,
    font: "sink",
    pos: vec2(8, 24)
  })
});
```

Callbacks are a key concept in JavaScript and Kaboom makes heavy use of them. Some callbacks, such as the one above, are called on every frame of the game (around 60 times a second). Others are called when specific events happen, such as user keypresses. Because this callback will be invoked so often, our score will always be up to date.

Run your repl now and marvel at the vast emptiness of space.

[287]https://kaboomjs.com/doc/#text
[288]https://kaboomjs.com/doc/#on
[289]https://en.wikipedia.org/wiki/Callback_(computer_programming)



Empty with score

The player’s ship

Now let’s populate that empty space. Enter the following code below the UI drawing code to create the player:

```
// The ship
const player = add([
    sprite("ship"),
    pos(160, 120),
    rotate(0),
    origin("center"),
    solid(),
    area(),
    "player",
    "mobile",
    "wraps",
    {
        turn_speed: 4.58,
        speed: 0,
        max_thrust: 48,
        acceleration: 2,
        deceleration: 4,
        lives: 3,
        can_shoot: true,
        laser_cooldown: 0.5,
        invulnerable: false,
        invulnerablity_time: 3,
        animation_frame: 0,
        thrusting: false
    }
]);
```

Here we're creating a game object with a number of components[290], each of which give our object some data or behavior. These are:

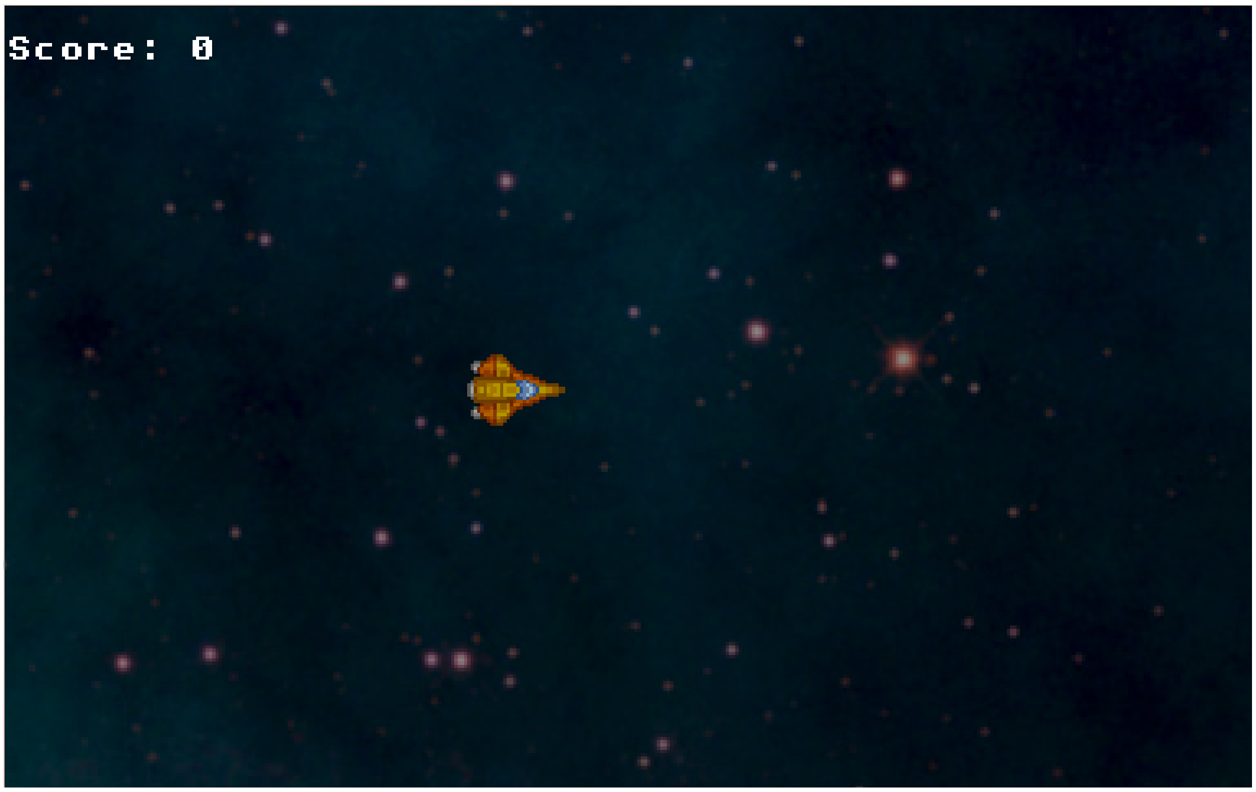
- The `sprite`[291] component, which draws the `ship` sprite.
- The `pos`[292] (position) component, which places the player near the center of the screen in the Replit browser.
- The `rotate`[293] component, which will allow the player to turn the ship with the left and right arrow keys.
- The `origin`[294] component, which sets the sprite's *origin* to "center", so that when we rotate the ship, it will rotate around the middle of its sprite rather than the default top-left corner.
- The `area`[295] component, which generates the collision area for the ship to detect collisions.

Following that initial configuration, we're giving the player object three tags: `player`, `mobile` and `wraps`. Kaboom uses a tagging system to apply behavior to objects – you can think of this as similar to multiple inheritance[296]. By tagging objects with shared behavior, we can save ourselves from duplicating code.

Finally, we assign a number of custom properties to our player object. We'll use these properties to handle a variety of gameplay functions, such as moving and shooting.

If you run your repl now, you should see the ship sprite in the middle of a blank screen. In the next section, we'll implement movement controls.

[290]https://kaboomjs.com/doc/#add
[291]https://kaboomjs.com/#sprite
[292]https://kaboomjs.com/#pos
[293]https://kaboomjs.com/#rotate
[294]https://kaboomjs.com/#origin
[295]https://kaboomjs.com/#area
[296]https://en.wikipedia.org/wiki/Multiple_inheritance



Ship in space

Movement controls

In this game, our player will turn the nose of the spaceship with the left and right arrow keys, and thrust forward and backward with the up and down arrow keys. We'll handle movement in two phases: user input, and actually moving the ship. First, let's allow the player to turn their ship to the left and right. Add the following code:

```
// Movement keys
onKeyDown("left", () => {
    player.angle -= player.turn_speed;
});
onKeyDown("right", () => {
    player.angle += player.turn_speed;
});
```

Run your repl now, and you should be able to turn the ship to the left and right by pressing the respective arrow keys. If you think it turns too fast or too slow, change the value of `turn_speed` in the player creation code.

Now let's implement the up and down keys, so the player can move around the scene. Enter the following code beneath the player turning code:

```
onKeyDown("up", () => {
    player.speed = Math.min(player.speed+player.acceleration, player.max_thrust);
    play("rocket_thrust", {
        volume: 0.1,
        speed: 2.0,
    });
});
```

Rather than having the spaceship go from zero to max speed immediately, we want to simulate a gradual acceleration to our max speed. To achieve this, we use Math.min()[297] to set the player's speed to the minimum value between current speed plus acceleration and its maximum speed. This will make the ship gradually increase in speed until it reaches `max_thrust`. Play around with the values of `acceleration` and `max_thrust` in the player creation code and see what feels right to you.

Additionally, we set our rocket thrust sound to play[298] while accelerating. Kaboom allows us to manipulate sounds in a few different ways, such as changing their speed and volume.

We'll handle deceleration by doing the opposite to acceleration, using Math.max()[299] to choose the maximum between the player's speed minus their deceleration, and their maximum speed in reverse (i.e. negative). Add the following code below the acceleration controls:

```
onKeyDown("down", () => {
    player.speed = Math.max(player.speed-player.deceleration, -player.max_thrust);
    play("rocket_thrust", {
        volume: 0.2,
        speed: 2.0,
    });
});
```

If you run your repl now and try to accelerate or decelerate, the sound will play, but the ship will go nowhere. This is because we're manipulating `player.speed`, which is a custom property that Kaboom hasn't attached any special behavior to (unlike `player.angle`). Let's add some movement parameters now.

Movement

Movement in Kaboom and most other 2D game development systems is handled using X and Y coordinates on a plane. To move an object left, you subtract from its X coordinate, and to move it right, you add to its X coordinate. To move an object up, you subtract from its Y coordinate, and to move it down, you add to its Y coordinate. Therefore, basic four-directional movement in games

[297]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/min
[298]https://kaboomjs.com/doc/#play
[299]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/max

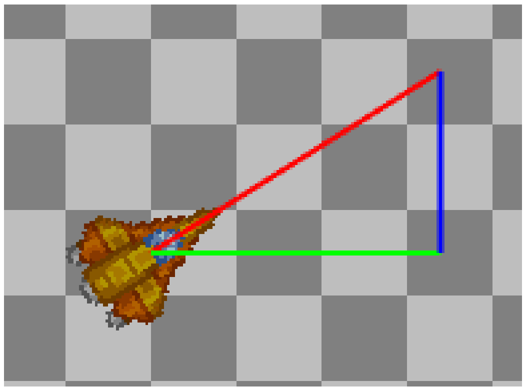
like Snake or Pacman is quite straightforward. The directional movement we need for Asteroids (commonly called tank controls[300]) is more complex, requiring some calculations.

At a high level, we want to move a given distance (`player.speed`) in a given direction (`player.angle`). As a first step, let's create an `onUpdate` callback for our `mobile` tag. This code, like our UI drawing code above, will be run by every object with the "mobile" tag on every frame of the game, so we can use it for more than just the player object. Add the following code at the bottom of your main scene:

```
// Movement
onUpdate("mobile", (e) => {
  e.move(pointAt(e.speed, e.angle));
});
```

First, we move our object, using the function `pointAt()`, which takes a speed and an angle and returns the corresponding X and Y co-ordinates as a `vec2`[301] object, Kaboom's 2D vector type. This data type is provided by Kaboom specifically for working with X and Y coordinates, and comes with a number of useful functions, such as addition and subtraction.

Now we need to create the `pointAt()` function. Before we jump into the code, let's think about the problem. Our movement can be drawn as an angled line on a plane, and its X and Y coordinates as horizontal and vertical lines connected to it, giving us a right-angled triangle.



Triangle

We know the values of the triangle's angles: one is 90 degrees, and the other two are equal to `player.angle`. The length of the triangle's hypotenuse (the red line in the diagram above) is `player.speed`: this is the *distance* our ship will be traveling.

We now need to calculate the lengths of the other two sides to get the X and Y values for our movement vector. We can do this using the trigonometric sine and cosine functions[302], like so:

[300]https://en.wikipedia.org/wiki/Tank_controls
[301]https://kaboomjs.com/doc/#vec2
[302]https://www2.clarku.edu/faculty/djoyce/trig/formulas.html

```
sine(angle) = y / distance
y = distance * sin(angle)

cosine(angle) = x / distance
x = distance * cos(angle)
```

Remember, in Kaboom and most other 2D game development platforms, the Y axis is inverted, so we have to make it negative. Add the `pointAt()` function below, at the top of the main scene code.

```
function pointAt(distance, angle) {
  let radians = -1*deg2rad(angle);
  return vec2(distance * Math.cos(radians), -distance * Math.sin(radians));
}
```

Run your repl now and take the ship for a spin. You should be able to thrust forward and backward, moving according to where the ship's nose is pointing.

There's one little problem though: thrust too long, and you'll fly off the screen. In any other game, we might solve that by enclosing the play area with walls, but that doesn't seem quite right in the infinite expanse of space. The original *Asteroids* solved this by having the player and other key objects wrap around the screen, i.e. appear at the bottom after going over the top, or at the left edge after going past the right edge. Let's implement that now, using the "wraps" tag we assigned to our player object when we created it. Add the following code to the bottom of your main scene:

```
// Wrap around the screen
onUpdate("wraps", (e) => {
    if (e.pos.x > width()) {
        e.pos.x = 0;
    }
    if (e.pos.x < 0) {
        e.pos.x = width();
    }
    if (e.pos.y > height()) {
        e.pos.y = 0;
    }
    if (e.pos.y < 0) {
        e.pos.y = height();
    }
});
```

This is a fairly straightforward piece of code that checks whether an object's position is outside of the room and, if so, places it on the other side. The `width()`[303] and `height()`[304] functions are Kaboom built-ins that return the size of the game canvas. Run your repl now and try it out.

[303]https://kaboomjs.com/doc/#width
[304]https://kaboomjs.com/doc/#height

Rocket animation

Our ship can move now, but it would be nice to see its rockets firing when the player presses the up arrow key, as well as hear them. Reactive animations like these make games feel more responsive and look more dynamic.

Kaboom has methods for handling animations when a game object uses an animated sprite (i.e. one with multiple images), but this isn't quite what we want here. Rather than animating the player's ship, we need to animate the rocket thrust that appears behind it when a thrusting key is pressed, so we'll need to handle all the animation code ourselves. Luckily, Kaboom makes this fairly simple.

First, let's create an array with our four rocket sprites. Add the following code at the bottom of your main scene:

```javascript
// Animate rocket
const thrust_animation = ["rocket1", "rocket2", "rocket3", "rocket4"];
```

Then we need some way to indicate when to start and stop the animation. We can use Kaboom's `onKeyPress`[305] and `onKeyRelease`[306] events for this, as well as two of the properties we defined for our player (`thrusting` and `animation_frame`). Add the following code:

```javascript
// rocket animation helpers
onKeyPress("up", () => {
    player.thrusting = true;
    player.animation_frame = 0;
});
onKeyRelease("up", () => {
    player.thrusting = false;
});
```

Now let's draw the animation. We'll use a draw event callback, which lets us make the player draw other things in addition to its own sprite. Add the following code:

[305] https://kaboomjs.com/doc#onKeyPress
[306] https://kaboomjs.com/doc/#onKeyRelease

```
// draw current rocket animation frame
onDraw("player", (p) => {
  if (player.thrusting) {
    // draw current frame
    drawSprite( {
      sprite: thrust_animation[p.animation_frame],
      pos: p.pos.add(pointAt(-p.height/2 , p.angle)),
      origin: "center",
      angle: p.angle
    });
  }
});
```

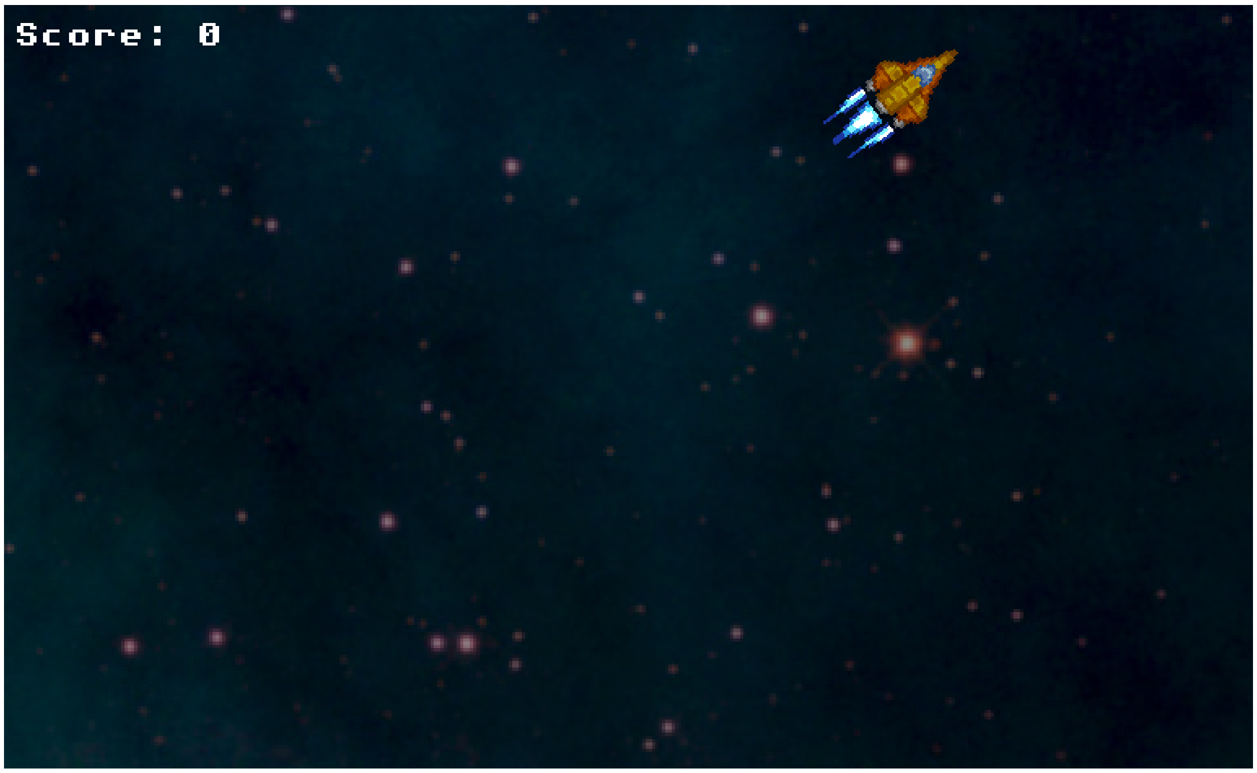
Here we're using our `pointAt` function again, but this time we're looking for the rocket end of the ship, rather than its nose. We use our `thrust_animation` array in conjunction with the player's `animation_frame` value to figure out which rocket image to draw.

To actually make the rocket animate (i.e. cycle through animation frames), we'll use Kaboom's `onUpdate`[307] function, and create a callback that changes the animation frame every 0.1 seconds. Add the following code:

```
let move_delay = 0.1;
let timer = 0;
// loop rocket animation
onUpdate(() => {
  timer += dt();
  if (timer < move_delay) return;
  timer = 0;

  if (player.thrusting) {
    player.animation_frame++;
    if (player.animation_frame >= thrust_animation.length) { // wrap to start
      player.animation_frame = 0;
    }
  }
});
```

That's it! Run your repl and move your ship around.

[307]https://kaboomjs.com/#onUpdate



Ship thrusting

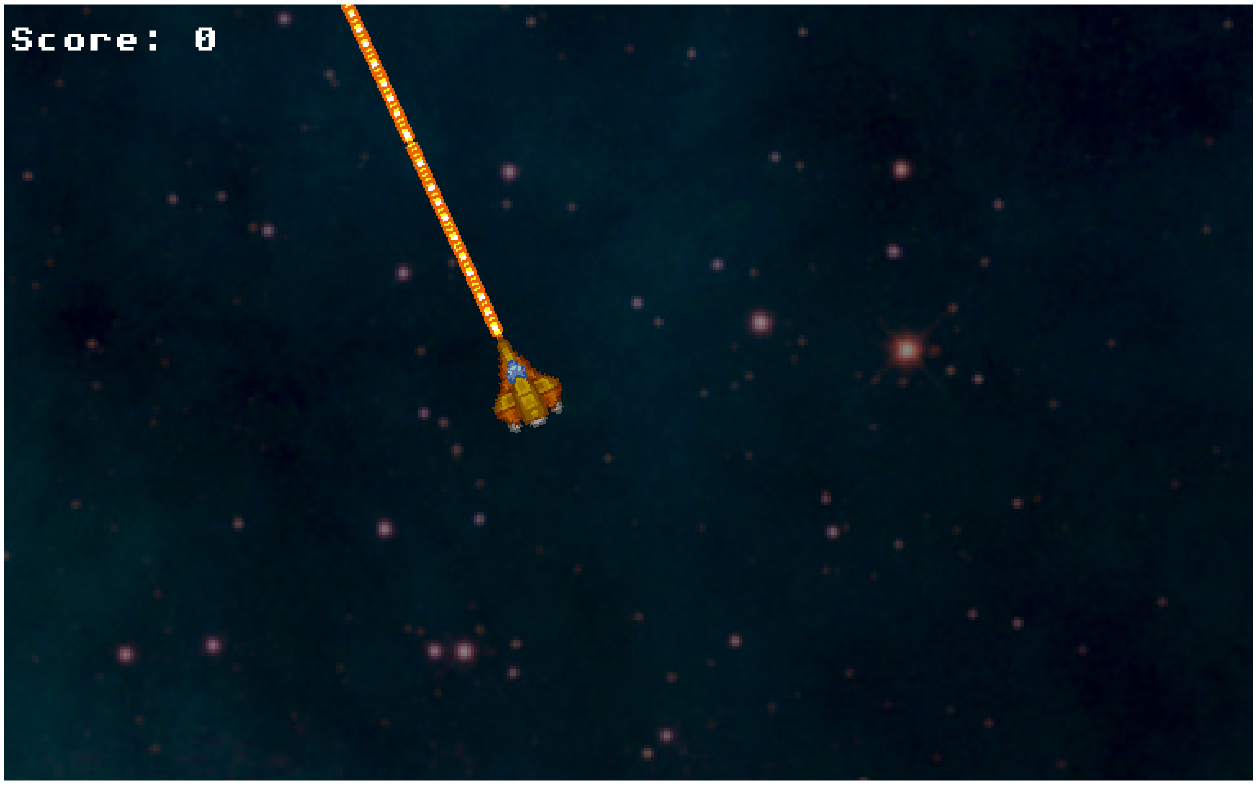
Shooting

To make our ship fire laser bullets, we need to create bullet objects just in front of the ship's nose, and have them travel in the same direction the ship is pointing. Here we can reuse our new `pointAt()` function. Add the following code beneath the existing movement control code:

```
// Shooting
onKeyDown("space", () => {
    add([
        sprite("bullet"),
        pos(player.pos.add(pointAt(player.width/2, player.angle))),
        rotate(player.angle),
        origin("center"),
        area(),
        "bullet",
        "mobile",
        "destructs",
        {
            speed: 100
        }
    ]);
    play("laser");
});
```

Here we're creating a bullet object at the tip of the ship's nose. We calculate the position of the tip by running our `pointAt()` function with a distance of half the ship sprite's width, and the sprite's angle. We use half the sprite's width because the ship sprite's origin is at its center. Additionally, we rotate the bullet according to the ship's angle (again using center as the origin) and assign a number of tags to it. Note that because we're tagging it with "mobile" and giving it both a `rotate` component and a speed property, it will use the same movement code we wrote for our player object.

Try the game out now, and hold down the spacebar to shoot. Do you notice anything about your ship's firing rate?



Too many bullets

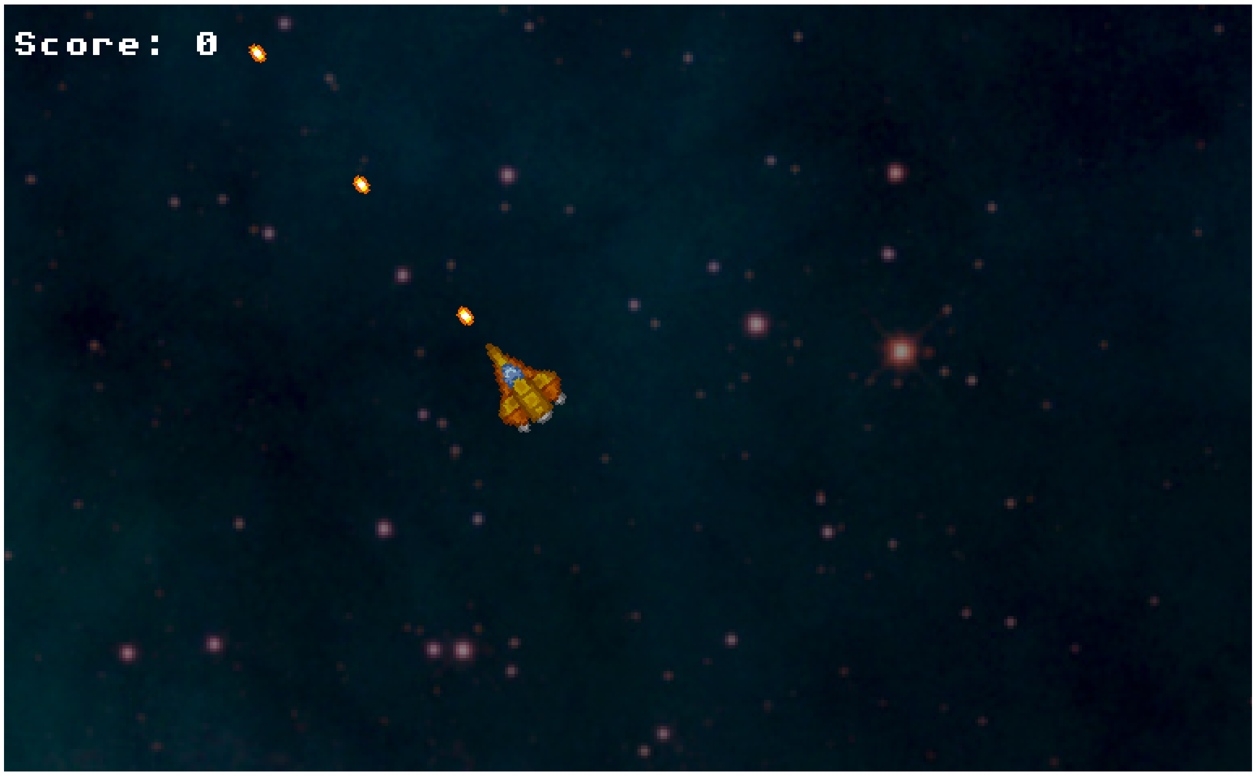
At the moment, a bullet object will be created in every frame while space is down. That's a lot of bullets, and might make the game too easy, as well as slowing it to a crawl on less capable devices. We need to add a cooldown period, and we can do so by altering our shooting code to look like this:

```
// Shooting
onKeyDown("space", () => {
    if (player.can_shoot) { // new if statement
        add([
            sprite("bullet"),
            pos(player.pos.add(pointAt(player.width/2, player.angle))),
            rotate(player.angle),
            origin("center"),
            area(),
            "bullet",
            "mobile",
            "destructs",
            {
                speed: 100
            }
        ]);
        play("laser");
        player.can_shoot = false; //
        wait(player.laser_cooldown, () => {
            player.can_shoot = true;
        });
    }
});
```

Here, we use two of the properties we defined in the player object above, `can_shoot` and `laser_cooldown`, to implement a cooldown mechanism. We will only create a bullet if `can_shoot` is true, and we set it to false immediately after each shot. Then we use Kaboom's `wait`[308] timer to set it to true after `laser_cooldown` number of seconds. Because this timer is an asynchronous callback[309], the rest of the game can continue while the laser cooldown period passes.

Run your repl and test whether the ship's laser fires at the expected intervals.

[308] https://kaboomjs.com/doc/#wait
[309] https://en.wikipedia.org/wiki/Callback_(computer_programming)



Laser rate

The asteroids

Now that we've added shooting, we need to add something to shoot at. It's time to create the asteroids this game gets its name from.

Creation

Add the following code at the bottom of the main scene:

```
// Asteroids
const NUM_ASTERIODS = 5;

for (let i = 0; i < NUM_ASTERIODS; i++) {
    var spawnPoint = asteroidSpawnPoint();
    var a = add([
        sprite("asteroid"),
        pos(spawnPoint),
        rotate(rand(1,90)),
        origin("center"),
        area(),
        solid(),
        "asteroid",
        "mobile",
        "wraps",
        {
            speed: rand(5, 10),
            initializing: true
        }
    ]);

    a.pushOutAll();

}
```

Here we're creating a constant number of asteroids, and assigning them a random position, direction of movement and speed. The asteroid creation code is largely similar to our player creation code, but with fewer custom properties. One key difference is the presence of the `solid`[310] component, which marks the asteroid as a solid object that other objects shouldn't be able to pass through.

The one custom property that's unique to asteroids is `initializing`. Because we're spawning each asteroid in a random position, there's a chance we might spawn one on top of another, or on top of the player.

One approach to avoiding this might be to ensure we don't spawn any two asteroids at the same coordinates, but we might still end up spawning them close enough to overlap with each other. We would then need to take into account the size of asteroids and prevent asteroids from spawning at any of those coordinates, and our code would quickly become complicated.

Instead of doing that, we can leverage Kaboom's collision detection to achieve the same effect. Right after we create the asteroid, we can check if it's overlapping with another "mobile"-tagged object (i.e. another asteroid, or the player's ship), and if so, we randomise its position again. We can use a while loop to repeat this action until our asteroid lands up in a free space. Add the following code inside the asteroid creation for-loop, below the `add` function:

```
    while (a.isColliding("mobile")) {
        spawnPoint = asteroidSpawnPoint();
        a.pos = spawnPoint;
        a.pushOutAll();
    }

    a.initializing = false;
```

[310] https://kaboomjs.com/doc/#solid

We want the asteroid to be in an "initializing" state while we're finding its starting position. When we implement its actions later on, we'll check the value of its `initializing` property to prevent it from harming the player or affecting other asteroids while it's still spawning.

Before we move on, let's implement the `asteroidSpawnPoint()` function. Add the following code near the top of the main scene, just beneath the `pointAt()` function:

```
function asteroidSpawnPoint() {
     // spawn randomly at the edge of the scene
     return choose([rand(vec2(0), vec2(width(), 0)),
             rand(vec2(0), vec2(0, height())),
             rand(vec2(0, height()), vec2(width(), height())),
             rand(vec2(width(), 0), vec2(width(), height()))]);
}
```

This function uses Kaboom's `choose()`[311] and `rand()`[312] functions to choose a random location on the edge of the scene to spawn an asteroid.

Collisions

If you've seen any movies set in outer space, you'll know that the main thing asteroids do is crash into spaceships and each other, even though the real-life asteroid belt[313] in our solar system is not nearly dense enough for asteroid collisions to be a frequent occurrence.

There are three types of collisions we need to account for:

- Player and asteroid, which damages the player, causing them to lose a life.
- Bullet and asteroid, which destroys the asteroid.
- Asteroid and asteroid, which causes both asteroids to bounce off each other.

Let's go through the code for each of these, adding it to our main scene just below the code we wrote to make objects wrap around the screen. First, player and asteroid:

```
// Collisions
onCollide("player", "asteroid", (p, a) => {
    if (!a.initializing) {
        p.lives--;
    }
});
```

This code reduces the player's lives by one as long as the asteroid is not initializing.

Next, add the code for collisions between a bullet and an asteroid:

[311] https://kaboomjs.com/doc/#choose
[312] https://kaboomjs.com/doc/#rand
[313] https://en.wikipedia.org/wiki/Asteroid_belt

```
onCollide("bullet", "asteroid", (b, a) => {
    if (!a.initializing) {
        destroy(b);
        destroy(a);
        play("explosion");
        score++;
    }
});
```

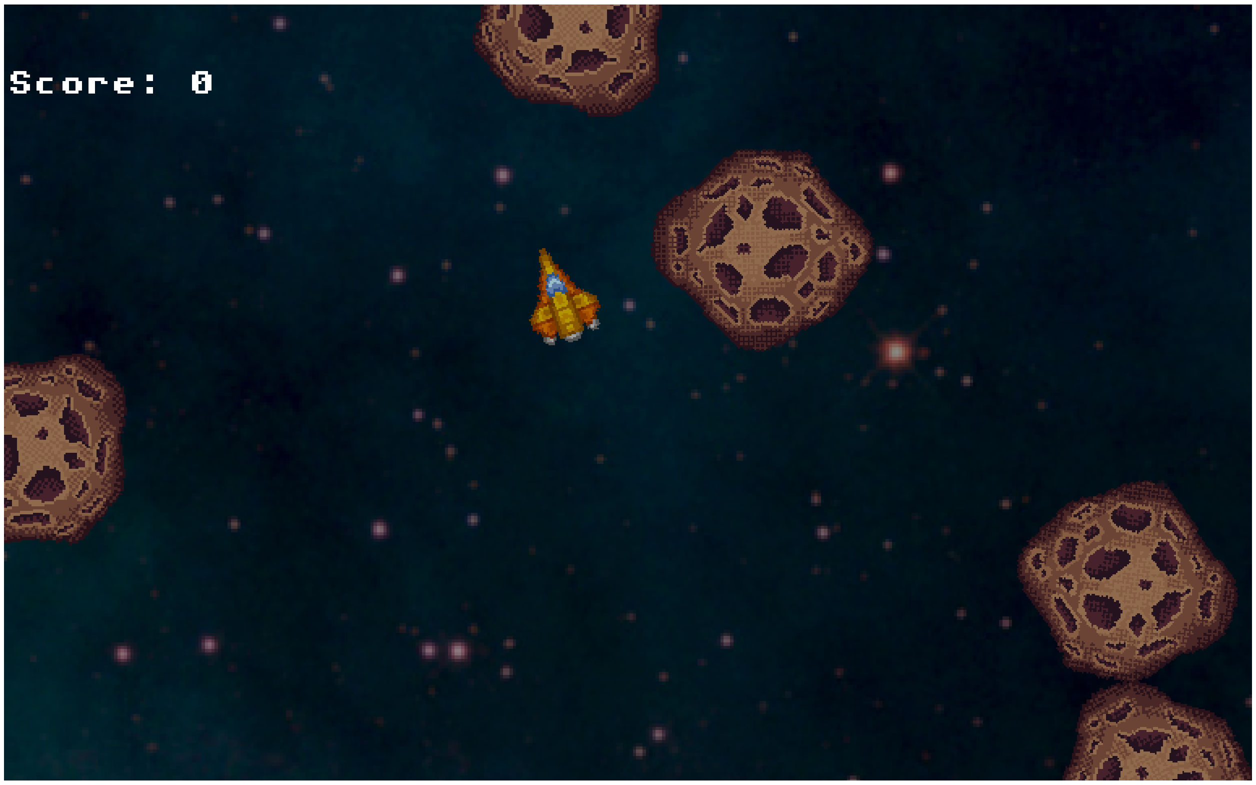
This very simple code destroys both the bullet and the asteroid, plays an explosion sound, and increments the game score.

Finally, add the following to handle asteroid collisions.

```
onCollide("asteroid", "asteroid", (a1, a2) => {
    if (!(a1.initializing || a2.initializing)) {
        a1.speed = -a1.speed;
        a2.speed = -a2.speed;
    }
});
```

This code makes the asteroids appear to bounce off each other by reversing their movement direction.

Run the game now and see what happens when you ram your ship into some asteroids!



Asteroids

Ending the game

The player's ship can now lose lives by crashing into asteroids, but this doesn't mean much at the moment, as we haven't added any code to end the game if the player runs out of lives, or even to display the lives remaining. Let's do that now. First, let's change the code in the player–asteroid collision to trigger[314] a custom "damage" event instead of just subtracting a life.

```
// Collisions
onCollide("player", "asteroid", (p, a) => {
    if (!a.initializing) {
        p.trigger("damage"); // previously lives--
    }
});
```

Now we can add the code to handle this event just below the collision code:

[314]https://kaboomjs.com/doc/#obj.trigger

```
// Take damage
player.on("damage", () => {
    player.lives--;

    // destroy ship if lives finished
    if (player.lives <= 0) {
        destroy(player);
    }
});
```

When objects are destroyed in Kaboom, the "destroy" event is triggered. We'll use this event to show a game over screen with the player's score by adding the following code:

```
// End game on player destruction
player.on("destroy", () => {
    add([
        text(`GAME OVER\n\nScore: ${score}\n\n[R]estart?`, { size: 20 }),
        pos(width()/2, height()/2),
        layer("ui")
    ]);
});
```

We need to give the player a way to restart the game and try again. Add the following code just below the previous block:

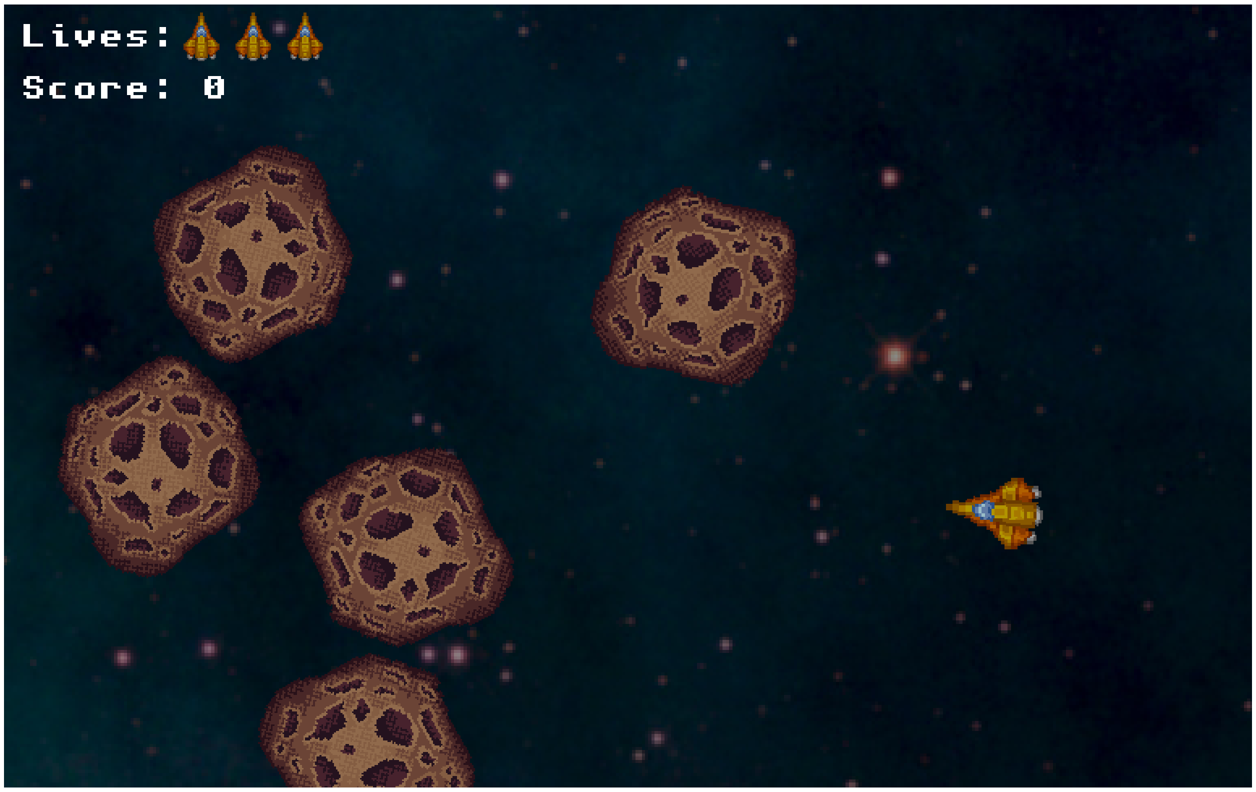
```
// Restart game
onKeyPress("r", () => {
    go("main");
});
```

The `go` function is a Kaboom built-in for moving between scenes. As this game only has one scene, we can use it to reset the scene and thus the game to its initial state.

Lastly, we need to add the player's lives to the game UI, so they know how much more damage they can afford to take. Find the `ui` object code near the top of the main scene and alter it to resemble the below:

```
ui.on("draw", () => {
  drawText({
    text: "Score: " + score,
    size: 14,
    font: "sink",
    pos: vec2(8, 24)
  })

  // lives (new code below)
  drawText({
    text: "Lives: ",
    size: 12,
    font: "sink",
    pos: vec2(8),
  });
  for (let x = 64; x < 64 + (16 * player.lives); x += 16) {
    drawSprite({
      sprite: "ship",
      pos: vec2(x, 12),
      angle: -90,
      origin: "center",
      scale: 0.5
    });
  }
});
```

This code draws a number of scaled down player spaceships equal to the number of remaining lives.



Lives

Final touches

Our game is fully playable now, but it's still missing some niceties, and one core gameplay feature that you should be able to identify if you've played *Asteroids* before. In this final section, we'll add the following:

- Some background music.
- Smaller, faster asteroids that our large asteroids break into when destroyed.
- Temporary invulnerability for the player for a few seconds after they take damage.

Background music

Add the following code somewhere in your main scene.

```
// Background music
const music = play("Steamtech-Mayhem_Looping");
music.loop();
```

The first line plays[315] the piece Steamtech Mayhem from Soundimage.org[316] and the second line will ensure that it repeats as long as the game is running.

Smaller asteroids

To create smaller asteroids when a large asteroid is destroyed, we'll use a destroy event callback, which will run every time an asteroid is destroyed. Add the following code to the bottom of your main scene:

```
// Asteroid destruction
on("destroy", "asteroid", (a) => {
    if (!a.is("small")) {
        // create four smaller asteroids at each corner of the large one
        positions = [a.pos.add(vec2(a.width/4, -a.height/4)),
                    a.pos.add(vec2(-a.width/4, -a.height/4)),
                    a.pos.add(vec2(-a.width/4, a.height/4)),
                    a.pos.add(vec2(a.width/4, a.height/4))];

        // small asteroids move out from the center of the explosion
        rotations = [16,34,65,87];

        for (let i = 0; i < positions.length; i++) {
            var s = add([
                sprite(`asteroid_small${i+1}`),
                pos(positions[i]),
                rotate(rotations[i]),
                origin("center"),
                area(),
                solid(),
                "asteroid",
                "small",
                "mobile",
                "wraps",
                {
                    speed: rand(15, 25),
                    initializing: false
                }
            ]);

            s.pushOutAll();
```

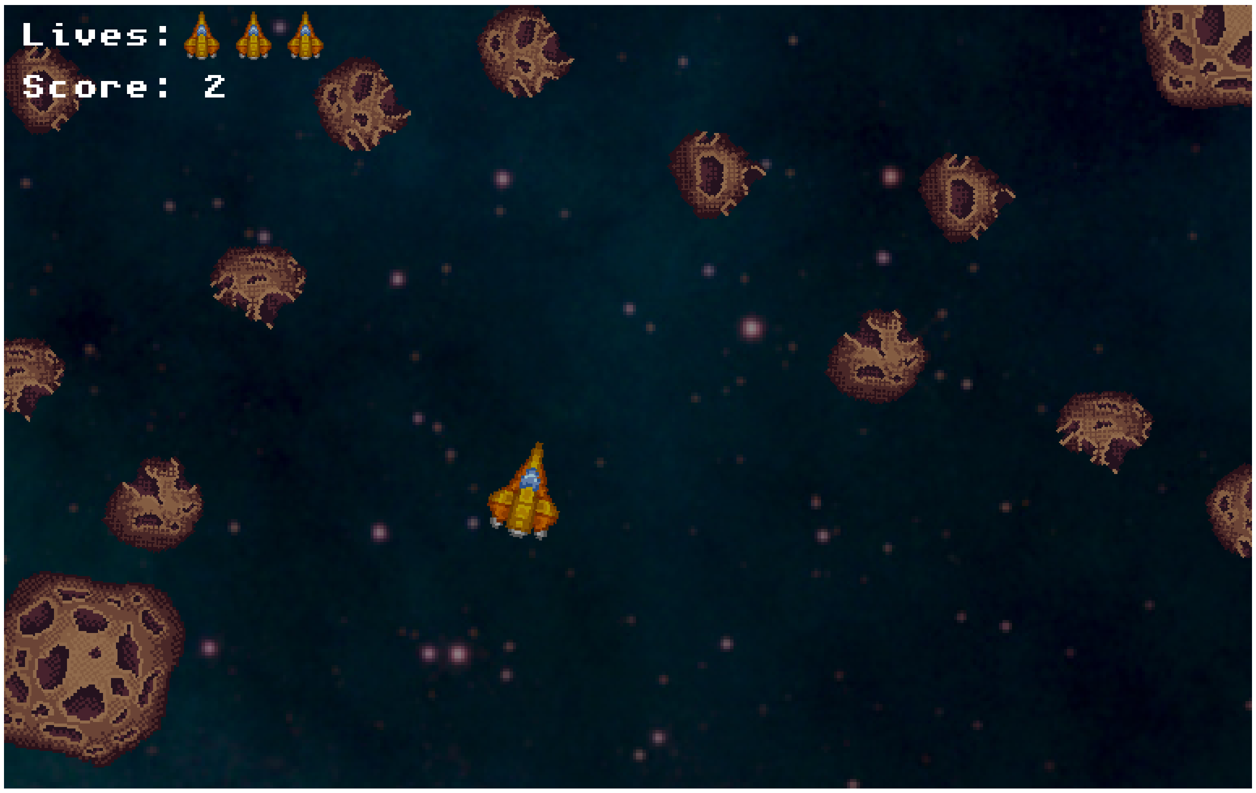
[315] https://kaboomjs.com/doc/#play
[316] https://soundimage.org/sci-fi/

```
        }
    }
});
```

Our small asteroids are mostly similar to our large ones. Differences include the addition of the `small` tag, the less random approach to initial placement, the higher speed, and the selection of one of four different possible small asteroid sprites.

To make our game true to the original *Asteroids*, we should give the player more points for destroying these fast, small asteroids. Find and modify the bullet–asteroid collision code as below:

```
onCollide("bullet", "asteroid", (b, a) => {
    if (!a.initializing) {
        destroy(b);
        destroy(a);
        play("explosion");
        score = a.is("small") ? score + 2 : score++; // 2 points for small, 1 for big
    }
});
```



Small asteroids

Temporary invulnerability

A nice touch to make our game a little more forgiving is temporary invulnerability for the player after they lose a life. We can add this by finding and altering the player's damage event callback as follows:

```
// Take damage
player.on("damage", () => {
    if (!player.invulnerable) { // new if statement
        player.lives--;
    }

    // destroy ship if lives finished
    if (player.lives <= 0) {
        destroy(player);
    }
    else // new code
    {
        // Temporary invulnerability
        player.invulnerable = true;

        wait(player.invulnerablity_time, () => {
            player.invulnerable = false;
            player.hidden = false;
        });
    }
});
```

Here we're making the player invulnerable and then using a `wait` callback to make them vulnerable again after a given number of seconds, similar to what we did for the laser timeout. We're also making sure the player is visible by setting `player.hidden`[317] to false, because the way we're going to indicate the player's invulnerability is by having their ship flash rapidly. Find and update the `onUpdate` event callback we added earlier for rocket thrust animation :

[317]https://kaboomjs.com/doc/#add

```
onUpdate(() => {
  timer += dt();
  if (timer < move_delay) return;
  timer = 0;

  if (player.thrusting) {
    player.animation_frame++;
    if (player.animation_frame >= thrust_animation.length) { // wrap to start
      player.animation_frame = 0;
    }
  }

  // new if statement
  if (player.invulnerable) {
    player.hidden = !player.hidden;
  }
});
```

Where to next?

We've covered a lot of ground in this tutorial and touched on a lot of Kaboom's features. From here, you can start making your own games with Kaboom, or if you want to extend this one, here are some ideas:

- Power-ups, such as extra lives and time-limited weapon upgrades.
- Enemy spaceships that fire at the player.
- A third, even smaller size of asteroid.
- More animations, for explosions and laser firing.

Code

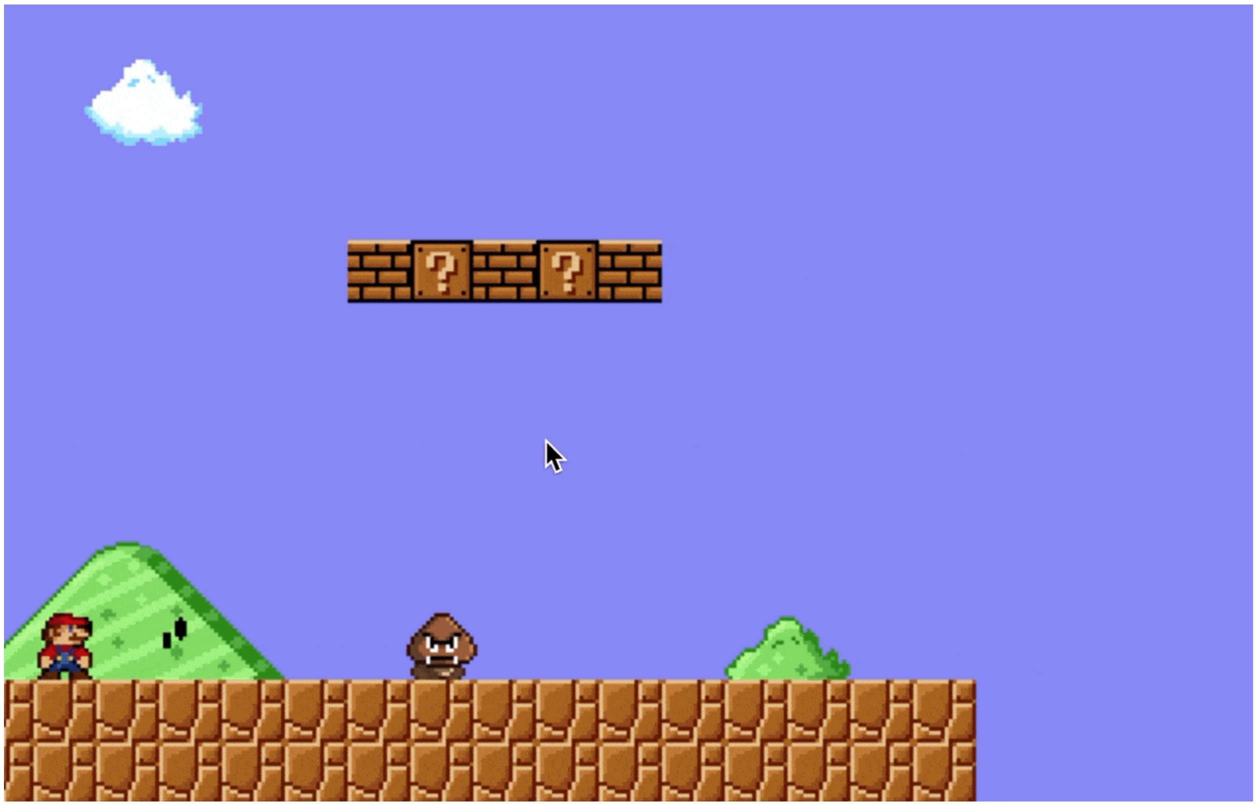
You can find the code for this tutorial on Replit[318]

[318]https://replit.com/@ritza/Asteroids-new

Building a *Mario*-like side-scroller with Kaboom.js

The *Mario* series is one of the most known and loved game series of all time. The first *Mario* game was released by Nintendo in the mid-80s, and people haven't stopped playing *Mario* since.

Tons of games still use the basic side-scroller formula of *Mario*, so it's a good game to build to learn the basics of game making. We'll build it in the new Kaboom[319] game engine. Kaboom has many useful functions for building platform games, and we'll try to go through as many as we can in this tutorial.



The finished game

Click to open gif[320]

[319]https://kaboomjs.com
[320]https://docs.replit.com/images/tutorials/32-mario-kaboom/bigger-kill-scenes.gif

Designing the game

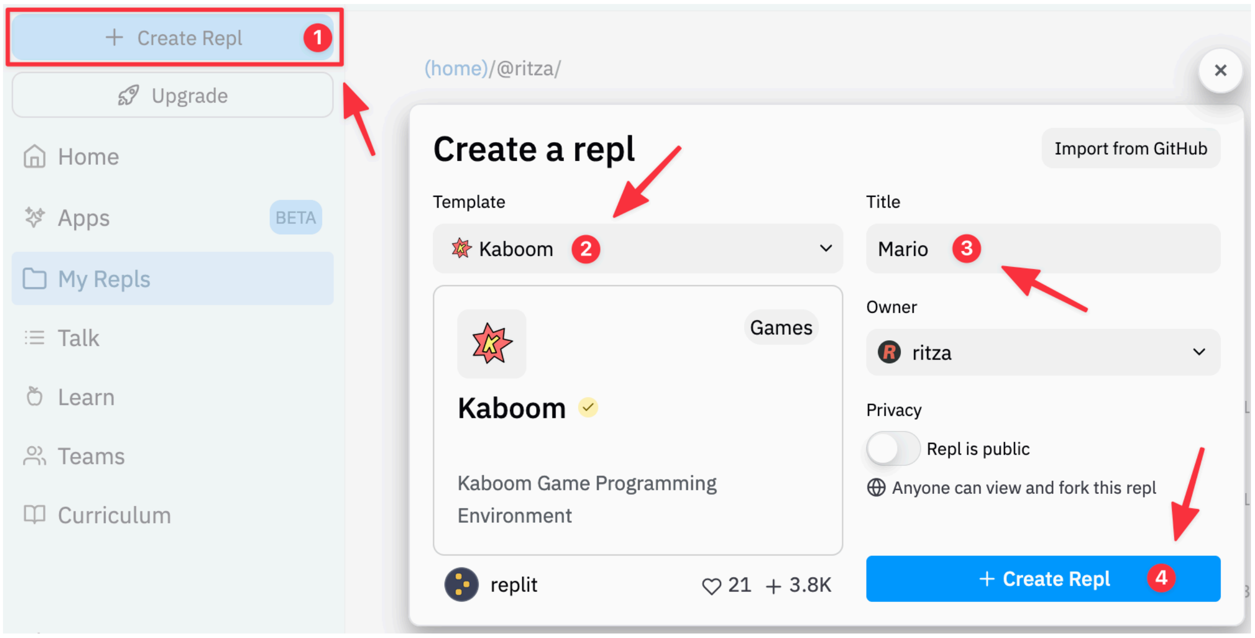
We'd like to make a game that has the *Mario* essence. This means we need a few things:

- The ability to jump and bump into reward boxes.
- A big and small character type.
- The ability to attack enemies by jumping on them.
- The classic *Mario* scrolling and camera motion.

For the graphics, we will use a tile set from this creator[321].

Creating a new project in Replit

Head over to Replit[322] and create a new repl. Choose **Kaboom** as your project type. Give this repl a name, like "Mario".



Template: Kaboom 2

Title: Mario 3

Owner: ritza

Privacy: Repl is public

Anyone can view and fork this repl

Kaboom Game Programming Environment

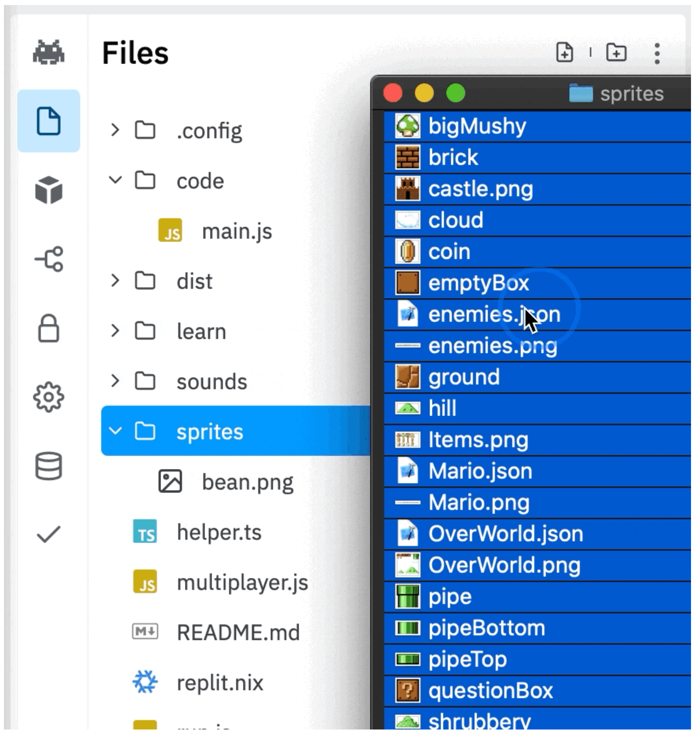
New Mario repl

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

Download this archive of sprite and asset files[323] that we'll 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

[321] https://dotstudio.itch.io/super-mario-1-remade-assets
[322] https://replit.com
[323] https://docs.replit.com/tutorial-files/mario-kaboom/mario-resources.zip

the sprite and asset files into the "sprites" folder. Once they have uploaded, you can click on the "Kaboom" icon in the sidebar, and return to the "main" code file.



Uploading sprites

Click to open gif[324]

Setting up Kaboom

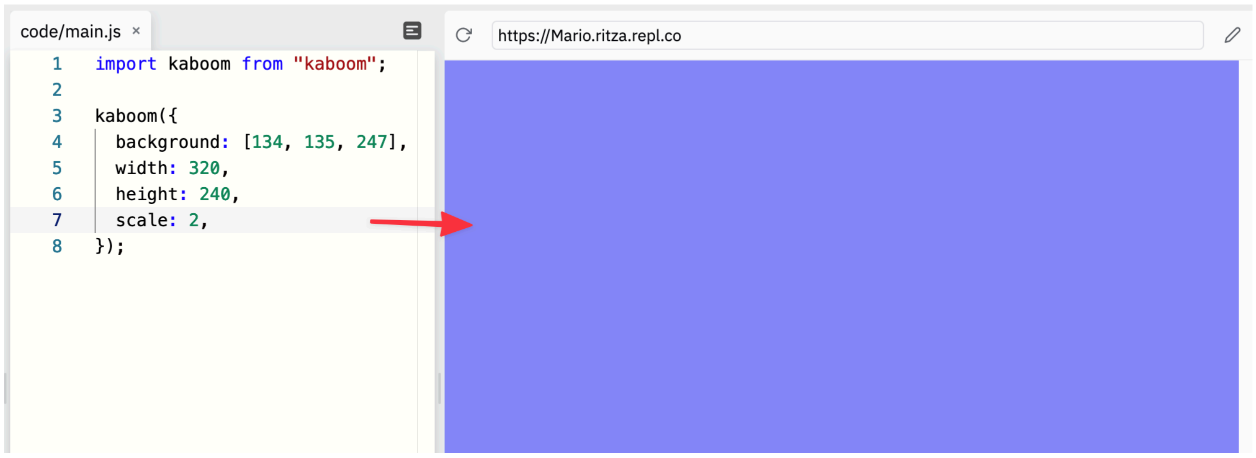
To start, we need to set up Kaboom with the screen size and colors we want for the game window. Replace the code in `main.js` with the code below:

```
import kaboom from "kaboom";

kaboom({
  background: [134, 135, 247],
  width: 320,
  height: 240,
  scale: 2,
});
```

This creates a new Kaboom canvas with a nice *Mario* sky-blue background. We also set the size of the view to 320x240 pixels, which is a very low resolution for a modern game, but the right kind of

[324]https://docs.replit.com/images/tutorials/32-mario-kaboom/upload-assets.gif

pixelation for a *Mario*-type remake. We use `scale` to make the background twice the size on screen - you can increase this value if you have a monitor with very high resolution. Click the "Run" button, and you should see a lovely blue sky in the output window.



```
import kaboom from "kaboom";

kaboom({
  background: [134, 135, 247],
  width: 320,
  height: 240,
  scale: 2,
});
```

Blue sky

Now, let's load up some of the sprites so we can add them to the blue sky scene. This code loads each of the graphic elements we'll use, and gives them a name so we can refer to them when we build the game characters:

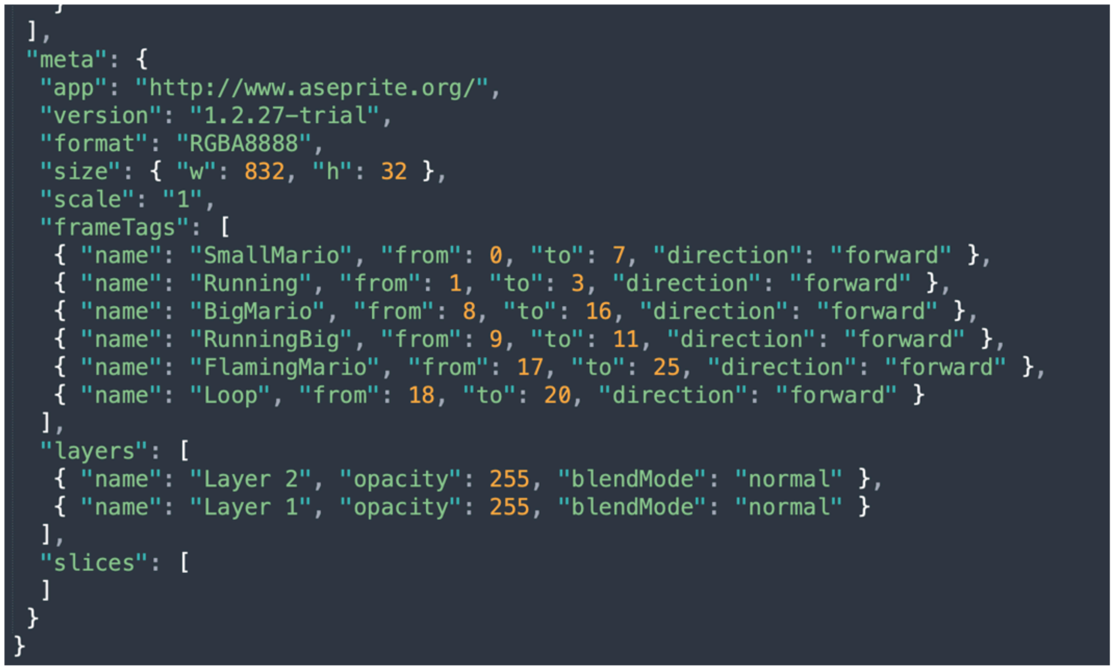
```
loadRoot("sprites/");
loadAseprite("mario", "Mario.png", "Mario.json");
loadAseprite("enemies", "enemies.png", "enemies.json");
loadSprite("ground", "ground.png");
loadSprite("questionBox", "questionBox.png");
loadSprite("emptyBox", "emptyBox.png");
loadSprite("brick", "brick.png");
loadSprite("coin", "coin.png");
loadSprite("bigMushy", "bigMushy.png");
loadSprite("pipeTop", "pipeTop.png");
loadSprite("pipeBottom", "pipeBottom.png");
loadSprite("shrubbery", "shrubbery.png");
loadSprite("hill", "hill.png");
loadSprite("cloud", "cloud.png");
loadSprite("castle", "castle.png");
```

The first line, `loadRoot`[325], specifies which folder to load all the sprites and game elements from, so we don't have to keep typing it in for each sprite. Then each line loads a game sprite and gives it a name so that we can refer to it in code later.

[325]https://kaboomjs.com/#loadRoot

Notice that the `mario` and `enemies` sprites are loaded with the function `loadAseprite`[326], and have an extra parameter specifying a `.json` file. This extra file is in a file format made by Aseprite[327], which is a pixel art and animation app. If you open the `Mario.png` file, you'll see that it has many different images of Mario in different positions, which are frames of Mario animations. The `.json` file from Aseprite contains all the information needed to animate Mario in our game. Kaboom knows how to interpret this file, and we can pick which animation we want to run at any time by choosing one from the `frameTags` list in the `.json` file and using the `.play()`[328] method on a sprite. We can also choose a particular frame to show at any time, using the sprite's `.frame`[329] property, and specifying the frame number to use, starting from `0`.

The contents of the Mario.png file


```
  ],
  "meta": {
   "app": "http://www.aseprite.org/",
   "version": "1.2.27-trial",
   "format": "RGBA8888",
   "size": { "w": 832, "h": 32 },
   "scale": "1",
   "frameTags": [
    { "name": "SmallMario", "from": 0, "to": 7, "direction": "forward" },
    { "name": "Running", "from": 1, "to": 3, "direction": "forward" },
    { "name": "BigMario", "from": 8, "to": 16, "direction": "forward" },
    { "name": "RunningBig", "from": 9, "to": 11, "direction": "forward" },
    { "name": "FlamingMario", "from": 17, "to": 25, "direction": "forward" },
    { "name": "Loop", "from": 18, "to": 20, "direction": "forward" }
   ],
   "layers": [
    { "name": "Layer 2", "opacity": 255, "blendMode": "normal" },
    { "name": "Layer 1", "opacity": 255, "blendMode": "normal" }
   ],
   "slices": [
   ]
  }
}
```
The contents of the Mario.json file. Each frametag is an animation sequence

Mario Aseprite file

Creating the level maps

Let's add 2 levels to start. You can create and add as many levels as you want - that's one of the great benefits of writing your own game!

[326]https://kaboomjs.com/#loadAseprite
[327]https://www.aseprite.org
[328]https://kaboomjs.com/#sprite
[329]https://kaboomjs.com/#sprite

Kaboom has a really cool way of defining levels. It allows us to draw a layout of the level using only text. Each letter or symbol in this text map can be mapped to a character in the Kaboom game. In Kaboom, characters are anything that makes up the game world, including floor, platforms, and so on, and not only the players and bots.

Add the following to define the levels:

```
const LEVELS = [
  [
    "                                                                               \
                 ",
    "                                                                               \
                 ",
    "                                                                               \
                 ",
    "                                                                               \
                 ",
    "                                                                               \
                 ",
    "                                                                               \
                 ",
    "                                                                               \
                 ",
    "      -?-b-                                                                    \
                 ",
    "                                                    ?        ?                 \
                 ",
    "                                                                               \
                 ",
    "                                      _                 ?                      \
                 ",
    "                                 _    |                                        \
                 ",
    "                           _     |    |                _                       \
                 ",
    "       E                   |     |    |   E   E        |                       \
     H           ",
    "================     ==========================================================\
=================",
    "================     ==========================================================\
=================",
  ],
  [
    "                                                                               \
              ",
    "                                                                               \
              ",
    "                                                                               \
              ",
    "                                       ?                                       \
              ",
    "                                                                               \
              ",
    "                                   -?-                                         \
              ",
    "                                                                               \
              ",
    "      -?-b-                  -?-                                               \
              ",
    "                                                                               \
              ",
    "                                                                               \
              ",
    "                                                                               \
              ",
    "                                                                               \
              ",
    "       _                                            _                          \
              ",
    "       |                                            |          E    E          \
  H           ",
    "================     ==========================================================\
==============",
    "================     ==========================================================\
==============",
  ]
];
```

Now we can map each symbol and letter in the levels to a character definition[330]:

[330]https://kaboomjs.com/#addLevel

```
const levelConf = {
  // grid size
  width: 16,
  height: 16,
  pos: vec2(0, 0),
  // define each object as a list of components
  "=": () => [
    sprite("ground"),
    area(),
    solid(),
    origin("bot"),
    "ground"
  ],
  "-": () => [
    sprite("brick"),
    area(),
    solid(),
    origin("bot"),
    "brick"
  ],
  "H": () => [
    sprite("castle"),
    area({ width: 1, height: 240 }),
    origin("bot"),
    "castle"
  ],
  "?": () => [
    sprite("questionBox"),
    area(),
    solid(),
    origin("bot"),
    'questionBox',
    'coinBox'
  ],
  "b": () => [
    sprite("questionBox"),
    area(),
    solid(),
    origin("bot"),
    'questionBox',
    'mushyBox'
  ],
  "!": () => [
    sprite("emptyBox"),
    area(),
    solid(),
   // bump(),
    origin("bot"),
    'emptyBox'
  ],
  "c": () => [
    sprite("coin"),
    area(),
    solid(),
    //bump(64, 8),
    cleanup(),
    lifespan(0.4, { fade: 0.01 }),
    origin("bot"),
    "coin"
  ],
  "M": () => [
    sprite("bigMushy"),
    area(),
    solid(),
    //patrol(10000),
    body(),
    cleanup(),
    origin("bot"),
    "bigMushy"
  ],
  "|": () => [
    sprite("pipeBottom"),
    area(),
    solid(),
    origin("bot"),
    "pipe"
  ],
  "_": () => [
    sprite("pipeTop"),
    area(),
    solid(),
    origin("bot"),
    "pipe"
  ],
  "E": () => [
    sprite("enemies", { anim: 'Walking' }),
    area({ width: 16, height: 16 }),
    solid(),
    body(),
    //patrol(50),
    //enemy(),
    origin("bot"),
    "badGuy"
  ],
  "p": () => [
    sprite("mario", { frame: 0 }),
    area({ width: 16, height: 16 }),
    body(),
    //mario(),
    //bump(150, 20, false),
    origin("bot"),
    "player"
  ]
};
```

That looks like a lot, but it's really one pattern repeated for each element. Let's take it apart.

The first 3 lines of the config set the default width and height of each element in the level maps. We use `pos` to specify where to position the whole map in the Kaboom canvas. This is normally going to be (0,0), i.e. at the top left of the screen.

Next we have definitions for each of the symbols we used in the map. Each definition is a function that returns an array of components[331]. In Kaboom, every character is made up of 1 or more components. Components give special properties to each character. There are built-in components for many properties, like `sprite`[332] to give the character an avatar, `body`[333], to make the character respond to gravity, `solid`[334] to make the character solid so other characters can't move through it, and many others.

Kaboom also allows you to write your own custom components to create any property or behavior you like for a character. The components `patrol`, `mario`, `enemy`, and `bump` are all custom here. You'll notice that those custom components are commented out (`//`), as we'll need to create the implementations for them before we can use them. We'll do that later in this tutorial.

Adding a scene

Kaboom "scenes"[335] allow us to group logic and levels together. In this game we'll have 2 scenes:

[331] https://kaboomjs.com/doc/intro
[332] https://kaboomjs.com/#sprite
[333] https://kaboomjs.com/#body
[334] https://kaboomjs.com/#solid
[335] https://kaboomjs.com/#scene

- A "start" or intro scene, which waits for a player to press a button to start the game. We'll also return to this scene if the player dies, so they can start again.
- A main "game" scene, which will contain the game levels and all the logic to move Mario, and the logic for the enemies and rewards, etc.

We can use the `go`[336] function to switch between scenes.

Let's add the "start" scene, and make the game go to that scene by default:

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

  add([
    text("Press enter to start", { size: 24 }),
    pos(vec2(160, 120)),
    origin("center"),
    color(255, 255, 255),
  ]);

  onKeyRelease("enter", () => {
    go("game");
  })
});

go("start");
```

We define the scene using the `scene`[337] function. This function takes a string as the scene name – we're calling the scene "start". We're using an inline function here, using arrow function notation[338]. You could also use the `function` keyword, if you'd like to specify a function in that way.

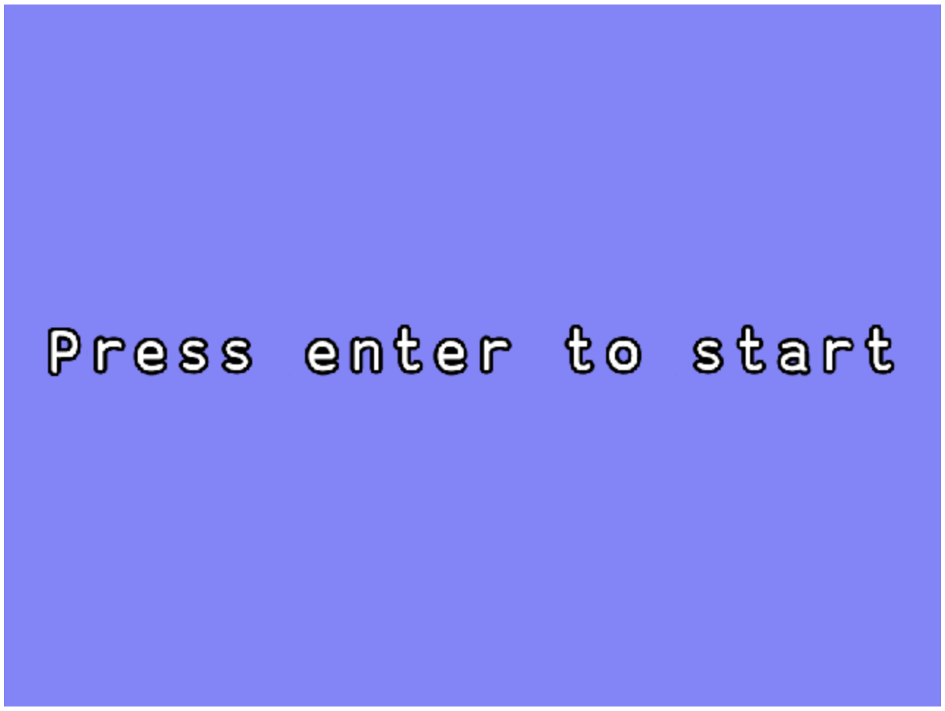
We also add some instruction text in the scene function, using the `text`[339] component and setting the text's content and size. The `pos`[340] component sets the position of the text on the screen, and the `origin`[341] component specifies that the center of the text should be used to position it. Finally, we set the color of the text to white using the `color`[342] component, which takes RGB (red, green, blue) values from `0`-`255`.

We also have a call to the function `onKeyRelease`[343], which listens for the enter key being pressed. If the enter key is pressed, we `go` to the main game scene (which we'll add shortly!).

Finally, we use the `go`[344] function to go to the start scene when the game starts up.

[336]https://kaboomjs.com/#go
[337]https://kaboomjs.com/#scene
[338]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions
[339]https://kaboomjs.com/#text
[340]https://kaboomjs.com/#pos
[341]https://kaboomjs.com/#origin
[342]https://kaboomjs.com/#color
[343]https://kaboomjs.com/#onKeyRelease
[344]https://kaboomjs.com/#go

After copying the code into your repl, press Command + S (Mac) or Control + S (Windows/Linux) to update the output window. You should see something like this:



Start scene

Note that if you push enter now to start the game, you'll get an error message. Don't worry, we'll sort that out soon.

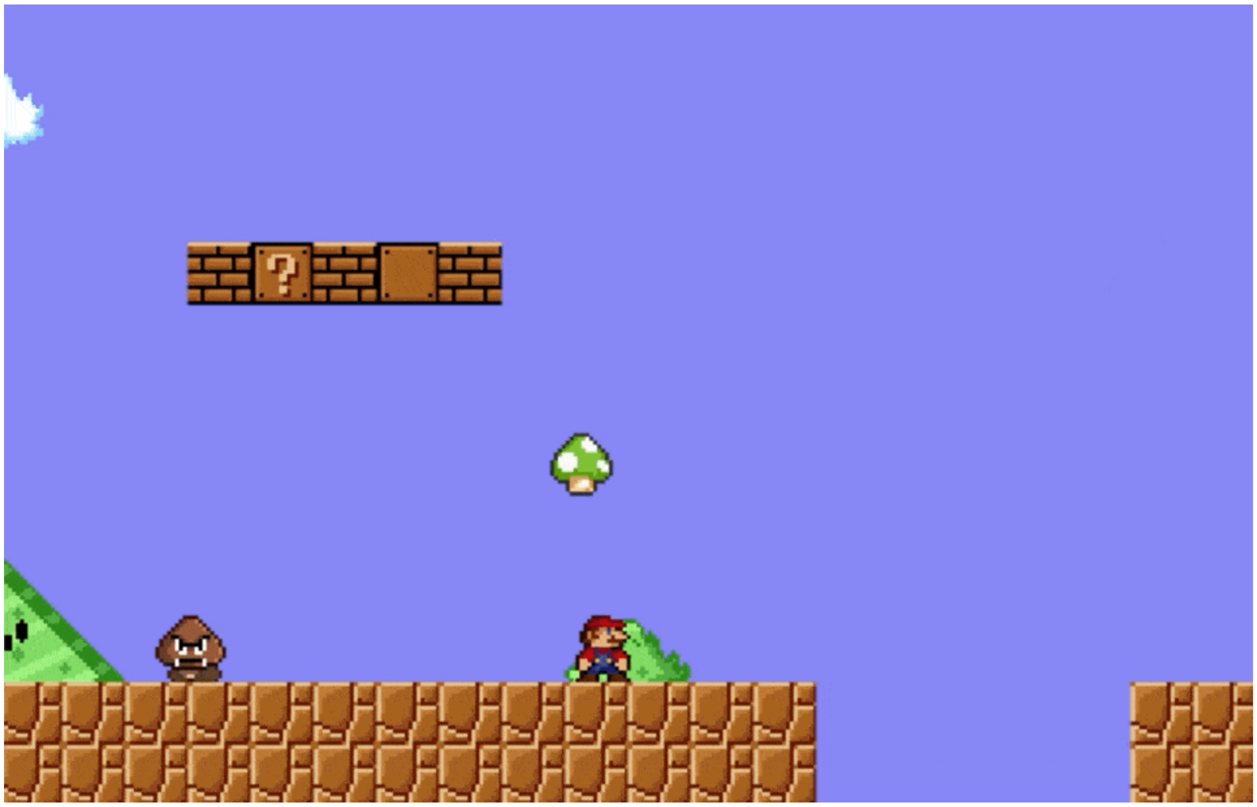
Adding the main game scene

Now let's get the game scene up and running. Add the code below:

```
scene("game", (levelNumber = 0) => {

  layers([
    "bg",
    "game",
    "ui",
  ], "game");


  const level = addLevel(LEVELS[levelNumber], levelConf);

  add([
    sprite("cloud"),
    pos(20, 50),
    layer("bg")
  ]);

  add([
    sprite("hill"),
    pos(32, 208),
    layer("bg"),
    origin("bot")
  ])

  add([
    sprite("shrubbery"),
    pos(200, 208),
    layer("bg"),
    origin("bot")
  ])

  add([
    text("Level " + (levelNumber + 1), { size: 24 }),
    pos(vec2(160, 120)),
    color(255, 255, 255),
    origin("center"),
    layer('ui'),
    lifespan(1, { fade: 0.5 })
  ]);

  const player = level.spawn("p", 1, 10)

});
```

Press Command + S or Control + S now, and push enter at the start screen prompt. You should see our replica of *Mario World* instead of the error message.

You should also see the enemy character wobble its feet, like it's trying to walk. This is because when we added the enemy definition `E` in the level config, we specified in the `sprite` component that it must use the `Walking` animation, which is defined in the `enemies.json` file. Kaboom starts the character using that animation.



Static Mario world

Click to open gif[345]

Ok, back to looking at the code we added and what it does. First, we define a new `scene`[346] like we did for the start scene. This time, we specify a parameter `levelNumber` that can be passed to the scene. We give this parameter a default value of `0`. This will be the first level in our `LEVELS` array - remember, arrays start at index `0`, so `0` is level `1`. This parameter will let us call the same scene again when we get to the end of the level, but with `1` as the parameter so that we can play the next level. You can specify any parameters you like or need when creating a scene, and you can pass values from one scene to another. This is very useful, for example if you want to pass the player score to an end game scene, or pass in player options from the start scene.

Next, we define some `layers`[347]. Layers allow us to have backgrounds that don't affect the game - we call that layer `bg`. We'll place all the main game objects (like our hero, his enemies, and any other objects he interacts with) on the `game` layer, and all the UI elements (like current score, health, etc.) on the `ui` level. We make `game` the default layer, so if we don't specify a layer component on a game object, it will be drawn on the `game` layer.

In the next line of code, we initialize and create the level by calling the `addLevel`[348] function. Here we pass in the level from the `LEVELS` array, using the index from the `levelNumber` parameter we

[345]https://docs.replit.com/images/tutorials/32-mario-kaboom/mario-world.gif
[346]https://kaboomjs.com/#scene
[347]https://kaboomjs.com/#layers
[348]https://kaboomjs.com/#addLevel

added to the scene. We also pass in the configuration for all the symbols in the level map that we assigned to the `levelConf` variable. At this point, the map and all the characters in it are drawn to the screen. Note that, because the map is much wider than the size we set in the settings for the Kaboom window, we only see part of the map. This is great, because it will allow us to show more of the map as Mario starts walking.

Then we add a few elements to the background layer - clouds, hills and shrubberies. Note the use of the `layer` component on these elements. We do this so that they don't interact with our game objects - they just add some visual interest. You can add as many as you like - the original *Mario* has them in a repeating pattern across the whole level.

We also add some temporary text to the `ui` layer to let the player know which level they are on. Notice that we use the `lifespan`[349] component here to automatically fade out and remove the info text after 1 second.

Finally, we add Mario to the game! We could have added him by placing his symbol, `p`, on our level map definition. However, by adding him manually to the scene using `level.spawn()`[350], we can get a reference to him. This will be useful later when we are dealing with collisions and other interactions. We also set the position we want the character to initially be placed at.

Making Mario move

The scene is all set up, so let's add in some interaction. The player will use the arrow keys to move Mario left and right, and the space bar to make him jump. We'll use the `onKeyDown`[351] function for moving left and right, as we want Mario to keep moving as long as the player holds down either key. Then we can use the `onKeyPress`[352] function to make Mario jump. The player will need to push the space key each time they want Mario to jump - it's always fun to smash buttons! Add the following code at the bottom of the game scene:

```
  const SPEED = 120;

  onKeyDown("right", () => {
    player.flipX(false);
    player.move(SPEED, 0);
  });

  onKeyDown("left", () => {
    player.flipX(true);
    if (toScreen(player.pos).x > 20) {
      player.move(-SPEED, 0);
```

[349]https://kaboomjs.com/#lifespan
[350]https://kaboomjs.com/#spawn
[351]https://kaboomjs.com/#onKeyDown
[352]https://kaboomjs.com/#onKeyPress

```
    }
  });

  onKeyPress("space", () => {
    if (player.grounded()) {
      player.jump();
    }
  });
```

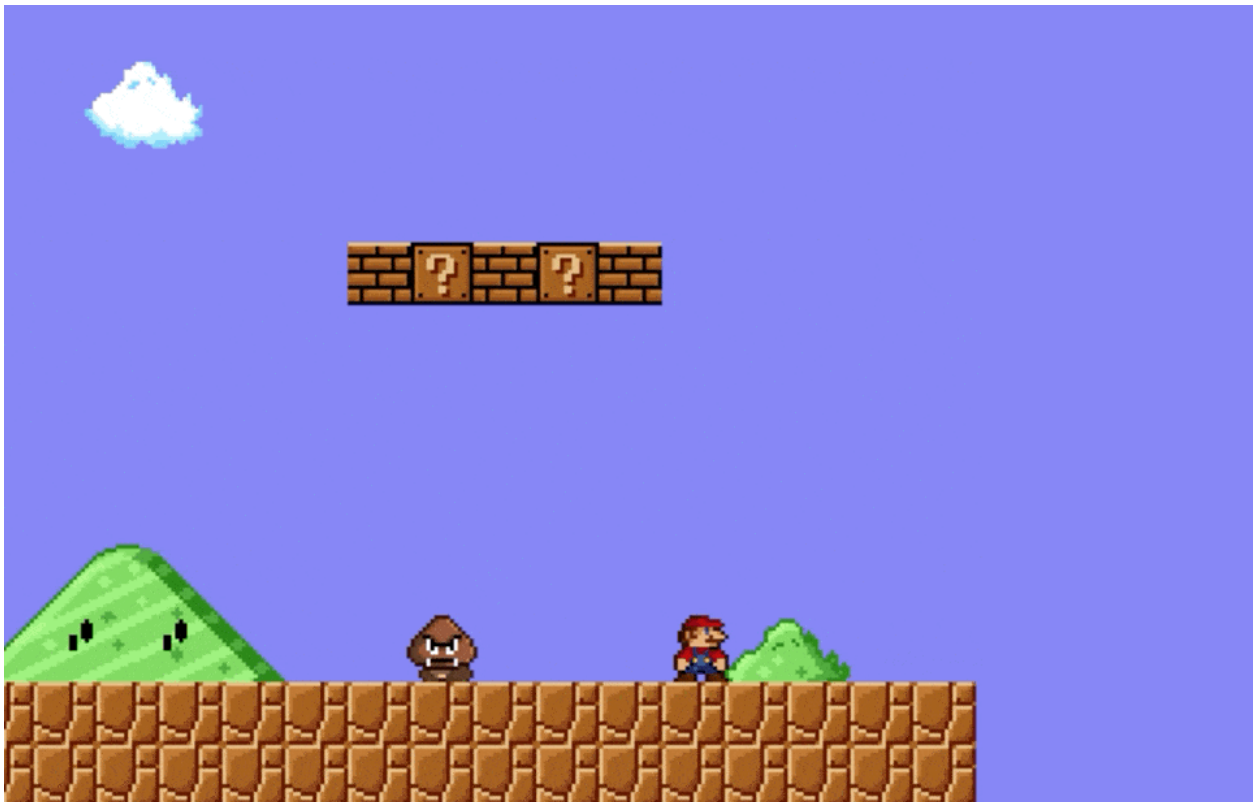
Take a look at the`onKeyDown` handlers `"right"` and `"left"`. We use the `flipX` method that is added to the character through the `sprite`[353] component. If this is `true`, it draws the sprite as a mirror image. This flip will make Mario face in the correct direction. We call the `move` method, which is added by the `pos`[354] component. Our `move` method takes in the number of pixels to move per second, which we set in the `SPEED` constant. You might want to move this constant definition nearer to the top of the file, so it's easier to find if you want to tweak it later.

In the `"left"` handler, there is also another check. In *Mario*, you can't walk back to previous parts of a level once it's gone off screen. We can simulate this by checking if Mario is near the left edge of the screen. We get Mario's current position by calling the `pos`[355] method which is added by the `pos` component. However, this position will be relative to the whole level, and not just the onscreen view. To help us figure out if Mario is near the edge of the screen, and not just at the beginning of the level, we can use the `toScreen`[356] function, which converts "game world" or level co-ordinates to actual screen co-ordinates.

When a player releases the space key, we want to make Mario jump. To do this, we can call the `jump`[357] method, which is added to the character through the `body`[358] component. However, `jump` will make the character shoot up, even if it is already in the air. To prevent this double jumping, we first check if the player is standing on some `solid`[359] object. The `body`[360] component also adds the `grounded()`[361] function, which returns `true` if the player is indeed standing on a solid object.

Press Command + S or Control + S to update the output, and test it out. Mario should move around, but it doesn't look very natural and *Mario*-like - yet! Another thing you'll notice is that the screen does not scroll when Mario walks to the right, so we can't get to the rest of the level. Let's fix that first.

[353] https://kaboomjs.com/#sprite
[354] https://kaboomjs.com/#pos
[355] https://kaboomjs.com/#pos
[356] https://kaboomjs.com/#toScreen
[357] https://kaboomjs.com/#body
[358] https://kaboomjs.com/#body
[359] https://kaboomjs.com/#solid
[360] https://kaboomjs.com/#body
[361] https://kaboomjs.com/#body



Mario moving

Click to open gif[362]

Adding scrolling

Kaboom has a number of functions to control the "camera" of the scene. The camera represents the field of view that the player can see. At the moment, the camera only shows the first part of the level. By using the `camPos`[363] function, we can move the camera to show more of the level as Mario walks across the scene.

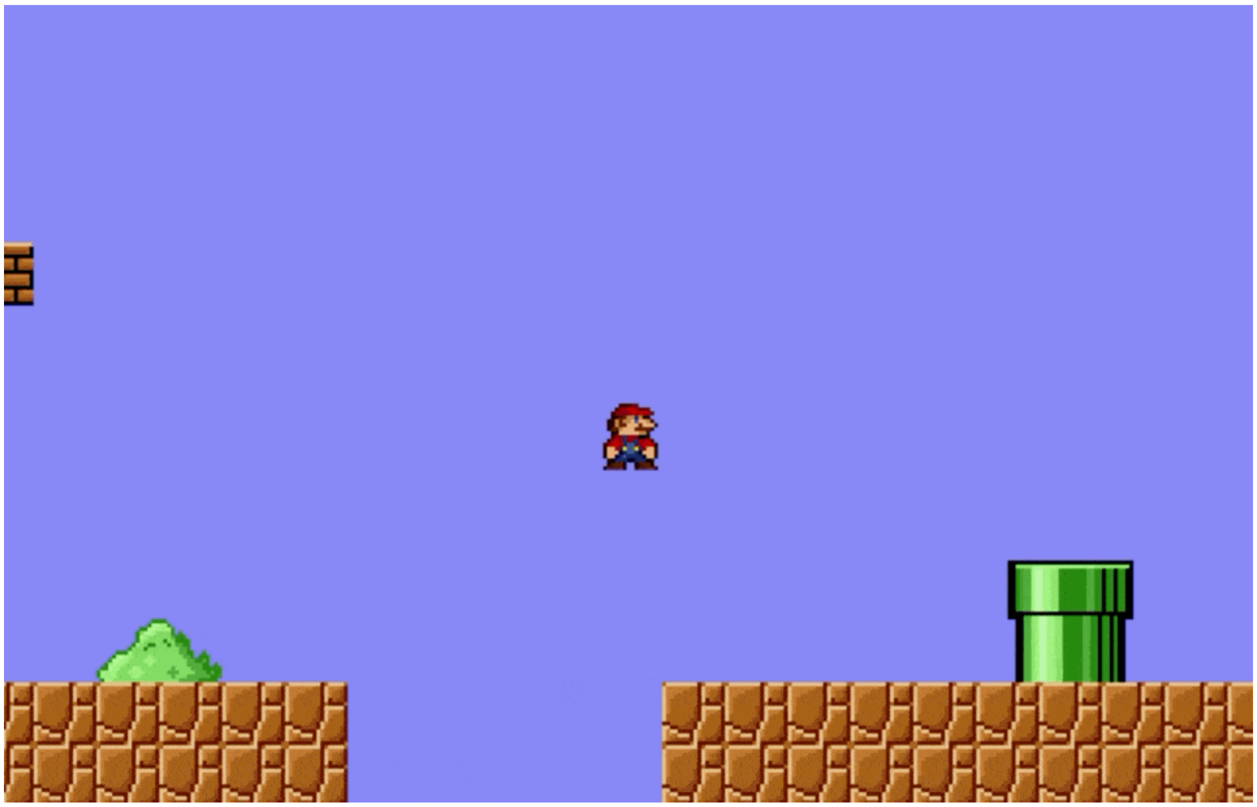
Let's add this code to the game scene:

[362]https://docs.replit.com/images/tutorials/32-mario-kaboom/mario-move.gif
[363]https://kaboomjs.com/#camPos

```
player.onUpdate(() => {
  // center camera to player
  var currCam = camPos();
  if (currCam.x < player.pos.x) {
    camPos(player.pos.x, currCam.y);
  }
});
```

Here we add a handler to the `onUpdate`[364] event for the player. This is called for each frame that is rendered. In this handler, we get the camera's current position by calling the `camPos`[365] function without any arguments. Then we can check if Mario is further to the right of the scene than the camera is. If he is, then we set the camera's X position to that of Mario, so essentially the camera is following Mario. We only do this if he is further to the right of the camera, and not for positions further to the left. This is because we don't want the player to be able to go back on a level.

Update the output again and test it out. As you move Mario past the center of the screen, the camera should start following him, giving the sense of scrolling.



Scrolling

Click to open gif[366]

[364]https://kaboomjs.com/#onUpdate
[365]https://kaboomjs.com/#camPos
[366]https://docs.replit.com/images/tutorials/32-mario-kaboom/scrolling.gif

Creating a custom component

To add more abilities to the characters and features to the game, we'll use components that we create ourselves and can add to our games objects. To define a new component in Kaboom, we make a function that returns an object with a few required fields and methods. Here's a template to illustrate code for a custom component (not part of the game code):

```
function customComponent(args) {
  return {
    id: "name",
    require: ["component1", "component2",],
    add() {
    },
    update() {
    },
  };
}
```

In the object we return, Kaboom requires an `id`, which is a unique name for the component. Kaboom also needs a `require` property, which is a list of other components this component needs in order to work. When a component is first initialized on a game object, Kaboom calls the `add()` method so we have the opportunity to run any setup code we need. The method `update()` is called on every game frame, so we can make animation and collision updates there.

One behavior we need is for the enemy characters to walk up and down, instead of just standing in one place. Let's make a custom component we can add to our enemy characters so that they automatically move back and forth, or patrol their part of the level. Add the code below to the bottom of `main.js`:

```
function patrol(distance = 100, speed = 50, dir = 1) {
  return {
    id: "patrol",
    require: ["pos", "area",],
    startingPos: vec2(0, 0),
    add() {
      this.startingPos = this.pos;
      this.on("collide", (obj, side) => {
        if (side === "left" || side === "right") {
          dir = -dir;
        }
      });
    },
    update() {
      if (Math.abs(this.pos.x - this.startingPos.x) >= distance) {
        dir = -dir;
      }
      this.move(speed * dir, 0);
    },
  };
}
```

We've called this custom component "patrol", as it will make a character move some `distance`, at a set `speed`, and then turn around and walk in the opposite direction, `dir`. The character will also turn and move in the opposite direction if it collides with another game object. Because we use the `move` method (which is part of the `pos`[367] component), and the `collide` event handler (which is part of the `area`[368] component), we add the `pos` and `area` components to the `require` list.

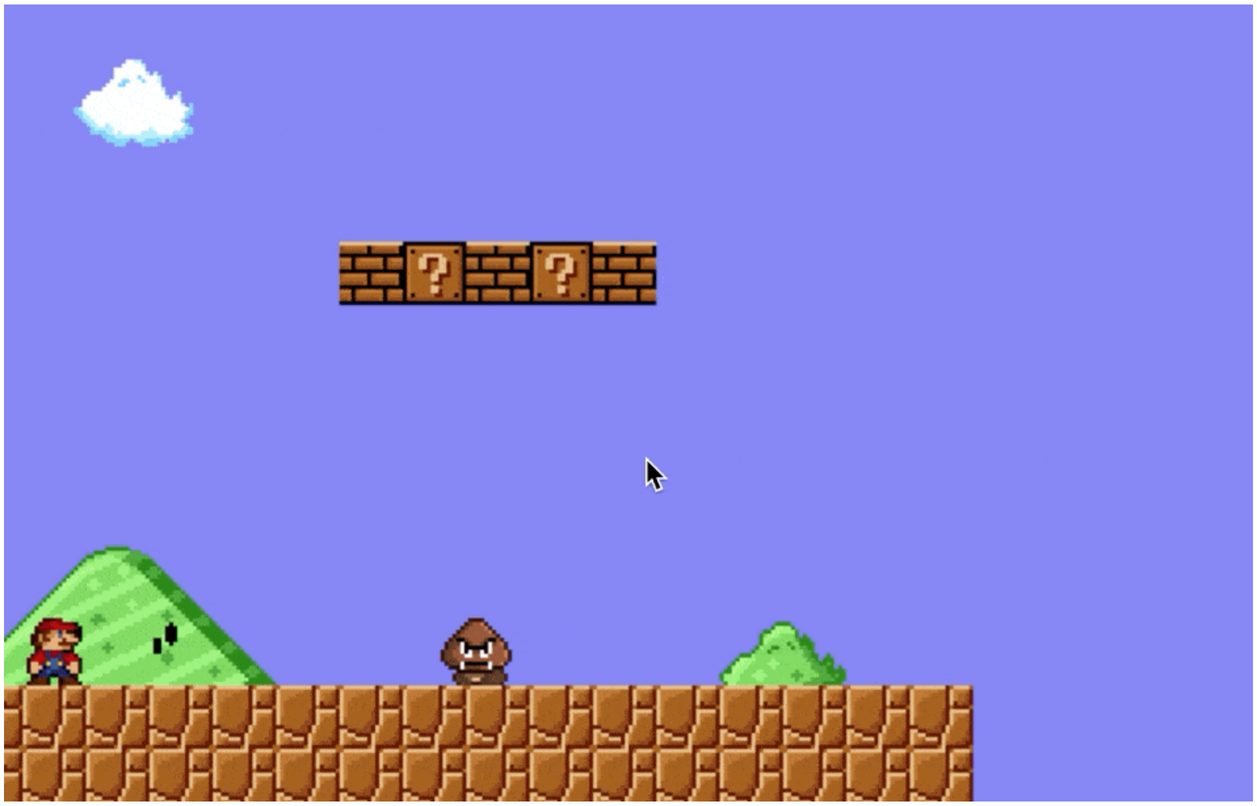
When the component is first initialized, we want to record its starting position. This is so that we know how far the character is from where it started off, and therefore we'll know when we must turn it to move in the opposite direction. We do this by making use of the `add()` method. As we mentioned above, Kaboom will call this method on our component when the character is added to the scene. We read the position of the character at that time by calling `this.pos`: `this` refers to our character (as our component is made part of the character, `this` is reference to the combination of all the components making up the character). We can save this initial position to a property of the component object, in our case one called `startingPos`. We then attach a `collide`[369] handler, so we know if the character bumps into anything, so we can turn it and move it in the opposite direction again.

The `collide`[370] handler has 2 arguments passed to it: the `obj` that our character collided with, and the `side` of character that was hit. We only want to flip the direction our character moves in if it was hit from the sides `left` and `right`. To change the direction, we flip the sign of our `dir` variable, which we'll use in the `update()` method.

The `update()` method is called for each frame. In it, we first check if the character is further than the specified maximum `distance` from its starting position. If it is, we switch the sign of the `dir` variable to make the character move in the opposite direction. Note that we we use the `Math.abs`[371] function, which returns the absolute value of a number. The absolute value of a number is always positive, and this allows us to compare it to our `distance` variable, which is also a positive value. Then we call the `move` method (provided by the `pos`[372] component) and use the `speed` variable (passed in when the component was created) along with the `dir` variable to specify how fast and in which direction to move the character for this frame.

[367]https://kaboomjs.com/#pos
[368]https://kaboomjs.com/#area
[369]https://kaboomjs.com/#area
[370]https://kaboomjs.com/#on
[371]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/abs
[372]https://kaboomjs.com/#pos

Now that we've created this component, let's use it on a character. Uncomment the lines `//patrol` wherever you see it in the `levelConf` setup we created. Update the output and test it out. You should see the enemy character walk back and forth.



Patrol component

Click to open gif[373]

You'll notice we also make use of the `patrol` component on the `bigMushy` character, which we'll use to make Mario grow from small Mario to big Mario. We'll get to that in a bit.

Creating a custom component for the enemies

Now that the enemies are moving around, we can give them some more behaviors and properties. One of the most important things to do is to squash the enemies if Mario jumps on them. If you take a look at the `enemies.png` sprite file, you'll see that the 3rd frame (index 2) is an image of the enemy, but squashed.

[373] https://docs.replit.com/images/tutorials/32-mario-kaboom/patrol.gif

Enemies-index

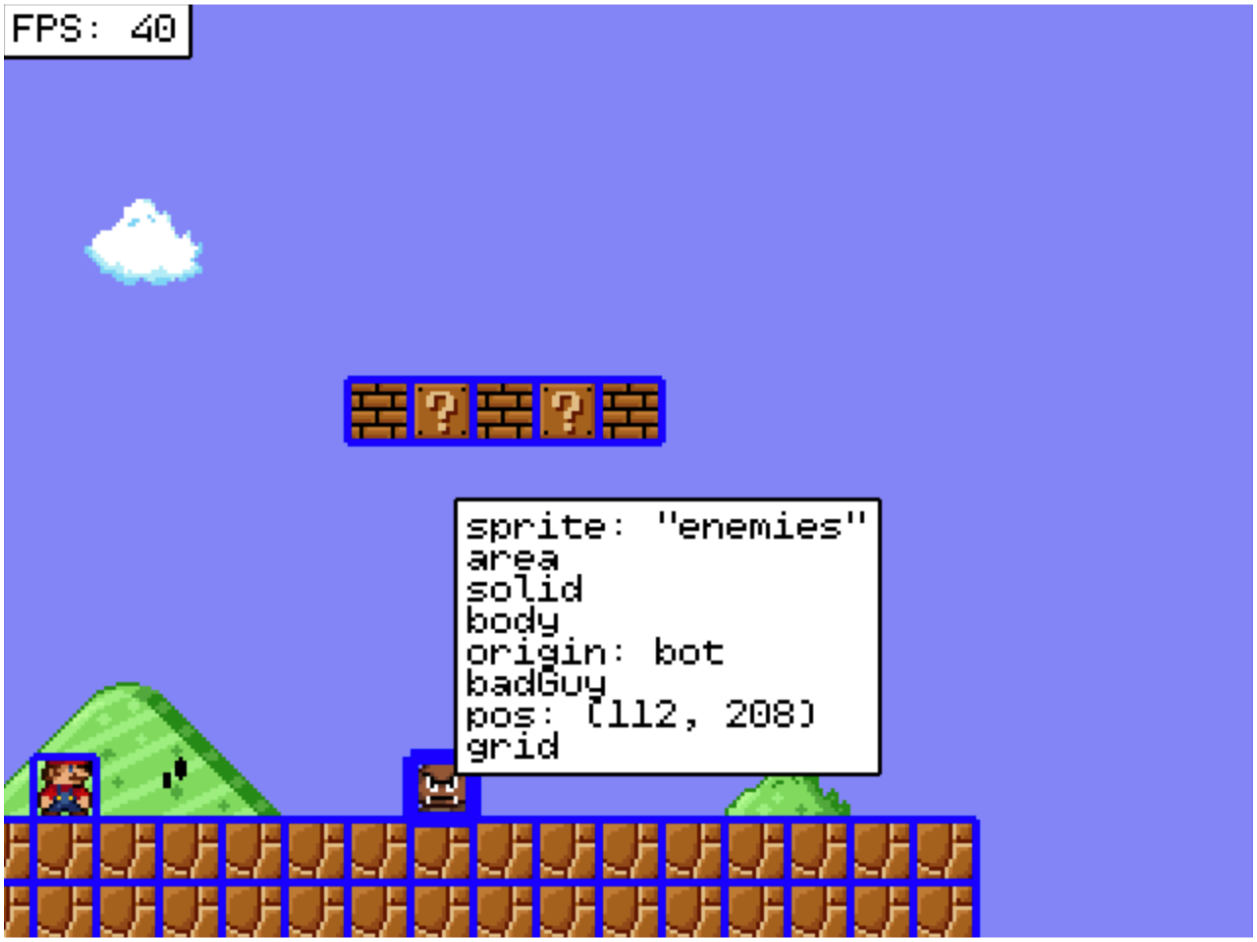
We can swap out the animation that is played when the enemy is patrolling for this frame. Kaboom has a built in `lifespan`[374] component that also has a fade out function. This component allows us to slowly fade out the squashed enemy from the scene, and then automatically remove it entirely once the specified lifetime is reached. Kaboom also allows us to dynamically add and remove components from characters using the `use` and `unuse` methods. These methods are not yet documented, but you can find them (and more tricks!) by looking at the Kaboom source code[375]. Let's use this knowledge to build a custom component to handle the enemy getting squashed and fading out of the scene:

```
function enemy() {
  return {
    id: "enemy",
    require: ["pos", "area", "sprite", "patrol"],
    isAlive: true,
    update() {

    },
    squash() {
      this.isAlive = false;
      this.unuse("patrol");
      this.stop();
      this.frame = 2;
      this.area.width = 16;
      this.area.height = 8;
      this.use(lifespan(0.5, { fade: 0.1 }));
    }
  }
}
```

We define the custom component as we did before. Because we need to stop the enemy from patrolling, we require the `patrol` custom component. We also require the `sprite`[376] component so we can stop the animation and set the squashed frame to display. The `area`[377] component is necessary, as the squashed enemy frame is half the height of the regular enemy frames (8 pixels vs 16 pixels). We're going to need to adjust the height of the area so that the collision zone the enemy occupies once squashed is correct.

[374]https://kaboomjs.com/#lifespan
[375]https://github.com/replit/kaboom
[376]https://kaboomjs.com/#sprite
[377]https://kaboomjs.com/#area

As a side note, pressing F1 in the game turns on Kaboom debugging, which will draw the `area`[378] box around each game element, so you can easily see when characters collide. It also shows other handy info, like the frame rate and character properties.



```
FPS: 40
```

```
sprite: "enemies"
area
solid
body
origin: bot
badGuy
pos: [112, 208]
grid
```

Debug mode

Back to our code. We execute our enemy squash in the `squash` method. We have a flag called `isAlive`, which we'll use to determine if the enemy is able to hurt Mario. This is usually `true`, but set to `false` once the enemy is squashed and harmless. We also `unuse` the patrol component so that the enemy stops walking back and forth. Then we call `stop`, which is a method added by the `sprite`[379] component. Calling `stop` stops playing the current animation. Then we set the `frame` of the sprite to use to `2`, which is the squashed enemy frame, and update the `area`[380] width and height to be the same size as the frame. Finally, we call `use` to add the `lifespan`[381] component so that the character is removed from the scene after `0.5` seconds, and fades out for `0.1` seconds.

Now let's add this custom component to the enemy. First, uncomment the `//enemy(),` line in the

[378] https://kaboomjs.com/#area
[379] https://kaboomjs.com/#sprite
[380] https://kaboomjs.com/#area
[381] https://kaboomjs.com/#lifespan

`levelConf` setup we created earlier. Now create a collision handler in the game scene between Mario and the enemy, so we know when it gets squashed:

```
let canSquash = false;

player.onCollide("badGuy", (baddy) => {
  if (baddy.isAlive == false) return;
  if (canSquash) {
    // Mario has jumped on the bad guy:
    baddy.squash();
  } else {
    // Mario has been hurt. Add logic here later...
  }
});
```

In this `onCollide` handler, we check if the player, Mario, collides with a `badGuy` - which is the tag we gave to the enemies in the `levelConf` setup above. Then we attach our handler, which takes the `baddy` character Mario collided with. We first check if the `baddy` is still alive - if not we leave early, as there is no real interaction between Mario and a dead enemy. Then, we check the `canSquash` variable - if it is set to `true`, that means Mario has jumped onto the enemy. In this case, we call the `squash` method, which we created in the custom component for the enemy. This will execute all the logic we added there, and "kill" the enemy. We leave a bit of room in the handler to come back and add logic if Mario collides with the enemy without jumping on it - we'll add in that logic later.

Modify the `onKeyPress` handler for the `space` key as follows:

```
  onKeyPress("space", () => {
    if (player.grounded()) {
      player.jump();
      canSquash = true;
    }
  });
```

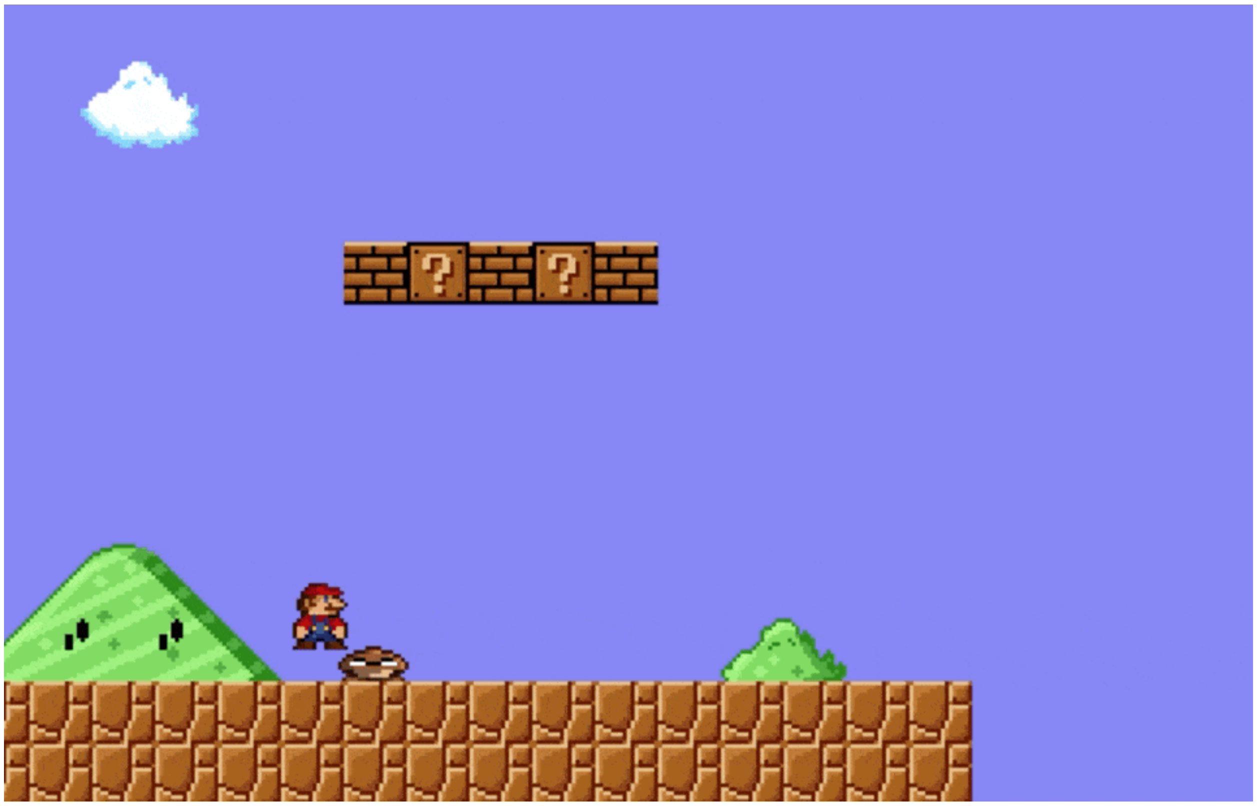
Here, we set the `canSquash` variable to `true` to allow the player to squash the enemy if the player has jumped over it upon collision.

Add the following code to the `player.onUpdate` handler:

```
if (player.grounded()) {
  canSquash = false;
}
```

This code will reset the `canSquash` variable so that the player will not squash the enemy if it hasn't jumped over it in the collision handler we added earlier.

Update the output and test our game out. If you jump on an enemy, it should be squashed and then disappear after half a second.



Squash enemy

Click to open gif[382]

Headbutting surprise boxes

Another key *Mario* action is headbutting the surprise boxes (the ones with "?" on them). In *Mario World*, this could release a coin, a super mushroom (one that makes Mario grow bigger), etc. When Mario headbutts these boxes, the box is 'bumped' and moves up and down quickly, while releasing its surprise. Once the box is empty, the "?" is removed from it. Let's create the logic to control these boxes. As above, we'll make more use of custom components.

We'll take care of the boxes for coins and for the grow-bigger mushrooms. If you take a look in the `levelConf` setup we added in the beginning, you'll see entries for coin and mushroom "question boxes". The only real difference between the two is the final tag, which marks which surprise the box should release. We also have definitions for an empty box (`!`), the coin (`c`), and the mushroom (`M`).

[382] https://docs.replit.com/images/tutorials/32-mario-kaboom/squash.gif

Let's create a component that makes the box jump up and fall back down when it is headbutted. We can also re-use it on the coin to make it flip as it's bumped out of the box. We'll call this new component `bump`:

```
function bump(offset = 8, speed = 2, stopAtOrigin = true) {
  return {
    id: "bump",
    require: ["pos"],
    bumpOffset: offset,
    speed: speed,
    bumped: false,
    origPos: 0,
    direction: -1,
    update() {
      if (this.bumped) {
        this.pos.y = this.pos.y + this.direction * this.speed;
        if (this.pos.y < this.origPos - this.bumpOffset) {
          this.direction = 1;
        }
        if (stopAtOrigin && this.pos.y >= this.origPos) {
          this.bumped = false;
          this.pos.y = this.origPos;
          this.direction = -1;
        }
      }
    },
    bump() {
      this.bumped = true;
      this.origPos = this.pos.y;
    }
  };
}
```

This looks a bit more complicated than our other custom components, but that's only because it has code for the object moving in both directions. First off, we have a few parameters when creating this component:

- `offset` is how far up we want the object to be bumped before settling down again.
- `speed` is how fast we want it to move when bumped.
- `stopAtOrigin` specifies whether we want the object to return to its original position after being bumped, or just keep falling down - if this parameter is `false`, then bumping the object will make it look like it got dislodged and it will fall down.

The object this component is added to must also have the `pos` component. We'll use that to move the object when it is bumped.

We add a method, `bump`, which we can call from a collision handler or elsewhere. This sets the property `bumped` to `true`. This is a flag in the `update` method that will trigger the bump behavior. In the `bump` method, we also record the `y` position of the object in a property `origPos` so that we can stop the object at its original position if the `stopAtOrigin` flag has been set.

In the `update` method, which is run on each frame, we first check if the `bumped` flag has been set. If it has, we update the `y` position of the object using the `speed` we set, and in the current `direction`. On a screen, the top of the screen is where y = `0`, and the bottom is the max height of the screen. Therefore to start, `direction` is set to `-1` to move the object up. Then we have a check to see if the new position of the object is higher than the `offset` parameter distance from the object's original position. If it is, we reverse the direction, so the object now starts moving back down.

Next, we have a check to see if the flag `stopAtOrigin` is set. If the object has fallen down to its original position (or further), we set the `bumped` flag back to `false` and update the object's position exactly back to its original position. We also set the `direction` flag back to `-1`, so the object is back in a state that it can be bumped again.

Now that this `bump` component exists, you can uncomment the `//bump(),` lines in the `levelConf` setup we created earlier.

To trigger the bump and add the code to make the surprise come out, we'll need to add a collision handler. Believe it or not, Kaboom has a special case collision event called `headbutt` (which is not documented, but you can also find it in the source code[383]) just for this type of thing!

```
player.on("headbutt", (obj) => {
  if (obj.is("questionBox")) {
    if (obj.is("coinBox")) {
      let coin = level.spawn("c", obj.gridPos.sub(0, 1));
      coin.bump();
    } else
    if (obj.is("mushyBox")) {
      level.spawn("M", obj.gridPos.sub(0, 1));
    }
    var pos = obj.gridPos;
    destroy(obj);
    var box = level.spawn("!", pos);
    box.bump();
  }
});
```

In the handler for `headbutt`, we are passed the object, `obj`, that the player headbutted. We check to

[383]https://github.com/replit/kaboom

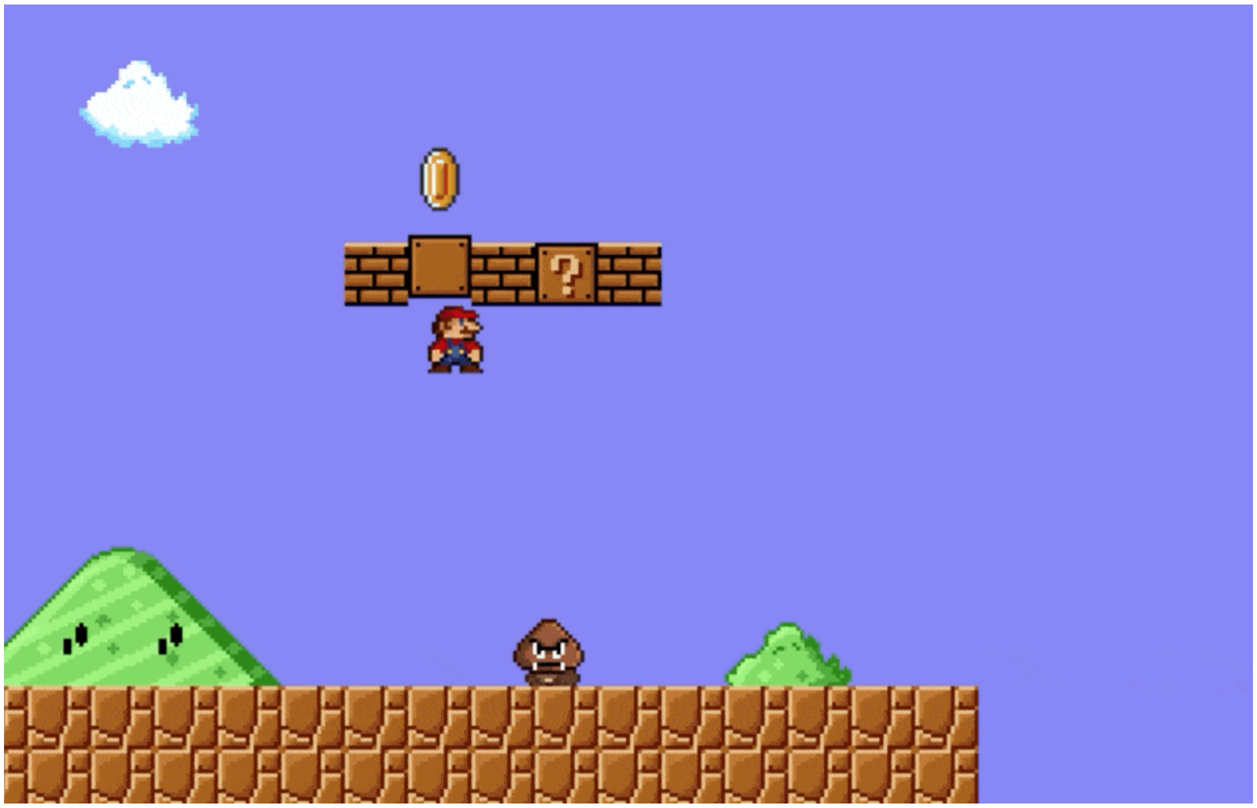
see if it is indeed one of our `questionBox` objects. If it is, we check if it is a `coinBox` (a coin must pop out) or a `mushyBox` (a grow-bigger mushroom should pop out).

If it is a `coinBox`, we `spawn` a new coin 1 block above the coin box, using the configuration `c` we setup for a coin in the `levelConf` in the beginning. Then we call `bump` on the coin to invoke our custom component's method to make it appear to flip up out of the box.

If it is a `mushyBox`, we do the same, except we don't bump the mushroom. The mushroom has our custom `patrol` component added to it (check in the `levelConf` for `M`), so it will start moving immediately. We set the patrol distance very large on the mushroom so it won't automatically turn around, it will just keep going until if falls of the screen.

Then, to replace the `questionBox` with an empty box, we first record its position, then `destroy`[384] it, and `spawn` a new empty box (`!` in the `levelConf`) to take its place. Finally, we `bump` this new box to give it the motion we want.

Cool, time to update the output and test this out. When you jump up using the `space` key and headbutt the question boxes now, they should move and have things pop out!



Question-boxes

Click to open gif[385]

[384] https://kaboomjs.com/#destroy
[385] https://docs.replit.com/images/tutorials/32-mario-kaboom/question-box.gif

Adding special behaviors to Mario

Now we've got the basics of all the other game elements down, it's time to create a custom component for Mario himself. This component will need to do quite a bit, as Mario is the main character. Here are the things it will need to handle:

- Make Mario get bigger or smaller.
- Run the "running" animation when Mario is running, and change to a standing or jumping frame in other cases.
- "Freeze" Mario when he gets to the castle or has died, so the player can no longer move him.
- Handle Mario dying, with a classic spring up and out of the scene.

Ok, here it is, our Mario custom component:

```
function mario() {
    return {
      id: "mario",
      require: ["body", "area", "sprite", "bump"],
      smallAnimation: "Running",
      bigAnimation: "RunningBig",
      smallStopFrame: 0,
      bigStopFrame: 8,
      smallJumpFrame: 5,
      bigJumpFrame: 13,
      isBig: false,
      isFrozen: false,
      isAlive: true,
      update() {
        if (this.isFrozen) {
          this.standing();
          return;
        }

        if (!this.grounded()) {
          this.jumping();
        }
        else {
          if (keyIsDown("left") || keyIsDown("right")) {
            this.running();
          } else {
            this.standing();
          }
        }
      },
      bigger() {
        this.isBig = true;
        this.area.width = 24;
        this.area.height = 32;
      },
      smaller() {
        this.isBig = false;
        this.area.width = 16;
        this.area.height = 16;
      },
      standing() {
        this.stop();
        this.frame = this.isBig ? this.bigStopFrame : this.smallStopFrame;
      },
      jumping() {
        this.stop();
        this.frame = this.isBig ? this.bigJumpFrame : this.smallJumpFrame;
      },
      running() {
        const animation = this.isBig ? this.bigAnimation : this.smallAnimation;
        if (this.curAnim() !== animation) {
          this.play(animation);
        }
      },
      freeze() {
        this.isFrozen = true;
      },
      die() {
        this.unuse("body");
        this.bump();
        this.isAlive = false;
        this.freeze();
        this.use(lifespan(1, { fade: 1 }));
      }
    }
  }
```

Firstly, we require the character to have a few other components: `body`[386], so we can determine if Mario is jumping or on the ground; `area`[387], so we can change the collision box area of Mario as

[386]https://kaboomjs.com/#body
[387]https://kaboomjs.com/#area

he grows or shrinks; `sprite`[388], so we can start and stop animations and set static frames; and our custom `bump` component, so we can throw Mario off the screen if he dies.

If we take a peek in the `Mario.json` file along with the `Mario.png` sprite file, we'll see that there are some animations defined in the `frameTags` section that we can use. The `Running` animation contains all the frames to make Mario appear to be running when he is in small size. Similarly, `RunningBig` has all the frames for when Mario is running while he is in big size. We can also see that a good frame for small Mario standing still, or stopped, is the first frame, or frame `0`. A good frame to use for big Mario standing still or stopped is frame `8`. Good frames for Mario jumping when he is small and big are `5` and `13` respectively. So we don't have to keep remembering all these magic strings and numbers, we set them as properties of the Mario component.

If we measure the size of the big Mario images, we'll see that the tightest crop we can get on them is about `24x32` pixels. For small Mario, the size is `16x16` pixels. We'll use this knowledge to set the correct Mario animation and `area`[389] collision boxes when changing between animations and sizes.

In the `mario` component, we define a number of custom methods. Let's go through them.

- The `bigger` and `smaller` methods provide a way to change the size of Mario. We set a flag `isBig` that we check in the other methods to choose appropriate animations and frames. We also set the collision `area`[390] size appropriate for the size of Mario.
- The `standing` and `jumping` methods are called from our main `update` method, which is called with each frame. In these `2` methods, we stop any animation that is currently running using the `stop` method provided by the `sprite`[391] component. Then, depending on the size of Mario, determined by the `isBig` flag, we set the appropriate static `frame` to make Mario look like he is standing still or jumping.
- In the `running` method, we find the correct running animation depending on whether Mario is big or small. Then we check if that animation is the same animation that Mario is currently using, by calling the `curAnim` method provided by the `sprite`[392] component. If they are not the same, we update the current animation by calling `play` to start the new animation. We first check the current animation because, if we set the animation regardless of what is currently playing, we'd keep resetting the current animation to the beginning with each frame and make it appear as a static frame.
- The `freeze` method sets a flag `isFrozen`, which is used in the `update` method to determine whether Mario can move.
- When Mario is killed, we can call the `die` method. This first removes the `body`[393] component on Mario so that he is no longer subject to gravity or collisions, because these are things that ghosts are not worried about. Then we call the `bump` method that is added by our custom `bump` component. This shoots Mario up into the air, and back down again. We also set the `isAlive` flag to false, to signal to any collision handlers that Mario is dead before they try kill him again,

[388]https://kaboomjs.com/#sprite
[389]https://kaboomjs.com/#area
[390]https://kaboomjs.com/#area
[391]https://kaboomjs.com/#sprite
[392]https://kaboomjs.com/#sprite
[393]https://kaboomjs.com/#body

or give him a 1-up mushroom or coin. We `freeze()` Mario so that he reverts to a standing pose and keyboard input doesn't affect him, and finally, we `use` the `lifespan`[394] component to fade Mario out and remove him from the scene.

We also have the required `update` method, which Kaboom calls every frame. In this method, we check if Mario is frozen, in which case we call our `standing` method to update Mario's pose. Then we check if Mario is not on the ground, in other words he is in the air, using the `grounded()` method provided by the `body`[395] component. This method returns `true` if he is on a `solid` object, or `false` otherwise. If this comes back `false`, we call the `jumping` method to stop any animations and set the static jumping frame.

If Mario is not jumping, i.e. he is `grounded`, then we check if the user currently has the `left` or `right` key down. If so, this means Mario is `running()`. The final condition is if Mario is on the ground but is not moving, then he must be `standing()`.

Nice! Uncomment the `mario` component line in the `levelConf` to activate this new component on Mario. Update the output and test the animation changes out. Instead of Mario in a single pose, you should see an animation as he runs, changing to a static frame as he stands still or jumps.

Now we can hook up collision handlers to check if Mario has eaten the mushroom to grow larger, or if he gets injured or killed by an enemy or falling off the screen.

Adding more Mario collisions and events

First, let's add a collision handler between the mushroom and Mario. Then we can call our `bigger` method from our custom `mario` component to grow him.

```
player.onCollide("bigMushy", (mushy) => {
  destroy(mushy);
  player.bigger();
});
```

In this handler, we remove the mushroom from the scene, and then make Mario `bigger()`.

Let's add some more code to the handler we created earlier for Mario colliding with an enemy. There, we only handled the case of Mario jumping on the enemy. We still need to account for Mario being injured or killed by the enemy. Update the `badGuy` collision handler like this:

[394] https://kaboomjs.com/#lifespan
[395] https://kaboomjs.com/#body

```
player.onCollide("badGuy", (baddy) => {
  if (baddy.isAlive == false) return;
  if (player.isAlive == false) return;
  if (canSquash) {
    // Mario has jumped on the bad guy:
    baddy.squash();
  } else {
    // Mario has been hurt
    if (player.isBig) {
      player.smaller();
    } else {
      // Mario is dead :(
      killed();
    }
  }
});
```

We add in a condition at the top of the handler to exit early if Mario is not alive.

In the `else` condition, if Mario did not squash the bad guy, we check if we are dealing with big Mario. If so, *Mario World* rules are that he doesn't die, he just gets `smaller()`. However, if Mario is already small, he unfortunately is now dead. We call out to another function `killed` to handle Mario's death scene. Let's add that function:

```
function killed() {
  // Mario is dead :(
  if (player.isAlive == false) return; // Don't run it if mario is already dead
  player.die();
  add([
    text("Game Over :(", { size: 24 }),
    pos(toWorld(vec2(160, 120))),
    color(255, 255, 255),
    origin("center"),
    layer('ui'),
  ]);
  wait(2, () => {
    go("start");
  })
}
```

This function does a few things. First, we run the code in the `die` method from the custom `mario` component. Then we `add`[396] some `text`[397] to notify the player that the game is over. We `wait`[398] for 2

[396]https://kaboomjs.com/#add
[397]https://kaboomjs.com/#text
[398]https://kaboomjs.com/#wait

seconds as we pay Mario our final respects, and then we `go`[399] back to the start scene to play again.

Another way Mario can die in *Mario World* is if he falls off the platform into the void. We can check for this by modifying the `player.onUpdate` handler we added earlier for moving the camera:

```
player.onUpdate(() => {
  // center camera to player
  var currCam = camPos();
  if (currCam.x < player.pos.x) {
    camPos(player.pos.x, currCam.y);
  }

  if (player.isAlive && player.grounded()) {
    canSquash = false;
  }

  // Check if Mario has fallen off the screen
  if (player.pos.y > height() - 16){
    killed();
  }

});
```

Here, we check if Mario's `y` co-ordinate is greater than the `height`[400] of the Kaboom window, less the size of one platform block, which is 16 pixels. If this is the case, it means Mario has fallen off the top row of the platform, and therefore has been `killed()`.

We also need to update our `left` and `right` key handler events to check if Mario `isFrozen`. In this case the handlers should just return early without moving Mario:

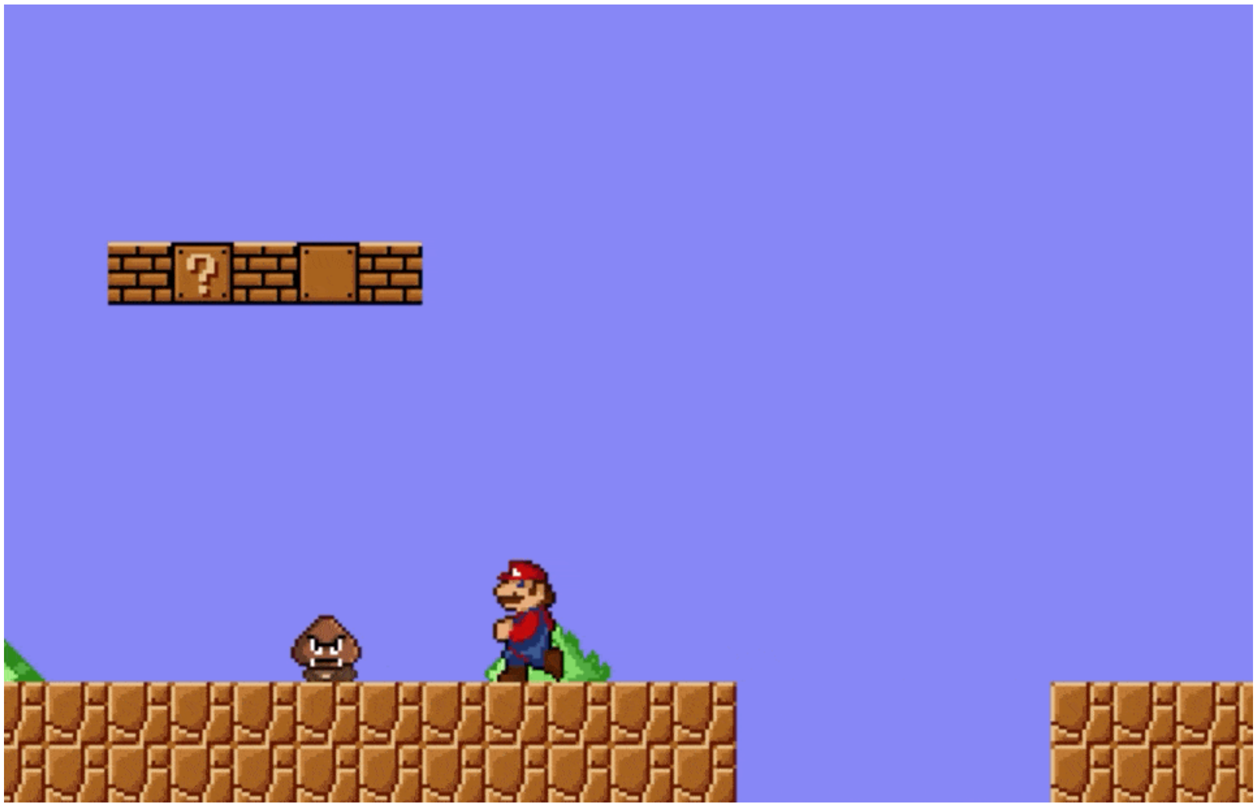
```
    onKeyDown("right", () => {
      if (player.isFrozen) return;
      player.flipX(false);
      player.move(SPEED, 0);
    });

    onKeyDown("left", () => {
      if (player.isFrozen) return;
      player.flipX(true);
      if (toScreen(player.pos).x > 20) {
        player.move(-SPEED, 0);
      }
    });
```

[399]https://kaboomjs.com/#go
[400]https://kaboomjs.com/#height

Then we also modify the `space` key handler to only jump if Mario is still alive:

```
onKeyPress("space", () => {
    if (player.isAlive && player.grounded()) {
      player.jump();
      canSquash = true;
    }
});
```

Time to update the output and test all these changes out! First thing to test is if Mario grows bigger by eating the mushroom. Second thing to check is if Mario then gets smaller again by colliding with an enemy. Also check if Mario is killed when colliding with an enemy when he is small, or when falling off the platform.



Mario bigger and killed

Click to open gif[401]

[401]https://docs.replit.com/images/tutorials/32-mario-kaboom/bigger-kill-scenes.gif

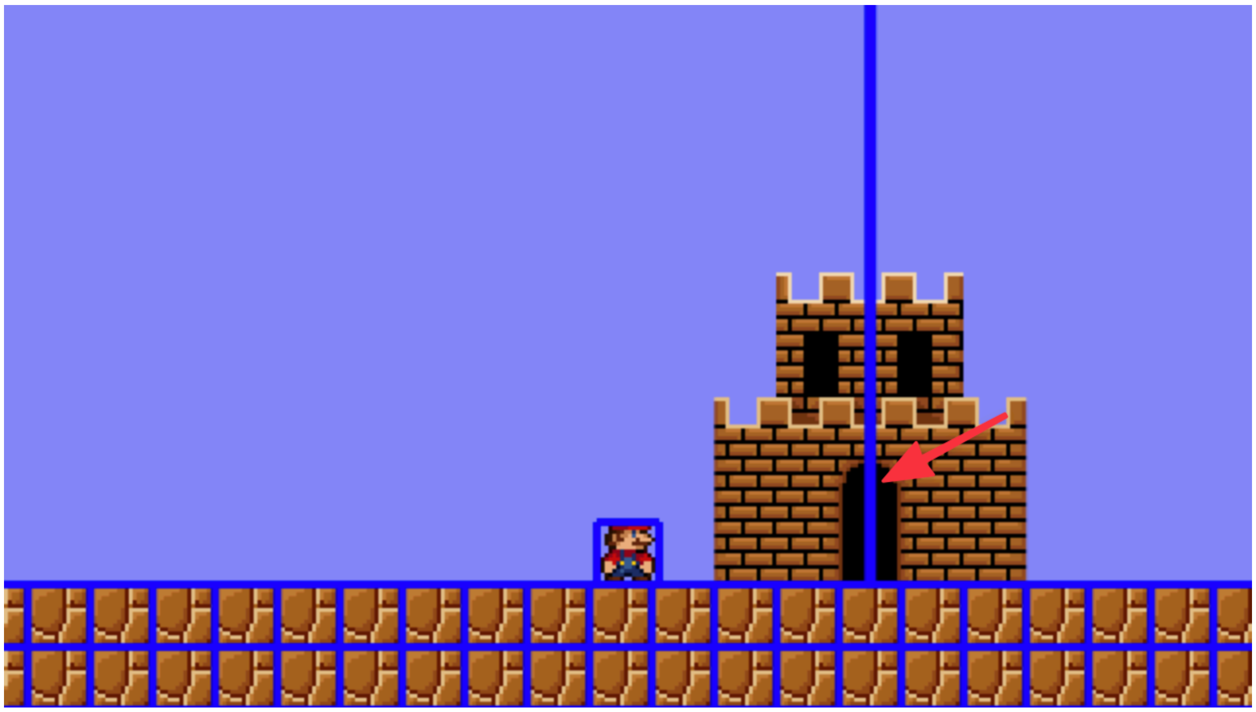
Ending when we get to the castle

The final thing to do for this tutorial is to handle the case when Mario reaches the castle. At this point, when Mario is at the door, we want him to `freeze`, a congratulations message to appear to the player, and then move on to the next level.

We can use a regular collision handler to check if Mario is at the castle. Notice in the setup for the castle in the `levelConf` we added earlier, we set the collision `area`[402] of the castle to a very narrow, completely vertical line of width 1 pixel and height 240, which is the screen height:

```
  "H": () => [
    sprite("castle"),
    area({ width: 1, height: 240 }),
    origin("bot"),
    "castle"
  ],
```

This is so that the collision between Mario and the castle is only registered when Mario gets to the center of the castle, where the door is. We can visualize this by pressing F1 in the game to enable the debugger and look at the area box at the castle:



Castle area

[402]https://kaboomjs.com/#area

The reason we make the area box the height of the screen is make sure the player can't accidentally jump over the ending point and fall off the end of the level.

Now let's add our collision handler for the castle to the game scene:

```
player.onCollide("castle", (castle, side) => {
  player.freeze();
  add([
    text("Well Done!", { size: 24 }),
    pos(toWorld(vec2(160, 120))),
    color(255, 255, 255),
    origin("center"),
    layer('ui'),
  ]);
  wait(1, () => {
    let nextLevel = levelNumber + 1;

    if (nextLevel >= LEVELS.length) {
      go("start");
    } else {
      go("game", nextLevel);
    }
  })
});
```

First, we `freeze` Mario so the player can't control him anymore. Then we `add`[403] our "Well Done!" `text`[404] message to the center of the screen. We `wait`[405] a second before incrementing our level number and going to the next level, or going back to the start of the game if we have completed all levels.

Next steps

There are few things left to do to complete the game:

- Add in some scoring. You can check out a previous Kaboom tutorial, like Space Shooter[406], to see how scoring works.
- Add in sounds and music. If you get your own copy of the *Mario* soundtrack and effects, you can use the `play`[407] sound function in Kaboom to get those classic tunes blasting as you play.

[403]https://kaboomjs.com/#add
[404]https://kaboomjs.com/#text
[405]https://kaboomjs.com/#wait
[406]https://docs.replit.com/tutorials/24-build-space-shooter-with-kaboom
[407]https://kaboomjs.com/#play

- Add in some more levels. This is the really fun part, where you get to create *Mario* levels you wish existed.
- You can also add in some more of the *Mario World* game characters.

Credits

https://twitter.com/Arrow_N_TheKnee[408] for the Mario Visual Assets[409]

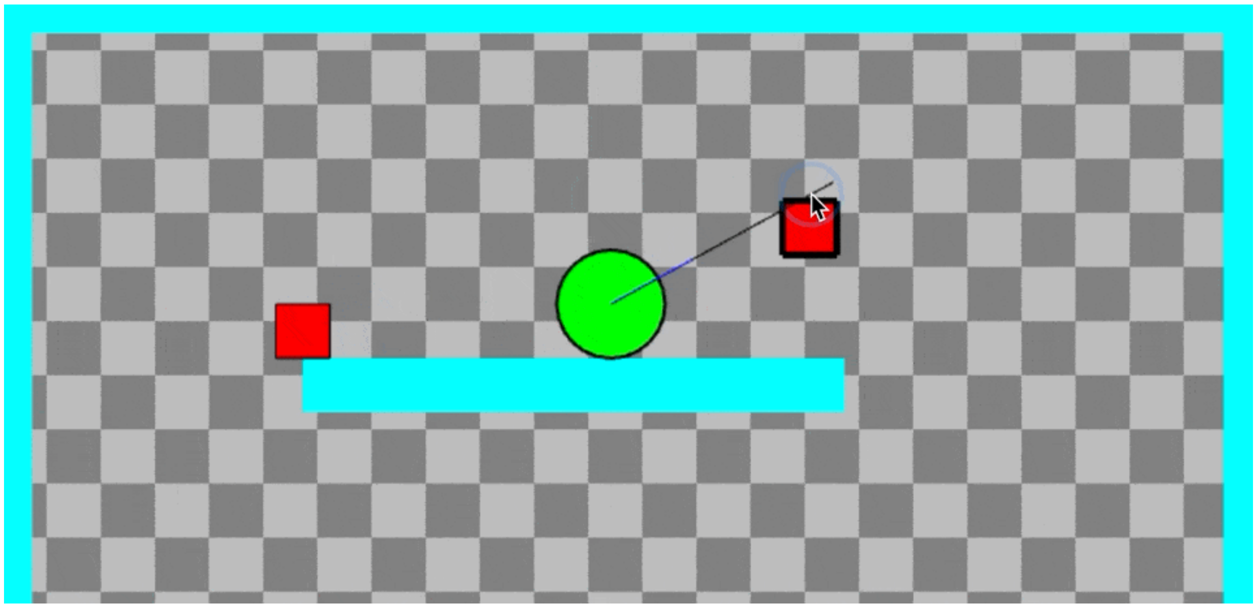
Code

You can find the code for this tutorial on Replit[410]

[408]https://twitter.com/Arrow_N_TheKnee
[409]https://dotstudio.itch.io/super-mario-1-remade-assets
[410]https://replit.com/@ritza/Mario

Build a Physics playground with Kaboom.js

In this tutorial, we will be building a simple physics playground with Kaboom.js[411]. This will be a 2D side-view platformer in which the player can use a gravity gun[412] to pick up and shoot objects in the world, similar to games like *Half-Life 2* and *Rochard*.



The finished game

Click to open gif[413]

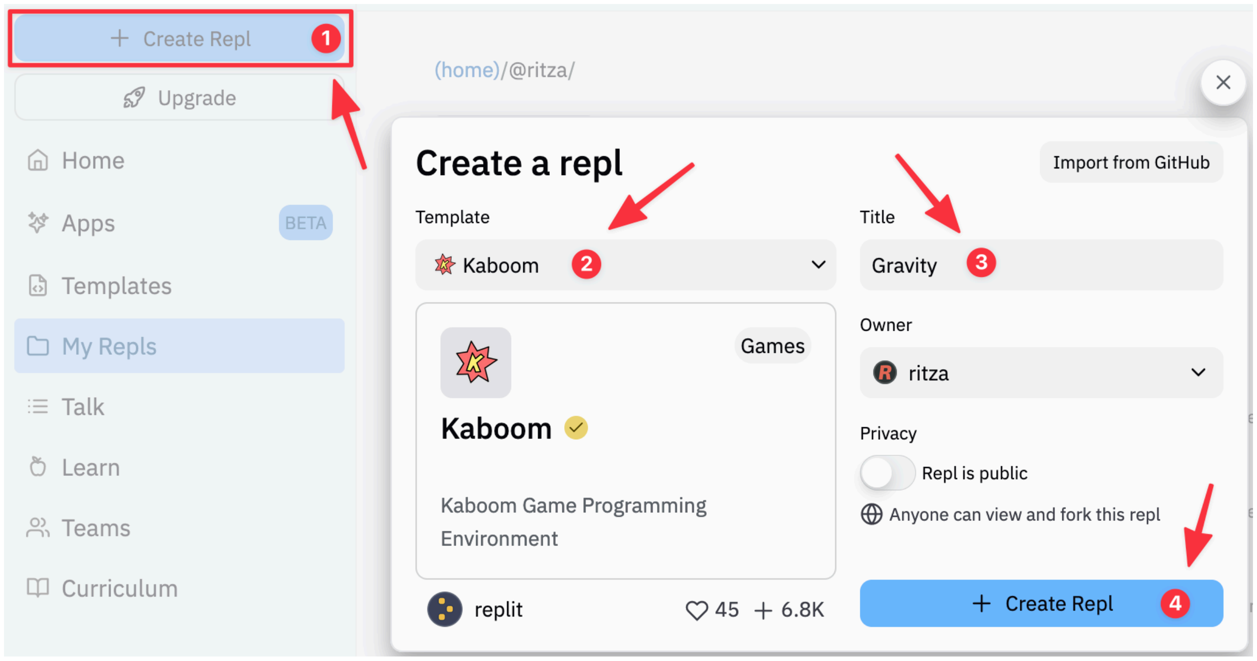
By the end of this tutorial, you will:

- Be familiar with advanced usage of the Kaboom JavaScript game development framework.
- Be able to build a simple 2D physics engine.
- Have the basis for a physics-based platformer, which you can extend into a full game.

Getting started

Log into your Replit[414] account and create a new repl. Choose **Kaboom** as your project type. Give this repl a name, like "gravity".

[411]https://kaboomjs.com/
[412]https://en.wikipedia.org/wiki/Gravity_gun
[413]https://docs.replit.com/images/tutorials/36-physics-playground/gameplay.gif
[414]https://replit.com



Creating a new Repl

Kaboom repls are quite different from other kinds of repls you may have seen before: instead of dealing directly with files in folders, you'll be dealing with code, sounds and sprites, the latter of which you can draw directly in Replit's image editor.

Setting the scene

When you first open your new Kaboom repl, you'll be greeted by a file containing the sample code below.

```
import kaboom from "kaboom";

// initialize context
kaboom();

// load assets
loadSprite("bean", "sprites/bean.png");

// add a character to screen
add([
    // list of components
    sprite("bean"),
    pos(80, 40),
    area(),
]);
```

This code loads a sprite for Kaboom's mascot Bean, and places Bean near the top of the screen. Before we start coding, we'll remove the code for adding Bean, leaving the following lines:

```
import kaboom from "kaboom";

// initialize context
kaboom();
```

Our edited code initialises Kaboom and gives us a blank canvas to work with. We'll start by defining a level containing walls, movable crates, and the player object. Add the following code beneath `kaboom();`:

```
// level
addLevel([
    "========================",
    "=                      =",
    "=                      =",
    "=                      =",
    "=                      =",
    "=            @         =",
    "=     ##               =",
    "=     ==========       =",
    "=                      =",
    "=                      =",
    "=                      =",
    "=                      =",
    "=                      =",
    "=                      =",
    "=                      =",
    "=                      =",
    "========================"
], {
    width: 32,
    height: 32,
    "=": () => [ // wall
        rect(32, 32),
        color(CYAN),
        area(),
        solid(),
        "wall"
    ],
    "#": () => [ // crate
        rect(32, 32),
        color(RED),
        z(1),
        outline(1),
        origin("center"),
        area(),
        body(),
        "movable"
    ],
    "@": () => [ // player
        circle(32),
        color(GREEN),
        z(2),
        outline(2),
        area({ width: 64, height: 64}),
        body(),
        origin("center"),
        {
            speed: 120,
            jumpspeed: 1000
        },
        "player"
    ],
});
```

This code uses Kaboom's `addLevel()`[415] function to visually construct a level. This function takes two arguments: an ASCII art representation of the level, and a JSON object defining the width and height of individual blocks and providing definitions for each of the objects used. Let's take a closer look at each of these definitions, starting with the wall object.

```
    "=": () => [ // wall
        rect(32, 32),
        color(CYAN),
        area(),
        solid(),
        "wall"
    ],
```

A game object definition[416] in Kaboom is a list of components and tags, and optionally custom attributes and functions. Components are a core part of Kaboom – they provide different functionality to game objects, from an object's appearance to functionality such as collision detection.

This wall object has four components:

[415]https://kaboomjs.com/#addLevel
[416]https://kaboomjs.com/#add

- `rect()`[417], which draws a rectangle to represent the object.
- `color()`[418], which gives the rectangle a color.
- `area()`[419], which provides collision detection for the object.
- `solid()`[420], which will prevent other objects from moving past it.

We've given the object the tag "wall". Objects can have multiple tags, which can be used to define custom behavior, such as collision detection between objects with particular tags.

Next, let's look at the crate definition:

```
    "#": () => [ // crate
        rect(32, 32),
        color(RED),
        z(1),
        outline(1),
        origin("center"),
        area(),
        body(),
        "movable"
    ],
```

Compared to our wall, we've given our crate the following additional components:

- `z()`[421], which defines the crate's z-order[422], ensuring it will be drawn on top of walls.
- `outline()`[423], which will draw a single-pixel outline around the object.
- `origin()`[424], which sets the sprite's origin to "center", so that we can move it around from its center, rather than the default top-left corner.
- `body()`[425], which makes our crate subject to gravity.

We've given it the tag "movable", a generic tag we can use for all objects that can be manipulated by the player's gravity gun.

Finally, let's look at the player object:

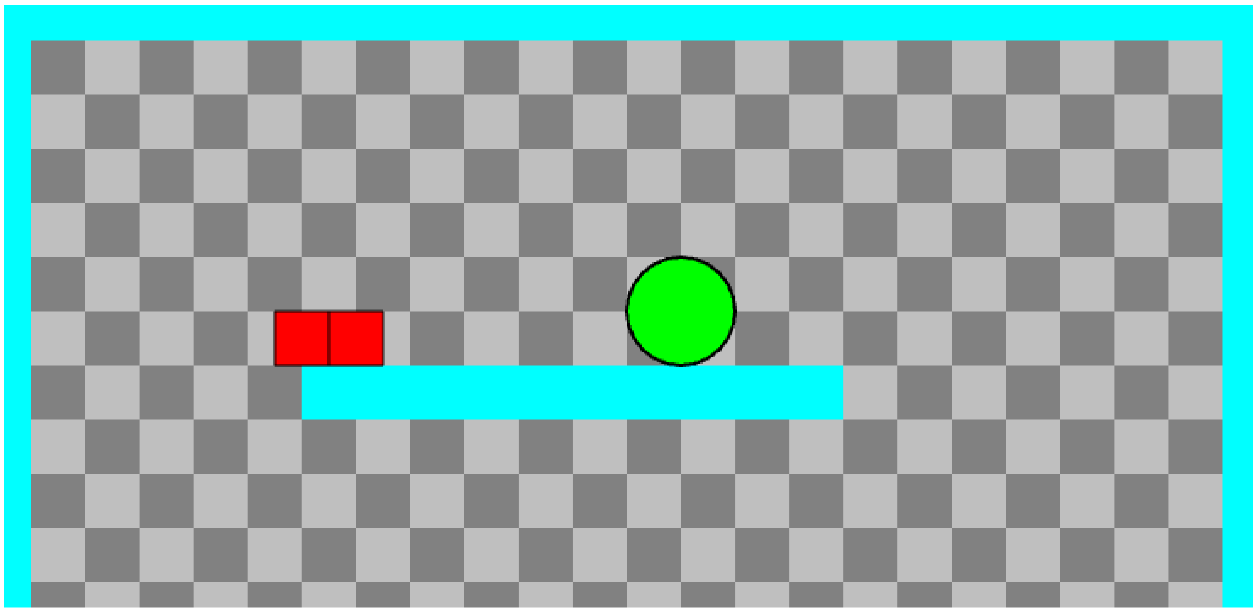
[417]https://kaboomjs.com/#rect
[418]https://kaboomjs.com/#color
[419]https://kaboomjs.com/#area
[420]https://kaboomjs.com/#solid
[421]https://kaboomjs.com/#z
[422]https://en.wikipedia.org/wiki/Z-order
[423]https://kaboomjs.com/#z
[424]https://kaboomjs.com/#origin
[425]https://kaboomjs.com/#body

```
    "@": () => [ // player
        circle(32),
        color(GREEN),
        z(2),
        outline(2),
        area({ width: 64, height: 64}),
        body(),
        origin("center"),
        {
            speed: 120,
            jumpspeed: 1000
        },
        "player"
    ],
```

Note the following:

- As we're using the `circle`[426] component to draw the player, we must specify a width and height for the `area()` component. All collision areas in Kaboom.js are rectangular (as of v2000.1.6).
- We've added `speed` and `jumpspeed` custom variables to the player object, which we'll use to control its movement speed and jump height.

Run your repl now, and you'll see your level, with player, crates and walls. As we've placed the player in mid-air, you should see them fall to the platform below, confirming the presence of gravity.



Level

[426]https://kaboomjs.com/#circle

Moving the player

Let’s write some code to control the player. First, we need to retrieve a reference to the player using `get()`[427]:

```
// player
player = get("player")[0];
```

Now we’ll add code to move the player left and right. As we’ll use the mouse to control the player’s gravity gun, it makes ergonomic sense to control the player with the WASD keys rather than the arrow keys. Add the following code:

```
onKeyDown("a", () => {
    player.move(-player.speed, 0)
});
onKeyDown("d", () => {
    player.move(player.speed, 0)
});
```

Here we detect the `onKeyDown`[428] event for the `A` key to move the player to the left and `D` key to move the player to the right. The `move()` function automatically checks for collisions with solid objects, so the player will only move if there is space to do so.

```
onKeyDown("w", () => {
    if (player.isGrounded()) {
        player.jump(player.jumpspeed)
    }
});
```

The `isGrounded()` and `jump()` functions are provided by the `body()` component, making basic platformer movement simple to implement in Kaboom.

Rerun your repl (or refresh your repl’s webview) now, and you should be able to move left and right with A and D, and to jump with W. If the player jumps off the platform, they will fall out of view. We can fix this by having the game camera follow the player. Add the following code below your keyboard-handling code:

[427]https://kaboomjs.com/#get
[428]https://kaboomjs.com/#onKeyDown

```
// camera follow player
player.onUpdate(() => {
    camPos(player.pos);
});
```

This code will run every frame[429] and keep the camera focused on the player as they move around. Refresh your in-repl browser and try it out.

Pulling objects with the gravity gun

Now that we've got our generic platformer functionality implemented, it's time to add the gravity gun. The player will aim their gravity gun with the mouse. It will have a range, which we'll show as a line. Movable objects that fall into that range will have a thicker outline drawn around them. When the player holds down the right mouse button, objects in range will travel towards them, until they reach the edge of a secondary, "holding" range. Objects in the holding range will move with the player until the left mouse button is clicked, at which point they will be launched into the air. Alternatively, the player can click the right mouse button to drop the held object.

In the code above, we've implemented horizontal movement (walking) and vertical movement (jumping). Implementing our gravity gun will require us to implement movement at arbitrary angles. To this end, we will need to write a couple of helper functions. Go to the top of your file, and add the following code beneath the `kaboom();` line:

```
// helper functions
function angleBetween(p1, p2) {
    return -Math.atan2(p2.y - p1.y, p2.x - p1.x);
}

function pointAt(distance, angle) {
    return vec2(distance*Math.cos(angle),
                -distance*Math.sin(angle));
}
```

The first function, `angleBetween()`, uses JavaScript's `Math.atan2()`[430] to determine the angle two points make with the horizontal. The second function, `pointAt()`, uses sines and cosines to determine the point at a certain distance and angle.

Next, we'll add two custom components:

- `gravitygun()`, which will define attributes and functions for our player's gravity gun.

[429]https://kaboomjs.com/#onUpdate
[430]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/atan2

- `physics()`, which will define attributes and functions for the movable objects the gravity gun will manipulate.

Add the following code for the `gravitygun()` component beneath your helper function definitions:

```
// custom components
function gravitygun(range, hold, firepower) {
    return {
        id: "gravitygun",
        require: ["pos"],
    }
}
```

Components are written as functions that return objects of a specific format. Each component should have an `id`, and a list of components it depends on attached to the `require` attribute. Beyond that, components can have arbitrary attributes, which will be assigned to their parent object, and special functions, which will be integrated into the game's event loop[431].

The only dependency for our component is `pos()`, as it will not work on an object that doesn't have a position in the world. Function arguments will allow us to customize this component, in the form of specifying the gravity gun's range, hold distance, and firepower.

Let's add some custom attributes to our component, to help us aim and fire the gravity gun. Expand your component code to match the following:

```
function gravitygun(range, hold, firepower) {
    return {
        id: "gravitygun",
        require: ["pos"],
        firepower: firepower, // new code from this line
        range: range,
        hold: hold,
        aimAngle: angleBetween(this.pos, mouseWorldPos()),
        rangeLine: {
            p1: this.pos,
            p2: this.pos,
            color: BLACK
        },
        holdLine: {
            p1: this.pos,
            p2: this.pos,
            color: BLUE
```

[431] https://kaboomjs.com/#on

```
        },
    }
}
```

In the above code:

- `firepower` is the amount of force to launch objects with.
- `range` is the length of our gravity gun's pulling range.
- `hold` is the distance from the player's center at which objects will be held once pulled in.
- `aimAngle` is the angle of our gravity gun's aim, which will be the angle created with the horizontal.
- `rangeLine` and `holdLine` are lines we will use to represent the gravity gun's range and hold.

Now we need to add a `draw()` function to draw both lines, and an `update()` function to recalculate the gravity gun's aim angle and the position of both lines as the mouse cursor moves around. Alter your component code to include these functions at the bottom:

```
function gravitygun(range, hold, firepower) {
    return {
        id: "gravitygun",
        require: ["pos"],
        firepower: firepower,
        range: range,
        hold: hold,
        aimAngle: angleBetween(this.pos, mouseWorldPos()),
        rangeLine: {
            p1: this.pos,
            p2: this.pos,
            color: BLACK
        },
        holdLine: {
            p1: this.pos,
            p2: this.pos,
            color: BLUE
        },
        draw() { // new code from this line
            drawLine(this.rangeLine);
            drawLine(this.holdLine);
        },
        update() {
            this.aimAngle = angleBetween(this.pos, mouseWorldPos());
            this.rangeLine.p1 = this.pos;
            this.rangeLine.p2 = this.pos.add(pointAt(this.range, this.aimAngle));
            this.holdLine.p1 = this.pos;
            this.holdLine.p2 = this.pos.add(pointAt(this.hold, this.aimAngle));
        }
    }
}
```

Our gravity gun component is complete. Now we need to add the `physics()` component, for objects affected by the gravity gun. Add the following code just below the last code you added:

```
function physics(mass) {
    return {
        id: "physics",
        require: ["area", "body"],
        mass: mass,
        inRange: false,
        inHold: false,
        held: false,
        dropping: false,
        direction: 0,
        speed: 0,
        draw() {
            if (this.inRange) {
                drawRect({
                    width: this.width,
                    height: this.height,
                    pos: vec2(this.pos.x - this.width/2, this.pos.y - this.height/2),
                    opacity: 0,
                    outline: { color: BLACK, width: 4}
                });
            }
        },
    }
}
```

This component requires both `area` and `body` – to be affected by the gravity gun, objects must have a collision area and be subject to regular gravity. We also give our physics objects a mass, direction and speed, and several variables to aid the gravity gun. Finally, we define a `draw()` function, which will draw a thick rectangle around the object when it is in gravity gun range.

Now we need to add the `gravitygun()` component to the player object and the `physics()` component to the crate object. Find your level creation code and add the new component lines to the objects as below:

```
    "#": () => [ // crate
        rect(32, 32),
        color(RED),
        z(1),
        outline(1),
        origin("center"),
        area(),
        body(),
        physics(100), // <-- NEW LINE
        "movable"
    ],
    "@": () => [
        circle(32),
        color(GREEN),
        z(2),
        outline(2),
        area({ width: 64, height: 64}),
        body(),
        origin("center"),
        gravitygun(150, 55, 40), // <-- NEW LINE
        {
            speed: 120,
            jumpspeed: 1000
        },
        "player"
    ],
```

We now have everything in place to write the code that will allow us to pull objects with the gravity gun. We'll do this in an `onUpdate()` event callback for objects with the "movable" tag. Add the following code to the bottom of the `main.js` file:

```
// gravity gun pull and hold
onUpdate("movable", (movable) => {
    // test collisions
    myRect = movable.worldArea()
    movable.inRange = testRectLine(myRect, player.rangeLine);
    movable.inHold = testRectLine(myRect, player.holdLine);
});
```

First, we get the coordinates for the movable object's collision rectangle, using `worldArea()`, a function provided by the `area()` component. We then use the `testRectLine()`[432] function to determine whether our player's range line or hold line intersects with this collision rectangle.

[432]https://kaboomjs.com/#testRectLine

Now let's have the gravity gun pull objects in range when the player holds down the right mouse button. Add the following code below the line where you assigned `movable.inHold` in the body of the `onUpdate` event callback:

```
    // gravity gun pull
    if (isMouseDown("right")) {
        if (movable.inHold) {
            movable.held = true;
        }
        else if (movable.inRange) {
            movable.moveTo(player.holdLine.p2);
        }
    }
```

If the object is in holding range, we set `held` to true. If the object is in pulling range, we move it into holding range. We use the `moveTo()` function, provided by the `pos()` component, without a speed argument. This means the object will teleport to the specified position.

Next, we'll handle holding objects once they've been pulled in. Add the following code beneath the code you added above:

```
    // gravity gun hold
    if (movable.held) {
        movable.moveTo(player.holdLine.p2)
    }
```

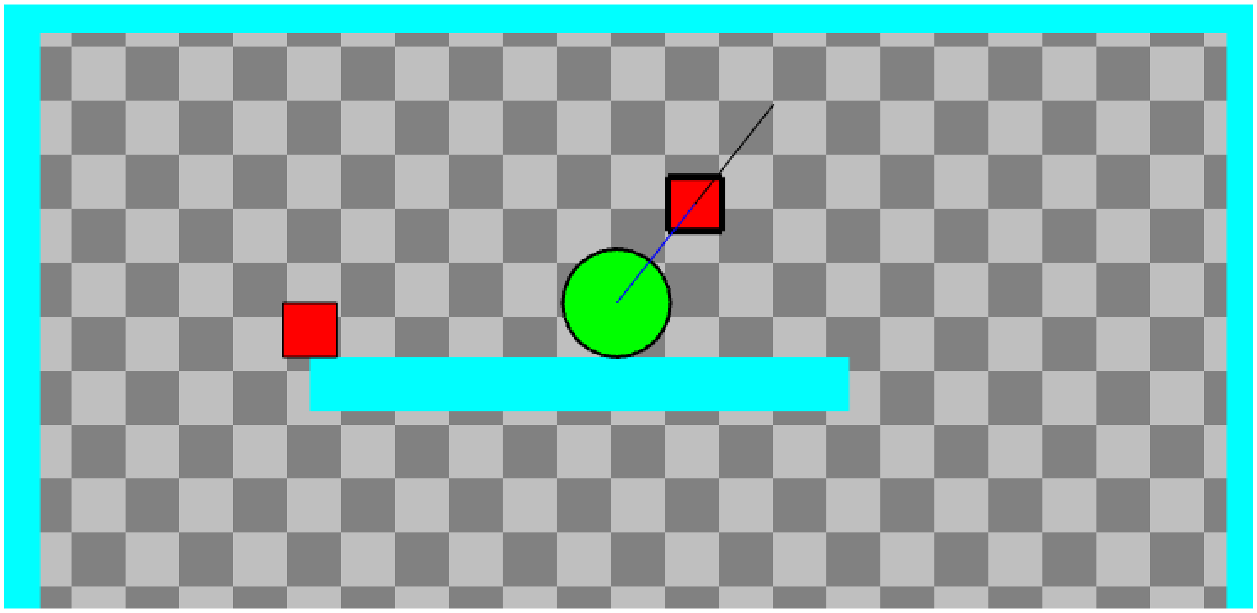
This code will ensure that objects stay in the holding position when held, so the player will be able to move them around in an arc.

Finally, we'll need to move our object after it's been launched. Add the following code below the lines you added above:

```
    // gravity gun launch
    movable.moveBy(dir(movable.direction).scale(movable.speed));
    movable.speed = Math.max(0, movable.speed-1); //friction
```

When we launch an object, we'll give it a direction and a non-zero speed. We use `moveBy()`, provided by the `pos()` component, to move the object unless there are other solid objects in the way. To give the object a more natural movement arc, we will simulate friction by decreasing its speed every frame until it reaches zero.

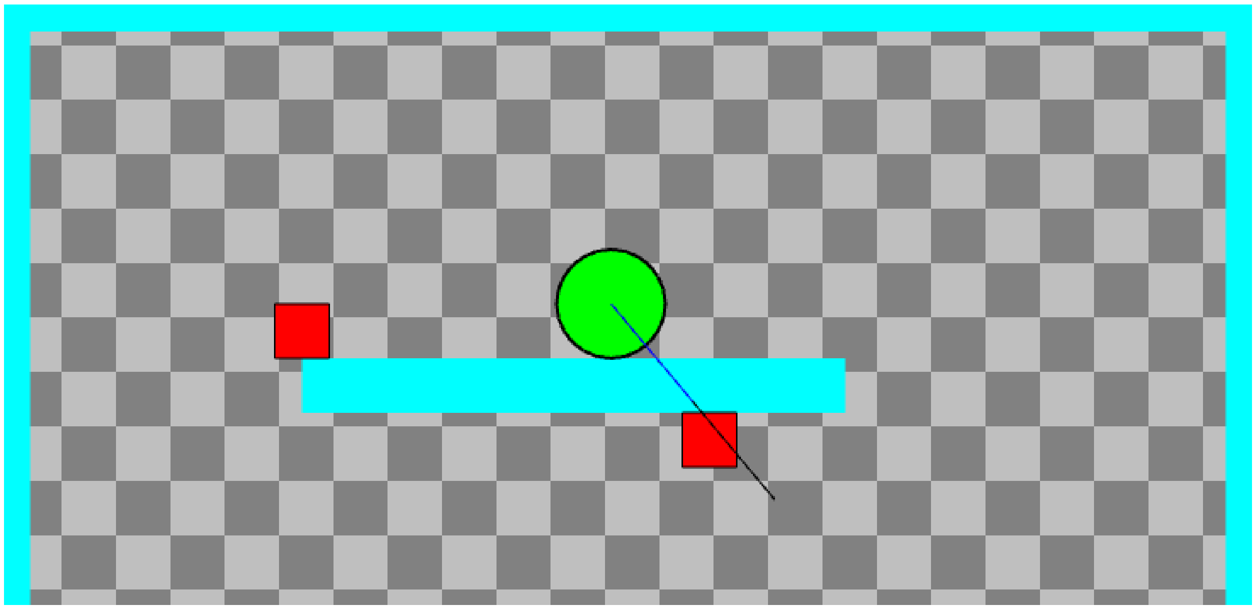
Run your repl now and try out the gravity gun. You should be able to pull crates into the holding position, and then move them around your head.



Gravity gun pull

Fixing the wall-clipping bugs

If you play around with the game, you should notice a couple of wall clipping bugs. When holding an object, you can force it to teleport below the platform you're standing on by aiming them. Similarly, objects will teleport through platforms if you pull them from the other side.



Wall clip

In both cases, this is because Kaboom's `moveTo()` function does not take solid objects into account, so we'll have to do that ourselves.

We'll fix the first bug by adding a new, collision-aware movement function to our `physics()` component. Find the component definition and append the function to the object it returns, just below the `draw()` function definition:

```
        moveToNoCollide(dest) {
            const diff = dest.sub(this.pos);
            return this.moveBy(diff.unit().scale(diff.len()));
        }
```

This function calculates the movement vector needed to move from the current position to the destination, and then passes this vector to the collision-aware function `moveBy()`. Our `moveBy()` function will return a `Collision` object if it detects a collision while moving. We'll return this and use it to cancel the hold if certain conditions are met.

Find the `if (movable.held)` block in your `onUpdate("movable")` event callback and alter it to match the code below:

```
    // gravity gun hold
    if (movable.held) {
        col = movable.moveToNoCollide(player.holdLine.p2)
        if (col != null && col.target.solid) {
            if (col.target.is("player")) { // disregard player collisions
                movable.moveTo(player.holdLine.p2);
            }
            else movable.held = false;
        }
    }
```

Here we've replaced `moveTo()` with `moveToNoCollide()`, and we're doing some checks on the `collision` object returned. If the collision is with the player, we ignore it and move the object back to the hold position. Otherwise, we cancel the hold.

Refresh your repl browser now and try to clip a held crate through the floor. You should be unsuccessful.

To fix the second clipping bug, we're going to need to detect solid objects between the movable object being pulled and the player's hold position. We'll write a new helper function to do this. Add the following code near the top of your file, just under the definition of `pointAt()`:

```
function checkCollisionLine(line, tag) {
    collision = get(tag).some((object) => {
        return testRectLine(object.worldArea(), line);
    });

    return collision;
}
```

The `checkCollisionLine()` function tests whether an object with a given tag intersects with a line. We do this by getting all objects with the tag[433] and testing each one until we find a collision. The `some()`[434] method will stop executing after a single `true` is returned, so this isn't quite as inefficient as it could be, but it's still probably not the optimal way to do this. Nonetheless, it should work fine while our game is relatively small.

To use this function, find the `if (isMouseDown("right"))` in your `onUpdate("movable")` event callback and make the modifications shown:

```
    // gravity gun pull
    if (isMouseDown("right")) {
        colLine = { // new object definition
            p1: movable.pos,
            p2: player.holdLine.p2
        }
        if (movable.inHold) {
            movable.held = true;
        }
        // expanded expression below
        else if (movable.inRange && !checkCollisionLine(colLine, "wall")) {
            movable.moveTo(player.holdLine.p2);
        }
    }
```

To prevent the player from pulling objects through walls, we check that the line between the object in range and the player's hold position is free of walls before moving it. Rerun your repl now, and you should be unable to clip objects through walls.

Dropping and launching objects with the gravity gun

Now that we have a stable pulling and holding implementation, we need to be able to drop and launch objects. Add the following code at the bottom of your file:

[433] https://kaboomjs.com/#get
[434] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/some

```
// gravity gun drop
onMousePress("right", () => {
    holdList = get("movable").filter((element) => {
        return element.held;
    });
    if (holdList.length) {
        selected = holdList[0];
        selected.held = false;
    }
});
```

When the right mouse button is clicked, we retrieve all the “movable”-tagged objects and filter[435] out the ones that aren’t currently held. Only one object can be held at a time, so we select that object and release it.

If you restart your repl now, you may have trouble getting dropping to work. Because we use the right mouse button both for pulling and dropping, you need to be quite precise to avoid immediately grabbing objects after dropping them.

Let’s implement a dropping timeout for dropped objects to make this more user-friendly. Add the following lines below `selected.held = false`:

```
        selected.dropping = true;
        wait(0.2, () => {
            selected.dropping = false;
        });
```

This code uses a `wait()`[436] callback, which will execute after the given number of seconds.

We defined the `dropping` attribute as part of our `physics()` component, so now all we need to do is make our pull respect the timeout. Find the `if (isMouseDown("right"))` line in your `onUpdate("movable")` event callback and alter it to resemble the following:

```
    // gravity gun pull
    if (isMouseDown("right") && !movable.dropping) {
```

Now you should be able to drop objects more easily.

Finally, let’s add some code to launch objects when the left mouse button is clicked. Enter the following code at the bottom of your file.

[435]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter
[436]https://kaboomjs.com/#wait

```
// gravity gun launch
onMousePress("left", () => {
    rangeList = get("movable").filter((element) => {
        return element.held;
    });
    if (rangeList.length) {
        selected = rangeList[0]; // 1st for now
        selected.held = false;
        selected.direction = rad2deg(-player.aimAngle);
        selected.speed = player.firepower;
    }
});
```

This code is largely similar to the dropping code, but instead of specifying a drop timeout, we give the previously held object a direction and a speed. The direction will be the player's `aimAngle`, which we need to convert to degrees[437] for Kaboom. The speed will be the `firepower` value we defined in our `gravitygun()` module.

Restart your repl and play around with picking up, dropping and launching crates. Some odd behavior you may notice is that objects launched straight will tend to slide along the ceiling for a while as they bleed off their speed. We can prevent this by adding the following code to kill a movable object's speed on collision with a wall:

```
onCollide("movable", "wall", (a, b) => {
    a.speed = 0;
});
```

Object collisions

Our physics playground wouldn't be complete without collisions between objects, so let's add some simple code to make objects push each other. Add the following collision handler to the bottom of your file:

[437]https://kaboomjs.com/#rad2deg

```
// physics object collisions
onCollide("movable", "movable", (a, b) => {
    if (a.speed > b.speed) b.direction = a.direction;
    else if (a.speed <= b.speed) a.direction = b.direction;
    finalSpeed = (a.speed) + (b.speed) / (a.mass + b.mass);
    a.speed = finalSpeed;
    b.speed = finalSpeed;
});
```

This code will assign the direction of the fastest object to both objects, and determine the speed of both from an altered form of a simple one-dimensional collision formula[438]. Play around with the values and calculations and see what sort of behavior you prefer. Remember, it's more important for a game to be fun than for it to be realistic.

Next steps

We've built a rudimentary physics-based platform engine. From here, there's a lot you can do to expand this into a full game. Consider the following ideas:

- Add enemies to throw crates at.
- Add more movable objects of different sizes and masses.
- Add puzzles that can be solved by arranging crates in specific patterns, and launching them from specific angles.
- Improve the physics simulation with 2D collisions, rotating objects, and objects with different behaviors, such as bouncing and shattering.

Code

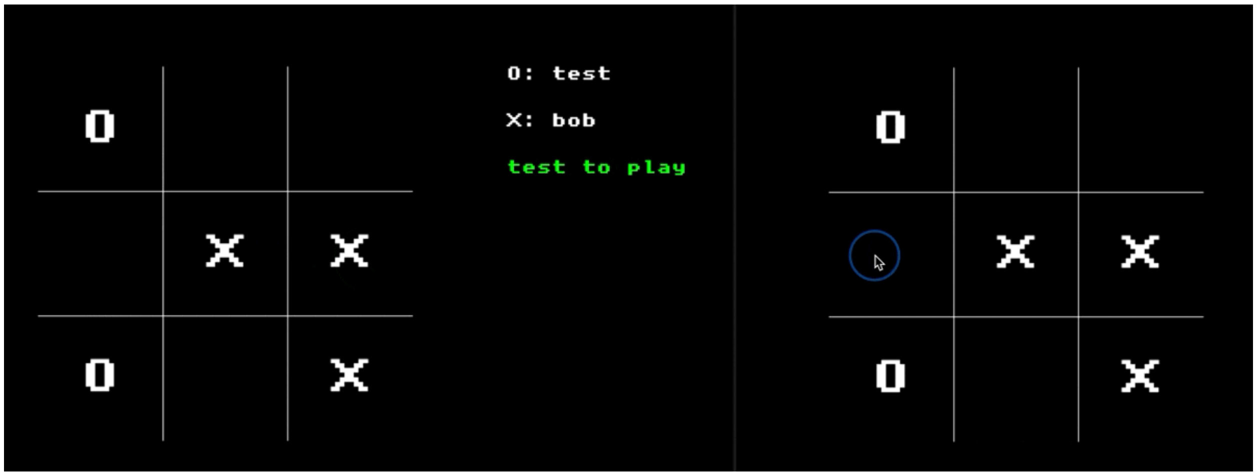
You can find the code for this tutorial on Replit[439]

[438]https://www.dummies.com/article/academics-the-arts/science/physics/how-to-find-the-velocity-of-two-objects-after-collision-174261
[439]https://replit.com/@ritza/Gravity

Building tic-tac-toe with WebSocket and Kaboom.js

Tic-tac-toe, or noughts and crosses, or Xs and Os, is a simple classic game for 2 players. It's usually played with paper and pen, but it also makes a good first game to write for networked multiplayer.

In this tutorial, we'll create a 2-player online tic-tac-toe game using a Node.js[440] server. Socket.IO[441] will enable realtime gameplay across the internet. We'll use Kaboom.js to create the game interface.



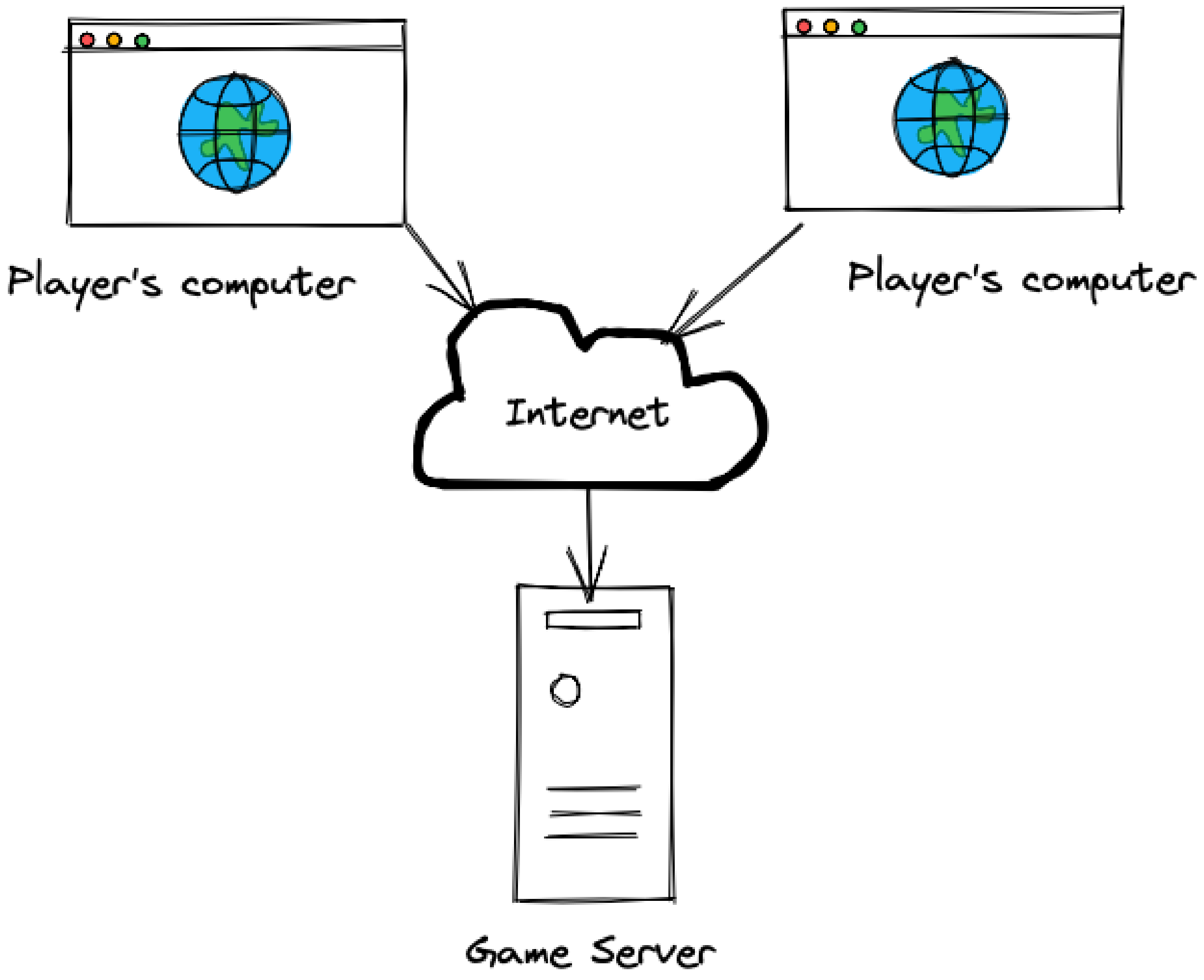
The finished game

Click to open gif[442]

How do multiplayer games work?

Multiplayer games have an architecture that typically looks something like this:

[440]https://nodejs.org/en/
[441]https://socket.io
[442]https://docs.replit.com/images/tutorials/27-tictactoe-kaboom/gameplay.gif



Game server architecture

Players (clients) connect to a game server over the internet. The game runs on the game server, where all the game rules, scores and other data are processed. The players' computers render the graphics for the game, and send player commands (from the keyboard, mouse, gamepad, or other input device) back to the game server. The game server checks if these commands are valid, and then updates the game state. The game state is a representation of all the variables, players, data and information about the game. This game state is then transmitted back to all the players and the graphics are updated.

A lot of communication needs to happen between a player's computer and the game server in online multiplayer games. This generally requires a 2-way, or bidirectional, link so that the game server can send data and notify players of updates to the game state. This link should ideally be quick too, so a more permanent connection is better.

With the HTTP[443] protocol that websites usually use, a browser opens a connection to a server, then

[443]https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

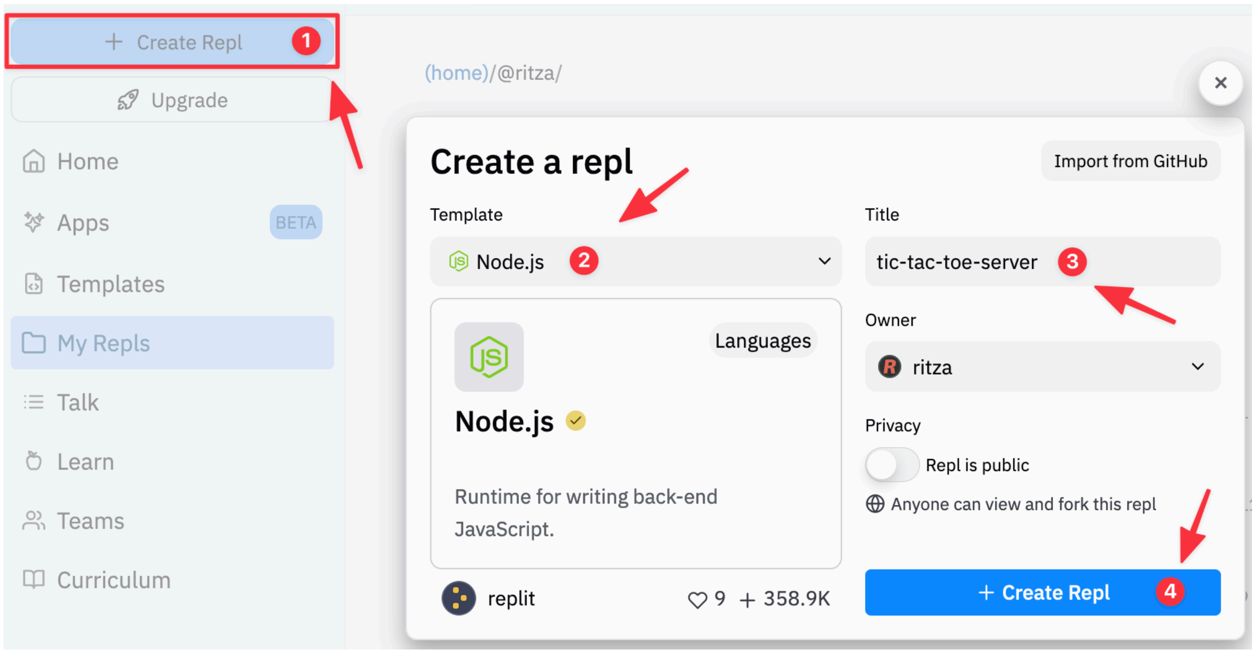
makes a request to the server, and the server sends back data, and closes the connection. There is no way for the server to initiate sending data to the browser. HTTP is also heavy on overhead, since it opens and closes a connection each time data is requested and sent.

WebSocket[444] is an advanced internet protocol that allows us to create a 2-way, persistent connection between a browser and a server. We'll use the Socket.IO[445] package to help us manage WebSocket connections in this project.

Creating a new project

For this project, we'll need to create 2 repls - 1 using Node.js for the game server, and 1 using Kaboom for the players. Head over to Replit[446] and create a two new repls:

- To create the server project, choose "Node.js" as your project type. Give this repl a name, like "tic-tac-toe-server".



Create a repl

Template: Node.js 2

Title: tic-tac-toe-server 3

Owner: ritza

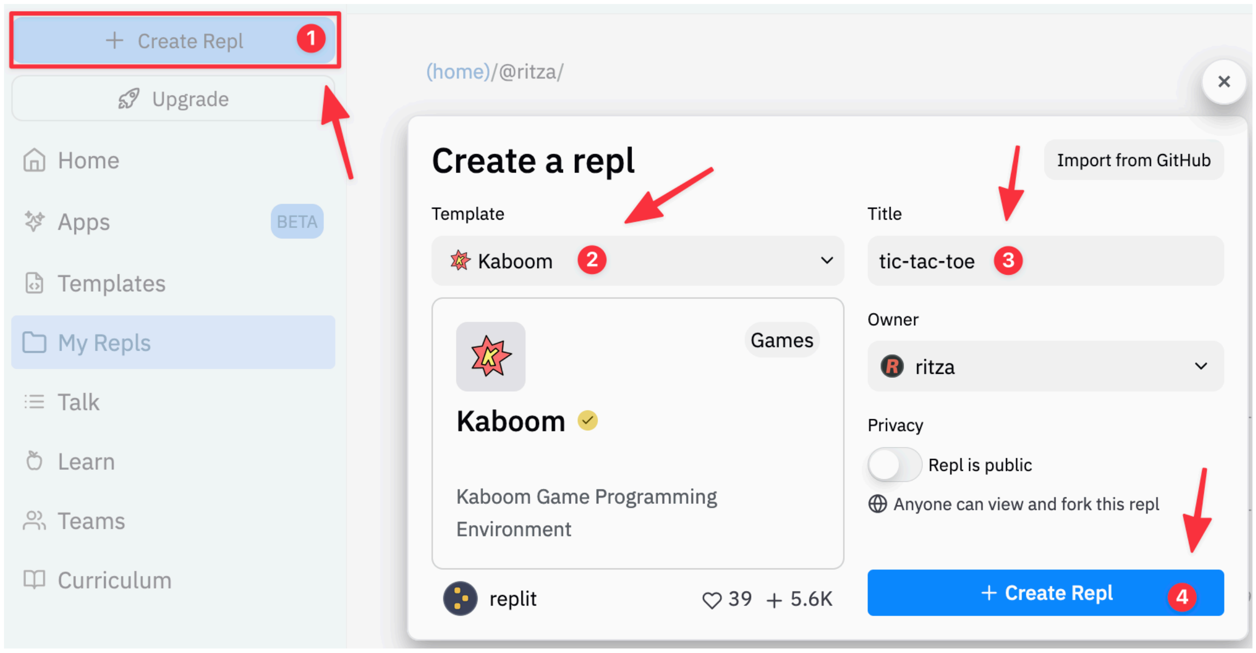
Privacy: Repl is public

Anyone can view and fork this repl

New Server repl

- To create the player project, choose "Kaboom" as your project type. Give this repl a name, like "tic-tac-toe".

444 https://en.wikipedia.org/wiki/WebSocket
445 https://socket.io
446 https://replit.com



New Player repl

We'll code in the server repl to start, and then switch between repls as we build the game.

Setting up Socket.IO on the server

Add the following code to the file called `index.js` in the server project to import Socket.IO:

```
const http = require('http');
const sockets = require('socket.io')

const server = http.createServer();
const io = sockets(server,  {
  cors: {
    origin: "https://tic-tac-toe.<YOUR-USER-NAME>.repl.co",
    methods: ["GET", "POST"]
  }
});

server.listen(3000, function() {
  console.log('listening on 3000');
});
```

In the first 2 lines, we import the built-in node `http`[447] package and the `socket.io`[448] package. The

[447]https://nodejs.org/api/http.html
[448]https://socket.io

`http` package enables us to run a simple HTTP server. The `socket.io` package extends that server to add WebSocket[449] functionality.

To create the HTTP server, we use the `http.createServer();` method. Then we set up Socket.IO by creating a new `io` object. We pass in the HTTP server object along with some configuration for CORS[450]. CORS stands for "Cross Origin Resource Sharing", and it's a system that tells the server which other sites are allowed to connect to it and access it. We set `origin` to allow our player repl to connect. Replace the `origin` value with the URL of the player project repl you set up earlier.

In the last 2 lines, we start the server up by calling its `listen` method, along with a port to listen on. We use 3000, as this is a standard for Node.js. If it starts successfully, we write a message to the console to let us know.

We'll add the rest of the server code above the `server.listen` line, as we only want to start the server up after all the other code is ready.

Tracking the game state

Now that we have a server, lets think a bit about how we will represent, or model, the game. Our tic-tac-toe game will have a few different properties to track:

- The status of the game: What is currently happening? Are we waiting for players to join, are the players playing, or is the game over?
- The current positions on the tic-tac-toe board. Is there a player in a grid block, or is it empty?
- All the players. What are their names, and which symbol are they using, X or O?
- The current player. Whose turn is it to go?
- If the game ends in a win, who won it?

For the status of the game, we'll add an enumeration[451] of the possible states we can expect. This makes it easier to track and use them as we go through the different phases of the game.

```
const Statuses = {
  WAITING: 'waiting',
  PLAYING: 'playing',
  DRAW: 'draw',
  WIN: 'win'
}
```

- WAITING: We are waiting for all the players to join the game.
- PLAYING: The players can make moves on the board.

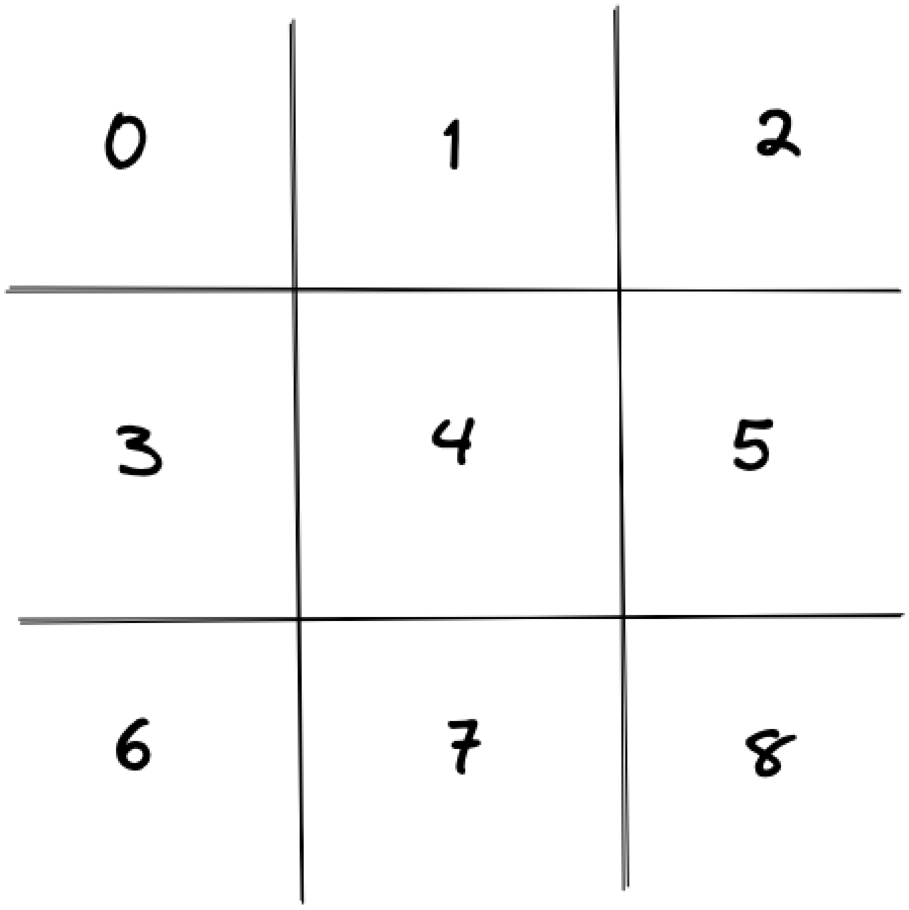
[449]https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
[450]https://developer.mozilla.org/en-US/docs/Glossary/CORS
[451]https://masteringjs.io/tutorials/fundamentals/enum

- DRAW: The game has ended in a draw.
- WIN: A player has won the game.

Now let's add a game state object to track everything:

```
let gameState = {
  board: new Array(9).fill(null),
  currentPlayer: null,
  players : [],
  result : {
    status : Statuses.WAITING
  }
}
```

First, we have a representation of the tic-tac-toe board as an array with 9 elements. This is how the array elements are mapped to the board:



Tic-Tac-Toe board mapped to array indices

Each number in the blocks represents the index at which the board position is represented in the array. Initially, we fill all the elements of the array with `null` to indicate that the block is open. When players make a move to occupy an open space, we'll add a reference to the player instead. That way we can keep track of which blocks are empty, and which are occupied by which player.

Next, we have `currentPlayer`, which we will alternately set to each player when it's their turn to move.

Then there is an array called `players`, which will hold references to both of the players in the game. This will allow us to show the names of the players on screen, as well as generally keep track of the players.

The `result` field is updated after every move. This field will contain the status of the game (as we defined above). As it's represented as an object, it will also be able to hold extra fields. We'll use that functionality to add a reference to the winner of the game, if the game ends in a win.

Accepting connections

When a player connects via WebSocket, Sockets.IO will fire a `connection`[452] event. We can listen for this event and handle tracking the connection, as well as creating listeners for other custom events. There are a few custom events[453] we can define here, that our players will emit:

- `addPlayer`: We'll use this event for a player to request joining the game.
- `action`: This is used when a player wants to make a move.
- `rematch`: Used when a game is over, but the players want to play again.

We can also listen for the built-in `disconnect`[454] event, which will alert us if a player leaves the game (for example, by closing the browser window or if their internet connection is lost).

Let's add the code that will hook up our listeners to the events:

```
io.on('connection', function (connection) {
  connection.on('addPlayer', addPlayer(connection.id));
  connection.on('action', action(connection.id));
  connection.on('rematch', rematch(connection.id));
  connection.on('disconnect', disconnect(connection.id));
});
```

Next we'll implement each of these listener functions, starting with `addPlayer`.

Side Note: Normally in examples for custom listeners, you'll see the handler code added immediately with an anonymous function, like this:

[452]https://socket.io/docs/v4/server-instance/#connection
[453]https://socket.io/docs/v4/emitting-events/
[454]https://socket.io/docs/v4/server-socket-instance/#disconnect

```
io.on('connection', function (connection) {
  connection.on('addPlayer', (data)=> {
          // some code here
  });

   connection.on('action', (data)=> {
          // some code here
  });

  // etc ...
});
```

This is convenient, especially when there are a couple of handlers, each with only a small amount of code. It's also handy because in each of the handler functions, you still have access to the `connection` object, which is not passed on each event. However, it can get a little messy and unwieldy if there are many event handlers, with more complex logic in each.

We're doing it differently so that we can separate the handlers into functions elsewhere in the code base. We do have one problem to solve though: if they are separate functions, how will they access the `connection` parameter in such a way that we can tell which player sent the command? With the concept of closures[455], which are well-supported in Javascript, we can make functions that return another function. In this way, we can pass in the `connection.id` parameter to the first wrapping function, and it can return another function that takes the data arguments from the Socket.IO event caller. Because the second function is within the closure of the first, it will have access to the `connection.id` parameter. The pattern looks like this:

```
io.on('connection', function (connection) {
  connection.on('addPlayer', addPlayer(connection.id));
  connection.on('action', action(connection.id)) ;
  // etc ...
});


function addPlayer(socketId){
        return (data)=>{
                // code here
        }
}

function action(socketId){
        return (data)=>{
                // code here
```

[455]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures

```
        }
}
```

Handling new players

Add the following function to handle adding players:

```
function addPlayer(socketId){
  return (data)=>{
    const numberOfPlayers = gameState.players.length;
    if (numberOfPlayers >= 2){
      return;
    }

    let nextSymbol = 'X';
    if (numberOfPlayers === 1){
      if (gameState.players[0].symbol === 'X'){
        nextSymbol = 'O';
      }
    }

    const newPlayer = {
      playerName: data.playerName,
      id: socketId,
      symbol: nextSymbol
    };

    gameState.players.push(newPlayer);
    if (gameState.players.length === 2){
      gameState.result.status = Statuses.PLAYING;
      gameState.currentPlayer = newPlayer;
    }
    io.emit('gameState', gameState);

  }
}
```

This function does quite a bit. Let's go through the main features.

- First it checks to see how many players are already in the game. If there are already 2 players, it returns early without changing anything. If this check passes, it goes on to add a new player. Note that even when there is no space in the game for a new player, we don't disconnect the player - they still get updates and can watch the match.

- Next, the function figures out which symbol, X or O, the new player should be. It will assign X to the first player. If there is already a player, and the existing player's symbol is X, then it will assign O to the new player. Note that there is a possible case where there is only one player, and their symbol is O. This would occur if there are 2 players, and the player with the X symbol disconnects from the game, leaving only the player with the O symbol. This is why we always check what symbol the existing player in the game has.
- Then the function constructs a new player object with some identifying information, including the name that the player sends through, the `socketId` they connected on, and their symbol. When a new player requests to join, we expect them to send an object with a field `playerName` to tell us their handle.
- Now we add the new player to the player array in our `gameState` object, so that they are part of the game.
- We go on to check if we have 2 players, and start playing if we do. We begin by updating the status of the game to `PLAYING`, and set the `currentPlayer`, i.e. the player who is first to go, as the latest player to have joined.
- Finally, we use the Socket.IO `emit`[456] function to send the updated `gameState` to all connections. This will allow them to update the players' displays.

Handling player actions

The next handler takes care of the moves players make. We expect that the incoming data from the player will have a property called `gridIndex` to indicate which block on the board the player wants to mark. This should be a number that maps to the numbers for each block in the board, as in the picture earlier on.

```
function action(socketId){
  return (data)=> {
    if (gameState.result.status === Statuses.PLAYING && gameState.currentPlayer.id =\
== socketId){
      const player = gameState.players.find(p => p.id === socketId);
      if (gameState.board[data.gridIndex] == null){
        gameState.board[data.gridIndex] = player;
        gameState.currentPlayer = gameState.players.find(p => p !== player);
        checkForEndOfGame();
      }
    }
    io.emit('gameState', gameState);
  }
}
```

In this function, we check a couple of things first:

[456] https://socket.io/docs/v4/emitting-events/

- The game status must be `PLAYING` - players can't make moves if the game is in any other state.
- The player attempting to make the move must be the `currentPlayer`, i.e. the player whose turn it is to go.

If these conditions are met, we find the player in the `gameState.players` array using the built-in `find`[457] method on arrays, by looking for the player by their `socketId`.

Now we can check if the board position (`gridIndex`) requested by the player is available. We check that the value for that position in the `gameState.board` array is `null`, and if it is, we assign the player to it.

The player has made a successful move, so we give the other player a turn. We switch the `gameState.currentPlayer` to the other player by using the array `find`[458] method again, to get the player who does not match the current player.

We also need to check if the move the player made changed the status of the game. Did that move make them win the game, or is it a draw, or is the game still in play? We call out to a function `checkForEndOfGame` to check for this. We'll implement this function a little later, after we're done with all the handlers.

Finally, we send out the latest `gameState` to all the players (and spectators) to update the game UI.

Handling a rematch request

Let's make it possible for a player to challenge their opponent to a rematch when the game has ended:

```
function rematch(socketId){
  return (data) => {
    if (gameState.players.findIndex(p=> p.id === socketId) < 0) return; // Don't let\
 spectators rematch
    if (gameState.result.status === Statuses.WIN || gameState.result.status === Stat\
uses.DRAW){
      resetGame();
      io.emit('gameState', gameState);
    }
  }
}
```

This function first checks if the connection sending the rematch request is actually one of the players, and not just a spectator. If we can't find a match for a player, we return immediately, making no changes.

[457] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/find
[458] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/find

Then we check if the game is in one of the final states, either `WIN` or `DRAW`. If it is, we call out to a function `resetGame` to set up the game again. Finally, we send out the latest `gameState` to all the players.

Let's implement the `resetGame` function:

```
function resetGame(){
  gameState.board = new Array(9).fill(null);

  if (gameState.players.length === 2){
    gameState.result.status = Statuses.PLAYING;
    const randPlayer = Math.floor(Math.random() * gameState.players.length);
    gameState.currentPlayer = gameState.players[randPlayer];
  } else {
    gameState.result.status = Statuses.WAITING;
    gameState.currentPlayer = null;
  }
}
```

Let's take a look at what we're doing here:

- First, our function creates a new array for the `gameState` board. This effectively clears the board, setting all the positions back to `null`, or empty.
- Then it checks that there are still `2` players connected. If there are, it sets the game status back to `PLAYING` and chooses at random which player's turn it is to go. We choose the first player randomly so that there isn't one player getting an advantage by going first every time.

If there is only one player remaining, we set the game status to `WAITING` instead, and listen for any new players who want to join. We also set the `currentPlayer` to null, as we will choose which player should go once the new player has joined.

Handling disconnects

The last handler we need to implement is if a connection to a player is lost. This could be because the player has exited the game (by closing the browser tab), or has other internet issues.

```
function disconnect(socketId){
  return (reason) => {
    gameState.players = gameState.players.filter(p => p.id != socketId);
    if (gameState.players !== 2){
      resetGame();
      io.emit('gameState', gameState);
    }
  }
}
```

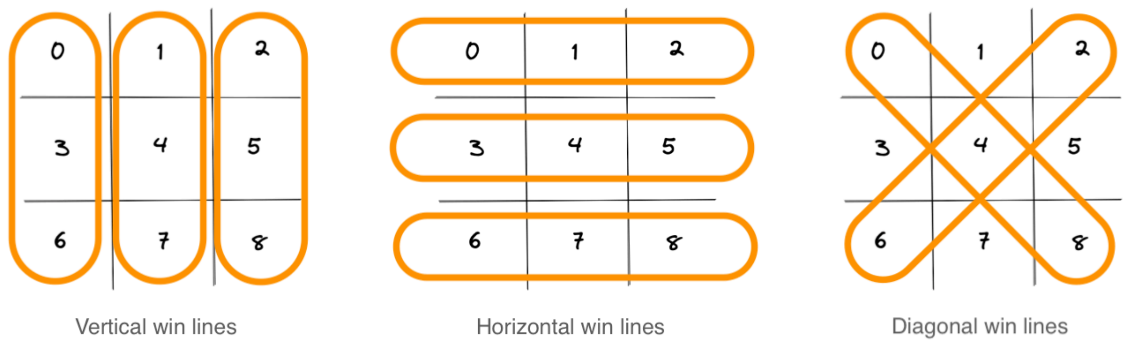
This function uses the built-in array `filter`[459] function to remove the player that disconnected from the server. Since it's possible that the disconnect event isn't from a player but from a spectator, we check the number of players left after filtering the disconnecting socket from the player list. If there aren't 2 players remaining after filtering, we reset the game and send out the updated game state.

Checking for the end of the game

Now we can get back to implementing the `checkForEndOfGame()` function we referenced in the `action` handler.

We're only interested in detecting 2 cases: A win or a draw.

There are just 8 patterns that determine if a player has won at tic-tac-toe. Let's map them to our board with its indexed blocks:



All possible win lines

We can encode each of these winning patterns into an array of 3 numbers each. Then we can add each of those patterns to a larger array, like this:

[459] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter

```
const winPatterns = [
  [0, 1, 2],
  [3, 4, 5],
  [6, 7, 8],
  [0, 3, 6],
  [1, 4, 7],
  [2, 5, 8],
  [0, 4, 8],
  [2, 4, 6]
]
```

Now that we have each winning pattern in an array, we can loop through each of them to see if there is a player that has positions that match any of the patterns.

Since the players are also in an array in `gameState.players`, we can loop through that array, and check each player against the winning pattern array. If a player matches any of these patterns, we can change the game status to `WIN` and set that player as the winner in the results.

Here is the code to do that:

```
function checkForEndOfGame(){

  // Check for a win
  gameState.players.forEach(player => {
    winPatterns.forEach(seq => {
      if (gameState.board[seq[0]] == player
          && gameState.board[seq[1]] == player
          && gameState.board[seq[2]] == player){
            gameState.result.status = Statuses.WIN;
            gameState.result.winner = player;
         }
    });
  });

  // Check for a draw
  if (gameState.result.status != Statuses.WIN){
    const emptyBlock = gameState.board.indexOf(null);
    if (emptyBlock == -1){
      gameState.result.status = Statuses.DRAW;
    }
  }
}
```

We also check for a draw in this function. A draw is defined as when all the blocks are occupied (no more moves can be made), but no player has matched one of the win patterns. To check if there are no more empty blocks, we use the array method `indexOf`[460] to find any `null` values in `gameState.board` array. Remember that `null` means an empty block here. The `indexOf` method will return `-1` if it can't find any `null` values. In that case, we set the game status to `DRAW`, ending the game.

Now we have all the functionality we need on the server, let's move on to building the Kaboom website the players will use to play the game.

Setting up Kaboom

To start, we need to set up Kaboom with the screen size and colors we want for the game window. Replace the code in `main.js` with the code below:

```
  import kaboom from "kaboom";

  kaboom({
    background: [0, 0, 0],
    width: 1000,
    height: 600
  });
```

This creates a new Kaboom canvas with a black background.

Setting up Kaboom with Socket.IO

Now we can add a reference to Socket.IO. Normally, in a plain HTML project, we could add a `<script>`[461] tag and reference the Socket.IO client script[462], hosted automatically on our game server. However, here we will add the script programmatically. We can do this by accessing the `document`[463] object available in every browser, and insert a new element with our script. Add the following code to the `main.js` file below the code to initialise Kaboom.

```
let script = document.createElement("script");
script.src = 'https://tic-tac-toe-server.<YOUR_USER_NAME>.repl.co' + '/socket.io/soc\
ket.io.js'
document.head.appendChild(script);
```

Replace the `<YOUR_USER_NAME>` part of the URL with your Replit username. This code inserts the new `<script>` tag into the `<head>`[464] section of the underlying HTML page that Kaboom runs in.

[460] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/indexOf
[461] https://www.w3schools.com/tags/tag_script.asp
[462] https://socket.io/docs/v4/client-installation/#Installation
[463] https://developer.mozilla.org/en-US/docs/Web/API/Document
[464] https://www.w3schools.com/tags/tag_head.asp

Let's move on to creating the relevant scenes for our game. Kaboom "scenes"[465] allow us to group logic and levels together. In this game we'll have 2 scenes:

- A "startGame" scene that will prompt for the player's name.
- A "main" scene, which will contain all the logic to play the tic-tac-toe game.

Let's move on to the code to prompt the player to enter their name.

```
scene("startGame", () => {
  const SCREEN_WIDTH = 1000;
  const SCREEN_HEIGHT = 600;

  add([
    text("What's your name? ", { size: 32, font: "sinko" }),
    pos(SCREEN_WIDTH / 2, SCREEN_HEIGHT / 3),
    origin("center")
  ]);

  const nameField = add([
    text("", { size: 32, font: "sinko" }),
    pos(SCREEN_WIDTH / 2, SCREEN_HEIGHT / 2),
    origin("center")
  ]);

  charInput((ch) => {
    nameField.text += ch;
  });

  keyRelease("enter", () => {
    go("main", { playerName: nameField.text });
  })

});

go("startGame");
```

To keep the calculations for the UI layout simpler, we'll use a fixed size for the screen. That's where the 2 constants for the screen width and height come in.

We use the Kaboom `add`[466] function to display the prompt "What's your name?" on the screen, using the `text`[467] component. We choose a position halfway across the screen, `SCREEN_WIDTH / 2`, and

[465]https://kaboomjs.com/#scene
[466]https://kaboomjs.com/doc/#add
[467]https://kaboomjs.com/doc/#text

about a third of the way down the screen, `SCREEN_HEIGHT / 3`. We add the `origin`[468] component, set to `center`, to indicate that the positions we set must be in the center of the text field.

Then we add another object with an empty `""` text component. This will display the characters the player types in. We position it exactly halfway down and across the screen. We also hold a reference to the object in the constant `nameField`.

To get the user's keyboard input, we use the Kaboom function `charInput`[469]. This function calls an event handler each time a key on the keyboard is pressed. We take that character and append it to the text in the `nameField` object. Now, when a player presses a key to enter their name, it will show up on the screen.

Finally, we use the Kaboom function `keyRelease`[470] to listen for when the player pushes the `enter` key. We'll take that as meaning they have finished entering their name and want to start the game. In the handler, we use the Kaboom `go`[471] function to redirect to the main scene of the game.

Adding the game board

Now we can add the UI elements for the game itself. Create the "main" scene in your Kaboom repl by adding the following code to draw the tic-tac-toe board:

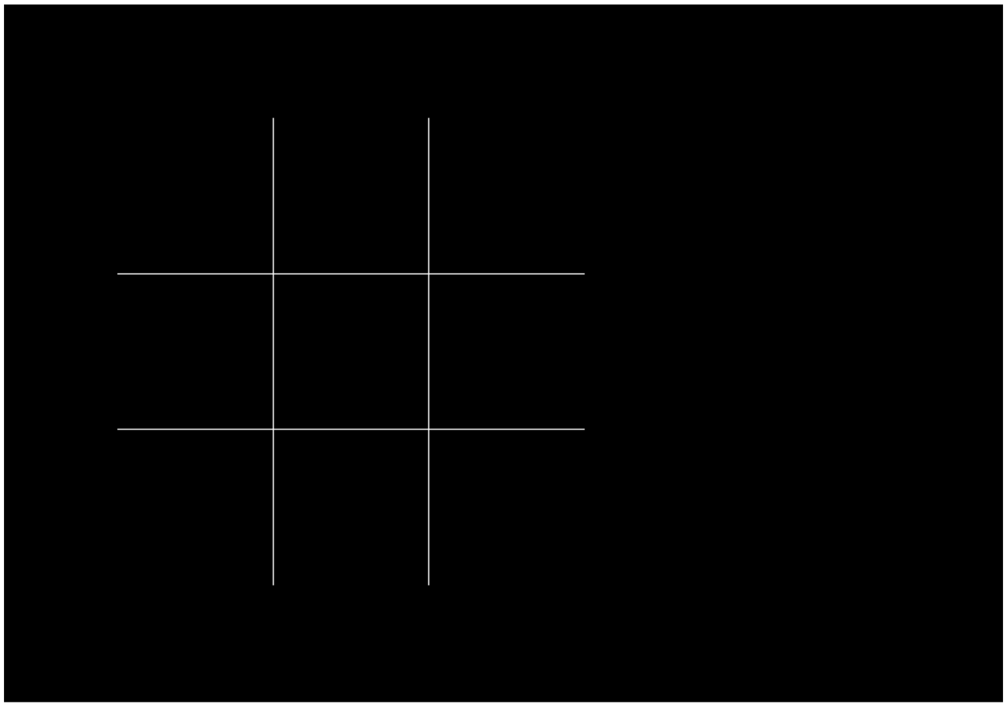
```
scene("main", ({ playerName }) => {

  // Board
  add([
    rect(1, 400),
    pos(233, 100),
  ]);

  add([
    rect(1, 400),
    pos(366, 100),
  ]);

  add([
    rect(400, 1),
    pos(100, 233),
  ]);

  add([
```

[468]https://kaboomjs.com/doc/#origin
[469]https://kaboomjs.com/doc/#charInput
[470]https://kaboomjs.com/doc/#keyRelease
[471]https://kaboomjs.com/doc/#go

```
    rect(400, 1),
    pos(100, 366),
  ]);

});
```

This adds 4 rectangles with a width of 1 pixel and length of 400 pixels to the screen - each rectangle is more like a line. This is how we draw the lines that create the classic tic-tac-toe board shape. The first 2 rectangles are the vertical lines, and the second 2 are the horizontal lines. We place the board closer to the left side of the screen, instead of the center, to save space for game information to be displayed on the right hand side of the screen.

If you run the game, and enter your name, you should see the board layout like this:



Board layout

Now we need to add a way to draw the X and O symbols in each block. To do this, we'll add objects with text components in each block of the board. First, we'll make an array containing the location and size of each block. Add the following code snippets within the "main" scene we created above:

```
const boardSquares = [
  {index: 0, x: 100, y: 100, width:133, height: 133 },
  {index: 1, x: 233, y: 100, width:133, height: 133 },
  {index: 2, x: 366, y: 100, width:133, height: 133 },
  {index: 3, x: 100, y: 233, width:133, height: 133 },
  {index: 4, x: 233, y: 233, width:133, height: 133 },
  {index: 5, x: 366, y: 233, width:133, height: 133 },
  {index: 6, x: 100, y: 366, width:133, height: 133 },
  {index: 7, x: 233, y: 366, width:133, height: 133 },
  {index: 8, x: 366, y: 366, width:133, height: 133 }
];
```

We can run through this array and create a text object that we can write to when we want to update the symbols on the board. Let's create a function to do that.

```
function createTextBoxesForGrid(){
  boardSquares.forEach((square)=>{
    let x = square.x + square.width*0.5;
    let y = square.y + square.height*0.5;
    square.textBox = add([
      text('', 40),
      pos(x, y),
      origin('center')
    ]);
  })
}

createTextBoxesForGrid();
```

This function uses the array `forEach`[472] method to loop through each "square" definition in the `boardSquares` array. We then find the center `x` and `y` of the square, and `add`[473] a new text object to the screen, and also add it to the square definition on the field `textBox` so we can access it later to update it. We use the `origin` component to ensure the text is centered in the square.

Finally, we call the function to create the text boxes.

Adding player names and game status

Now let's add some areas for the player's names and for the current status of the game (whose turn it is to play, if someone has won, or if it's a draw).

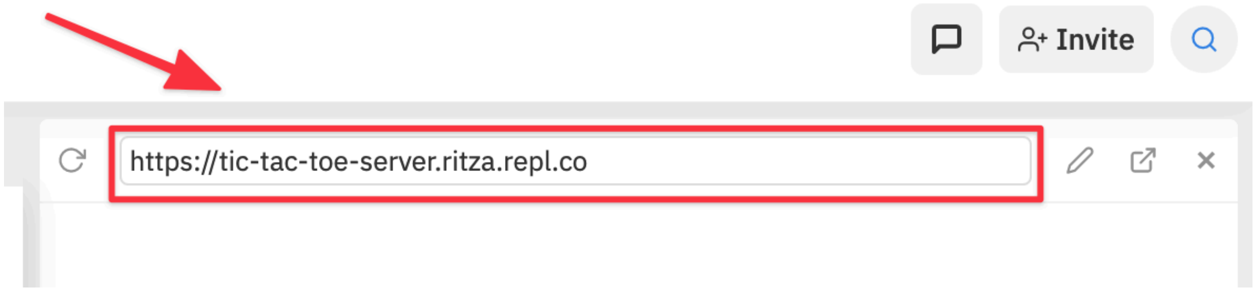
[472] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach
[473] https://kaboomjs.com/doc/#add

```
// Players and game status elements
const playerOneLabel = add([
  text('', { size: 20, font: "sinko" }),
  pos(600, 100),
]);

const playerTwoLabel = add([
  text('', { size: 20, font: "sinko" }),
  pos(600, 150),
]);

const statusLabel = add([
  text('', { size: 20, font: "sinko" }),
  pos(600, 200),
  color(0, 255, 0)
])
```

Here we add 3 objects with `text`[474] components. The first 2 are placeholders for the player names and symbols. The third one is for the game status. They are positioned to the right of the screen, and contain empty text to start. We'll change the contents as we receive new game states from the server. The last object has a `color` component to set the color of the text to green. This is to make the status message stand out from the rest of the text.

Connecting to the server

To connect to the game server, we need to initialize the Socket.IO library we dynamically added earlier. We need to provide the URL to the server repl, so copy that from the output window of the server repl:



https://tic-tac-toe-server.ritza.repl.co

Copying server url

Now add this code along with the server URL to the "main" scene in the player repl:

[474] https://kaboomjs.com/doc/#text

```
var socket = io('https://tic-tac-toe-server.<YOUR_USER_NAME>.repl.co');

socket.on('connect', function(){
  socket.emit("addPlayer", {
    playerName: playerName
  });
});
```

In the first line, we initialize the Socket.IO client library[475] to connect to the server. Then we add a listener to the `connect`[476] event. This lets us know when we have established a connection to the server.

If we have a connection, we then `emit`[477] an event to the server, with our custom event type `addPlayer`. We also add in the player name, which we passed to this scene from the `startGame` scene. Emitting the `addPlayer` event to the server will cause the `addPlayer` event handler to fire on the server side, adding the player to the game, and emitting back the game state.

Handling updated game state

Remember that our server emits a `gameState` event whenever something changes in the game. We'll listen for that event, and update all the UI elements in an event handler.

First, we need to add the definitions of each status as we have done on the server side, so that we can easily reference them in the code:

```
const Statuses = {
        WAITING: 'waiting',
        PLAYING: 'playing',
        DRAW: 'draw',
        WIN: 'win'
}
```

Now we can add a listener and event handler:

[475] https://socket.io/docs/v4/client-initialization/
[476] https://socket.io/docs/v4/client-socket-instance/#Socket-connected
[477] https://socket.io/docs/v4/emitting-events/

```
socket.on('gameState', function(state){
  for (let index = 0; index < state.board.length; index++) {
    const player = state.board[index];
    if (player != null){
      boardSquares[index].textBox.text = player.symbol;
    } else
    {
      boardSquares[index].textBox.text = '';
    }
  }

  statusLabel.text = '';
  switch (state.result.status) {
    case Statuses.WAITING:
      statusLabel.text = 'Waiting for players....';
      break;
    case Statuses.PLAYING:
      statusLabel.text = state.currentPlayer.playerName + ' to play';
    break;
    case Statuses.DRAW:
      statusLabel.text = 'Draw! \nPress R for rematch';
    break;
    case Statuses.WIN:
      statusLabel.text = state.result.winner.playerName + ' Wins! \nPress R for rema\
tch';
    break;
    default:
      break;
  }

  playerOneLabel.text = '';
  playerTwoLabel.text = '';
  if (state.players.length > 0){
    playerOneLabel.text = state.players[0].symbol + ': ' + state.players[0].playerNa\
me;
  }

  if (state.players.length > 1){
    playerTwoLabel.text = state.players[1].symbol + ': ' + state.players[1].playerNa\
me;
  }

});
```

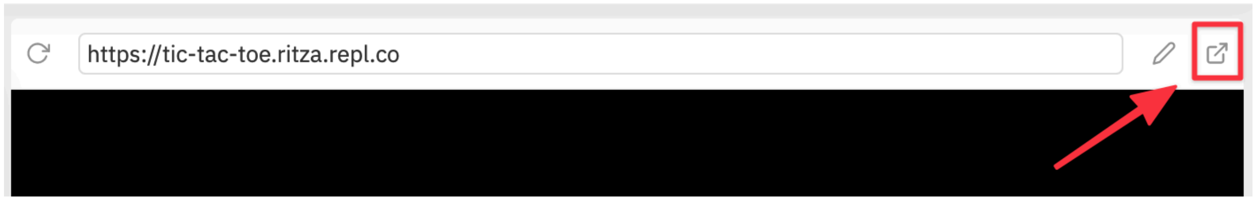
This function looks quite long, but it's mainly just updating the text boxes we added.

First, we loop through the board positions array that is passed from the server on the `state` payload, to check each block for a player positioned on it. If there is a player on a block, we write that player's symbol to the corresponding text box, found in the `boardSquares` array we created above. If there is no player in the block, i.e it's a `null` value, we write an empty string to the text block.

Then we update the `statusLabel` to show what is currently happening in the game. We use a `switch`[478] statement to create logic for each of the possibilities. We write a different message to the `statusLabel` text box depending on the status, drawing from data in the `gameState` object.

Next we update the player name text boxes. First we reset them, in case one of the players has dropped out. Then we update the text boxes with the players' symbols and names. Note that we first check if there are the corresponding players in the array.

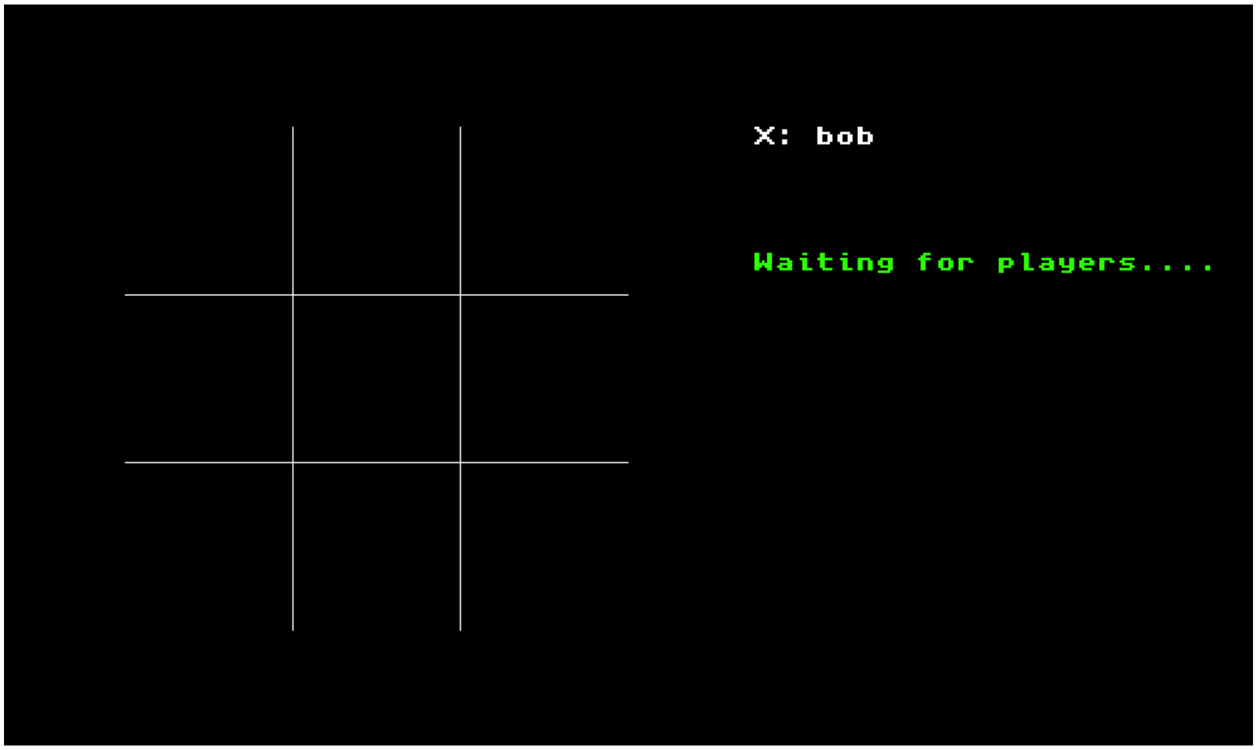
Now that we're done with updating from the game state, let's try running the game again. Open the game window in a new tab so that requests to the repl server don't get blocked by the browser due to the CORS header 'Access-Control-Allow-Origin' not matching in the embedded window.



Open in new tab

Make sure the server is also running, and enter your name. You should see something like this:

[478] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/switch



Waiting for another player

You can connect to your game in another browser tab, and enter another name. Then you should see both names come up, and the status message change to allow a player to make a move. Of course, we haven't yet implemented the code to enable making a move from the UI, so let's do that now.

Handling player moves

We want a player to be able to click on a block to place their move. Kaboom has a function `onMouseRelease`[479] that we can use to handle mouse click events. All we need then is the position the mouse cursor is at, and we can map that to one of the board positions using our `boardSquares` array to do the lookup. We'll use the Kaboom function `mousePos`[480] to get the coordinates of the mouse:

[479]https://kaboomjs.com/doc/#onMouseRelease
[480]https://kaboomjs.com/doc/#mousePos

```
onMouseRelease(() => {
  const mpos = mousePos();
  // find the square we clicked on
  for (let index = 0; index < boardSquares.length; index++) {
    const square = boardSquares[index];
    if (mpos.x > square.x
        && mpos.x < square.x + square.width
        && mpos.y > square.y
        && mpos.y < square.y + square.height){
          socket.emit("action", {
            gridIndex: square.index
          });
          break;
        }
  }
});
```

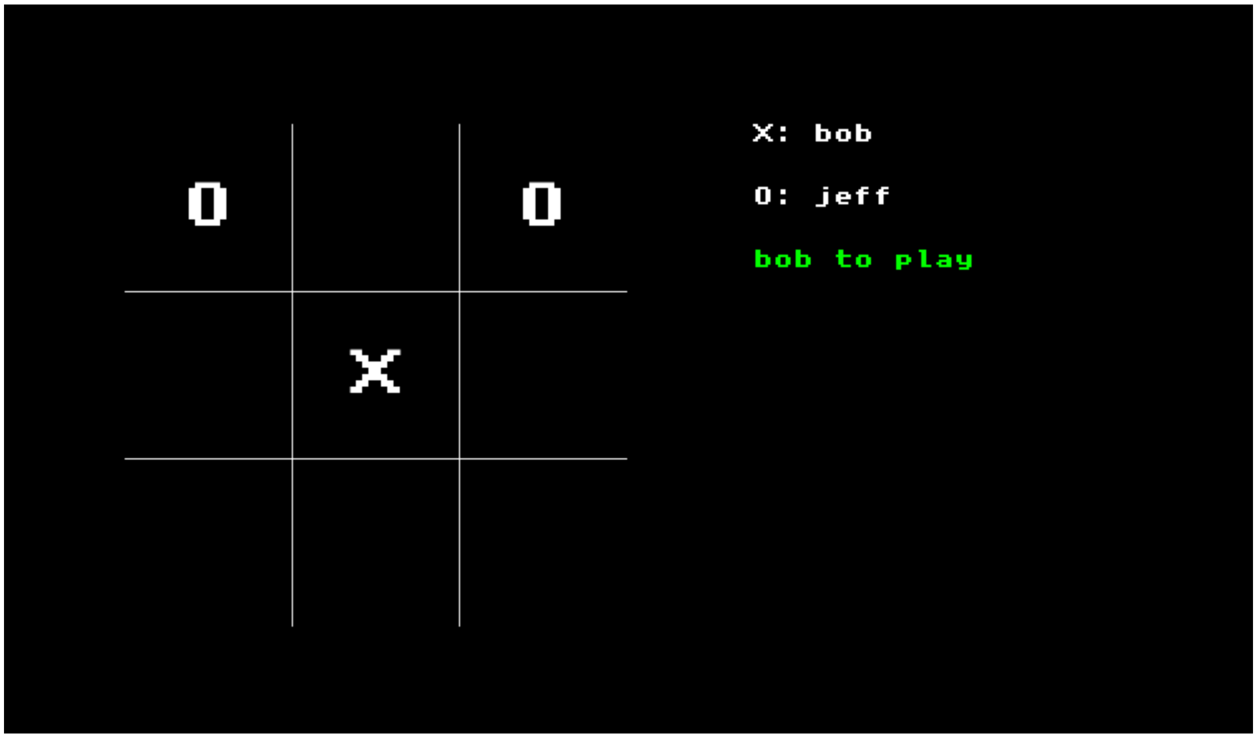
If we find a 'hit' on one of the board squares, we emit our `action` event. We pass the index of the square that was clicked on as the payload data. The server listens for this event, and runs the logic we added for the `action` event on the server side. If the action changes the game state, the server will send back the new game state, and the UI elements update.

The only other input we need to implement is to check if the player wants a rematch. To do that, we'll assign the `r` key as the rematch command. We can use the Kaboom function `charInput`[481] to listen for key press events. We'll check if the key is `r`, or `R`, then emit the `rematch` event. We don't have any data to pass with that, so we'll just pass `null`.

```
charInput((ch) => {
  if (ch === 'r' || ch === 'R'){
    socket.emit("rematch", null)
  }
});
```

Now you can run the game (and the server), and open the game in another tab, and you should be able to play tic-tac-toe against yourself! Send a link to the game to a friend, and see if they can join and play against you.

[481]https://kaboomjs.com/doc/#charInput



Playing tic tac toe

Next Steps

Now that you know the basics of creating a multiplayer online game, try your hand at making some different games, like checkers or chess or go.

Code

You can find the code for this tutorial on Replit[482]

[482]https://replit.com/@ritza/tic-tac-toe-new