CodeMonkey
LESSON PLANS

MOON LANDER

Game Creation Course:
Coding With Physics

Table of Contents

Topic	Page
Introduction	4
Before We Begin	5
Coding in CoffeeScript	6
Preliminary Coding Knowledge	7
Lesson 1 Objectives	8
Challenge 1	9
Challenge 2	11
Challenge 3	13
Challenge 4	15
Challenge 5	17
Challenge 6	19
Challenge 7	21
Challenge 8	22

Table of Contents

Topic	Page
Lesson 2 Objectives	25
Challenge 9	26
Challenge 10	29
Challenge 11	31
Challenge 12	33
Challenge 13	36
Challenge 14	39
Challenge 15	41
Challenge 16	43
Challenge 17	46
Reference Card	47
Sprite Review	50

Introduction

Thank you for choosing **Moon Lander Game Builder** to provide your students with a fun and innovative way to learn how physics simulation is used in game programming.

Spaceships approaching the moon (or other celestial objects) are attracted or pulled towards it due to the force of gravity. To land safely, the spaceship must use its engines and burn fuel to produce thrust, a force that pushes it upwards. The right balance between the forces will result in a safe landing. In the upcoming challenges students will learn how these ideas can be simulate using CodeMonkey's Game Builder platform.

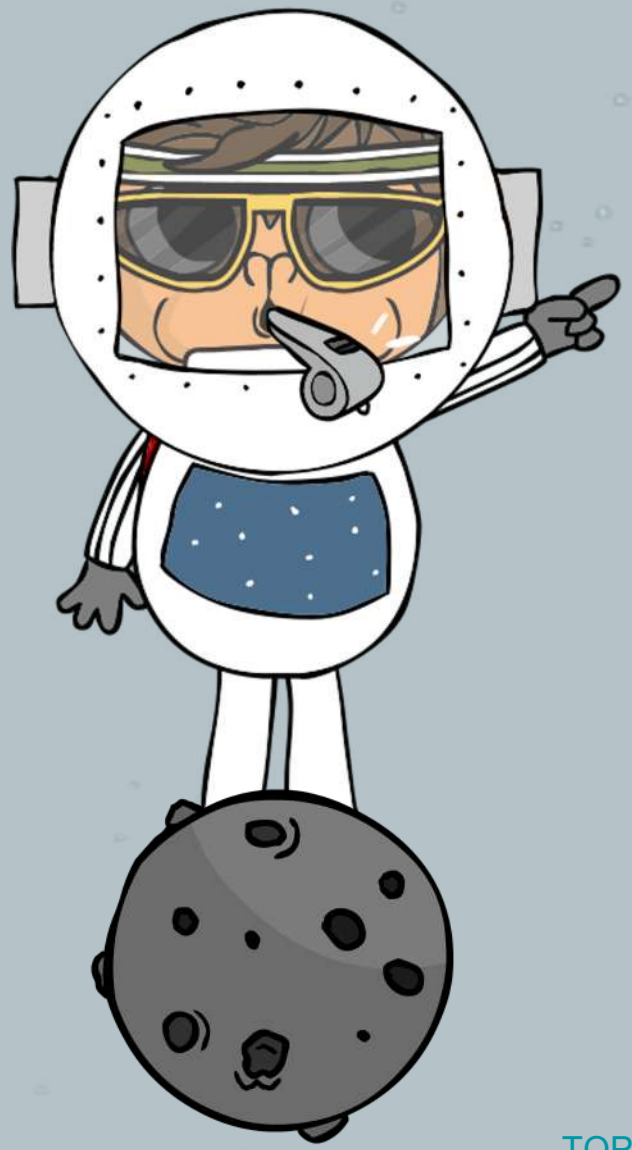
The following lesson plans will guide you on how to successfully integrate **Moon Lander Game Builder** into your class.

Before We Begin...

Click here to access a beginner's guide that will help you get started with creating accounts for your students and managing your classroom. Should you have any questions, you can contact us anytime at: **info@codemonkey.com.**

Have fun!

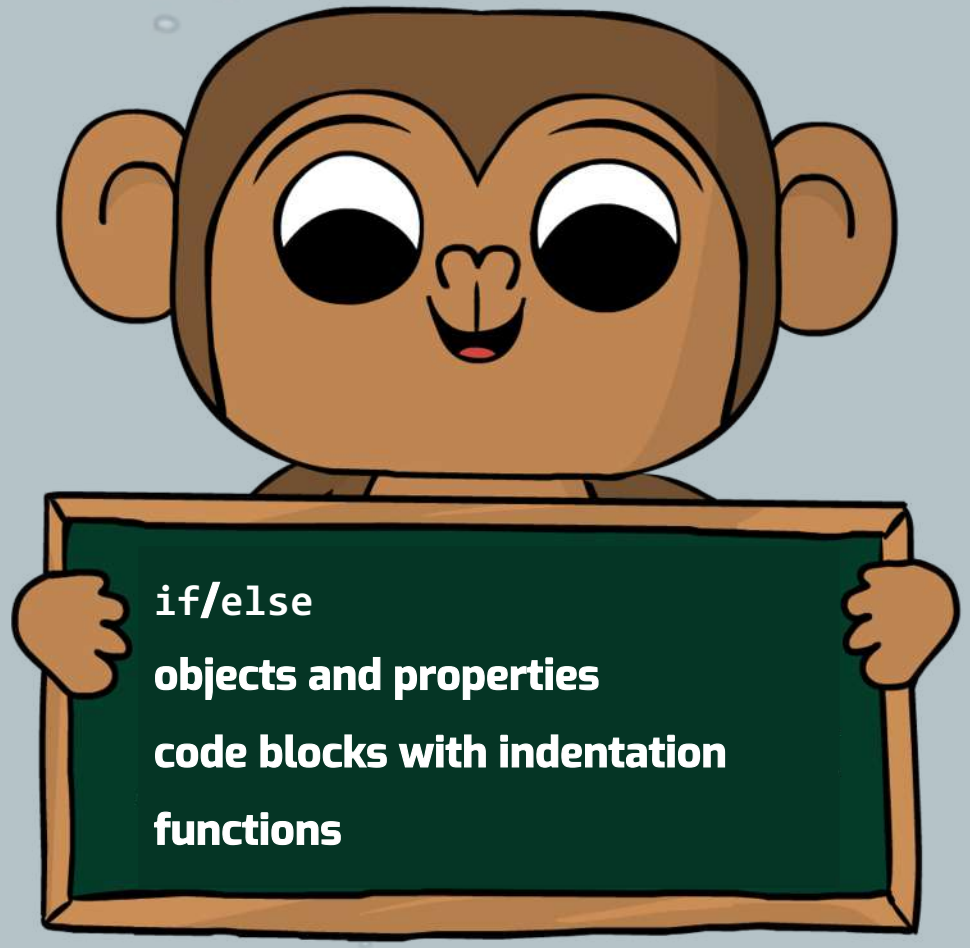
The CodeMonkey Team



Coding in CoffeeScript

CoffeeScript is the programming language taught in **Moon Lander Game Builder** requires basic knowledge of CoffeeScript in every challenge. The language compiles to JavaScript. Similarly to JavaScript, it is used in the industry primarily for web applications. The language was chosen mainly because of its friendly syntax, which resembles written English.

Preliminary Coding Knowledge



Lesson 1 – Navigate in Space with Moon Lander Game Builder

In this lesson, students will:

- Learn the basics about the forces of gravity and thrust
- Solve challenges **1 - 8**
- Modify game parameters
- Navigate the spaceship in space and make their first landing

Challenge 1

Goal: Students will launch the spaceship and reach the yellow star, using any key on the keyboard

Means: Students need to apply the force of thrust when pressing any keyboard key, to make the spaceship launch and overcome the force of gravity.

New coding tasks \ concepts:

- **sprite**: A sprite is visual object (“bitmap”) placed on the screen, which can have its own code. In our case the spaceship is a sprite and the star is a sprite.
- **`onKey()`**: This is an example of an event handler. The `onKey()` function is called whenever the player presses a key on the keyboard.
- **`thrust()`:** The `thrust()` function applies a force to the “bottom” of the spaceship. This usually causes the spaceship to move, unless other forces (like gravity) or obstacles prevent it. The `thrust()` function takes a number that represents the strength of the force as an argument.

Challenge 1 - Solution

Your students will add the code in red:

spaceship

```
1.  @onKey = (key) =>
2.      # Use any key to make the spaceship go
    up
3.      # Add your code here
4.      @thrust 130
```

TIP:

Remind your students that when they call a function in the code window of a sprite:

- If the function should act on the **same sprite**, then the symbol **@** should be used before the function name.
- If the function should act on a **different sprite**, then the sprite's name should be written, followed by a dot (e.g.. `spaceship.thrust`)

Challenge 2

Goal: Students will launch the spaceship and reach the yellow star, using the up key on the keyboard

Means: Students need to apply the force of thrust when pressing the up arrow key, to make the spaceship launch and overcome the force of gravity.

New coding tasks \ concepts:

- **`if` *`condition`***: The indented code underneath executes if and only if the *condition* is TRUE.
- **`key == keyboard.up`**: The variable `key` is an argument to the event handler function `onKey()` based on which keyboard key was pressed. This condition compares the value of `key` with the constant `keyboard.up`, which represents the up arrow key on the keyboard.

TOP

Challenge 2 - Solution

Your students will add the code in red:

spaceship

```
1.  @onKey = (key) =>
2.      # Use the up key to make the spaceship go up
3.          if key == keyboard.up
4.          # Add your code here
5.              @thrust 130
```

TOP

Challenge 3

Goal: Students will navigate the spaceship to reach each of the two yellow stars

Means: Students need to rotate the spaceship to the left (using the left arrow key) and to the right (using the right arrow key) , in addition to applying the thrust force to move forwards.

New coding tasks \ concepts:

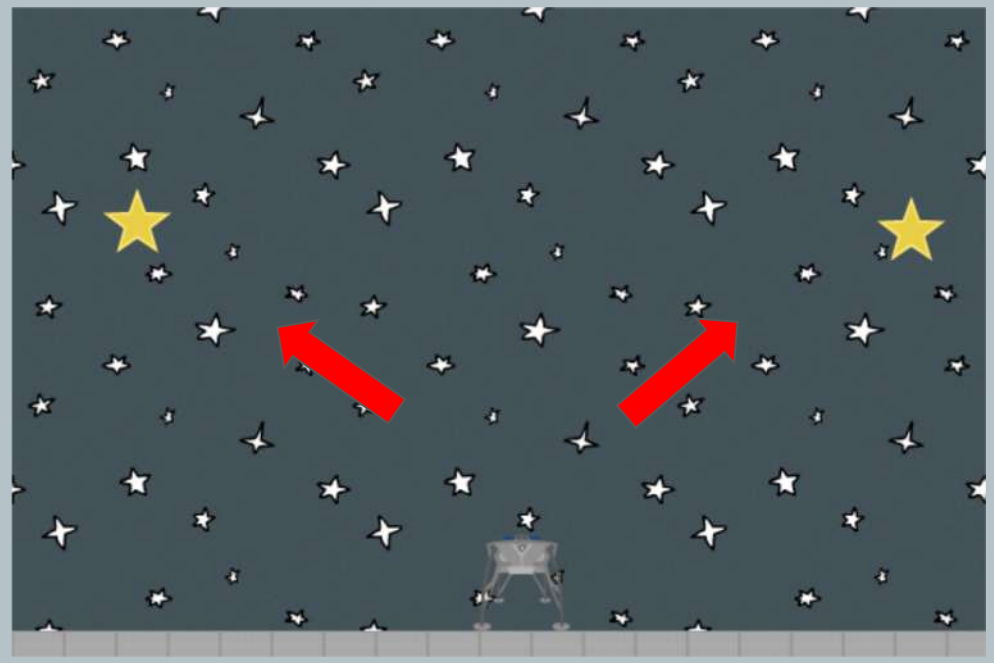
- **`if key == keyboard.right`**: This if statement compare the value of the argument key with the constant keyboard.right, which represents the right arrow key. If the key pressed was the right arrow, the indented code below is executed.
- **`if key == keyboard.left`**: This if statement is similar, but checks for the left arrow key
- **`rotateLeft()`**, **`rotateRight()`***:* These two functions make the sprite spin in the corresponding directions. The argument represents how much force is used to make the sprite spin. Larger values cause the sprite's spinning to accelerate faster.

Challenge 3 - Solution

Your students will add the code in red:

spaceship

```
1.  @onKey = (key) =>
2.      if key == keyboard.up
3.              @thrust 130
4.          if key == keyboard.right
5.              @rotateRight 5
6.          # Add the possibility to rotate the
    spaceship
7.          # to the left using the Left arrow key
8.          # Add your code here
9.          if key == keyboard.left
10.             @rotateLeft 5
```



Challenge 4

Goal: Students will modify some of the game's parameters and reach a distant, initially unreachable star.

Means: Students need to change the **background image**, the **world width** parameter and the **camera target** parameter, and then fly the spaceship to the 3rd star.

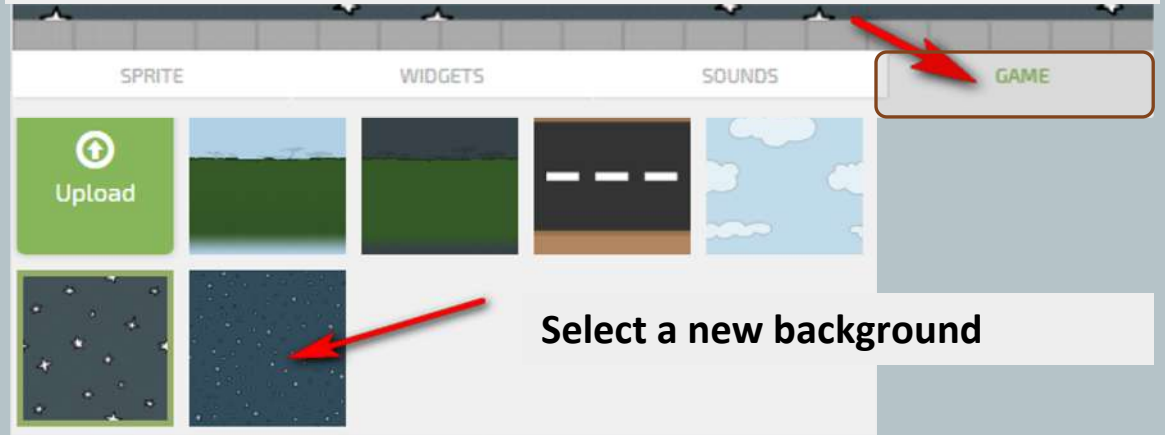
New coding tasks \ concepts:

Students will modify some of the game's parameters, including:

- **Background image**: Chosen from a set of available images
- **World width**: Set in pixels
- **Camera target**: Allows us to designate a sprite to be followed by the game's virtual camera so it stays on screen even when the game world scrolls

Challenge 4 - Solution

In the GAME tab:



Select a new background

Modify the World height and Camera target

TIP:

- Sprites may be located outside the boundaries of the game. In such cases they cannot be seen and cannot be reached by other sprites. For example, in this challenge there is a star located in an X value bigger than the initial World width, and the spaceship can only fly till it reaches the World width. Only when your students change the **World width** parameter, they can have the spaceship go there and reach the star.
- Changing the **Camera target** to "spaceship" enables the player to follow the spaceship wherever it goes within the game's boundaries.

Challenge 5

Goal: Students will increase the game's height, will add a new star and navigate towards it.

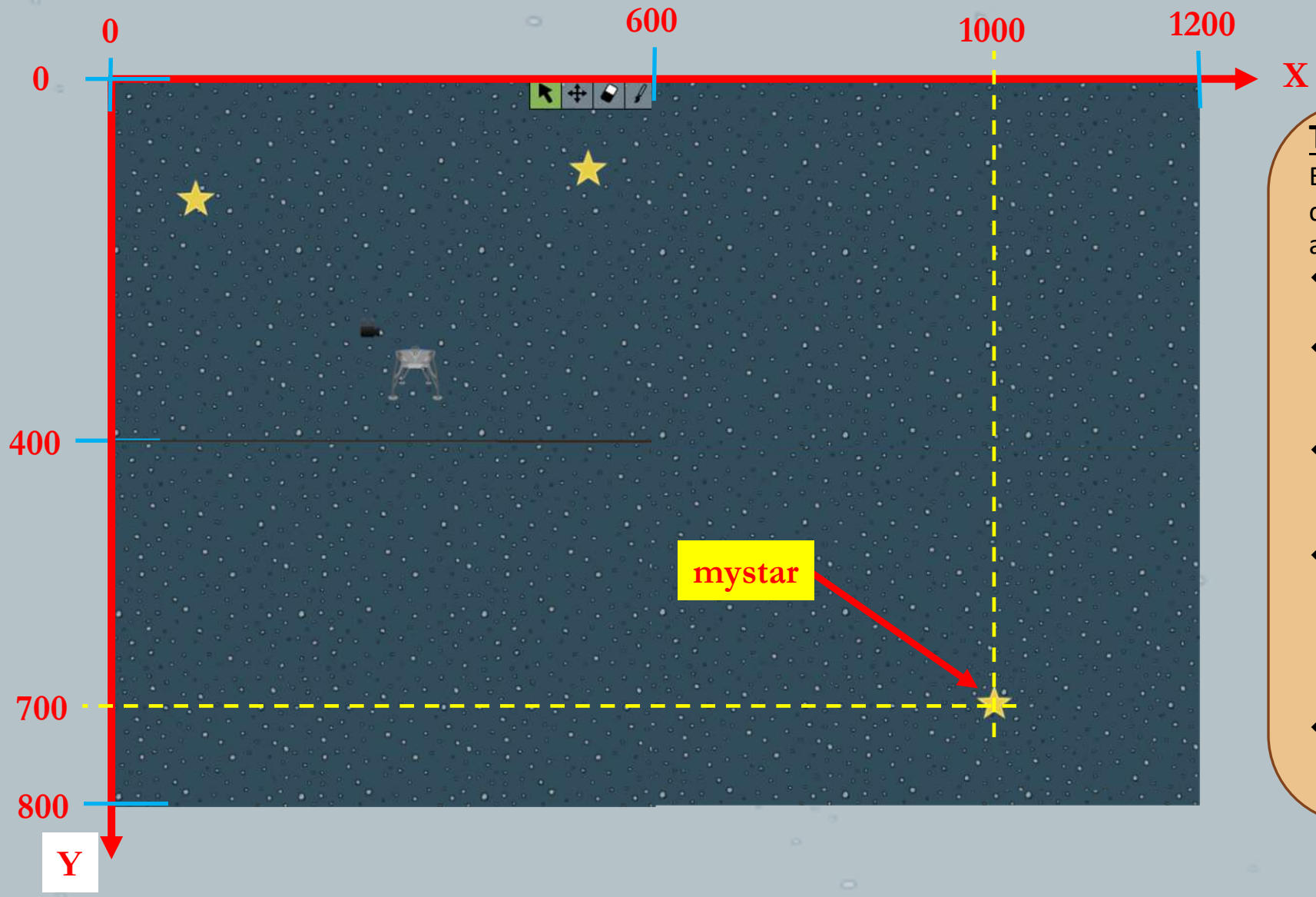
Means: Students will change another GAME parameter and will learn how to add a new sprite.

New coding tasks \ concepts:

- **World height**: The **world height** parameter can be found on the GAME tab.
- **Adding sprites**: New sprites can be added to the world using the SPRITES tab.
  - Add a new sprite with the plus green button.
  - Change its name to "mystar"
  - Uncheck the **Allow Gravity** option, so that the star will not be affected by the gravity force in the game.
  - Check the **Immovable** option, so that the star will not move when the spaceship runs into it.
  - Set its location to coordinates to **X=1000, Y=700**

TOP

Challenge 5 - Solution



TIP:

Explain to your students the coordinates system in the game (and actually in any digital screen):

- The top left corner is the origin (X=0, Y=0).
- Going to the right increases the **X** coordinate (up to the **world width**)
- Going down increases the **Y** coordinate (up to the **world height**)
- Each of the sprite (the spaceship, star1, star2, star3 and the new mystar) has X and Y coordinates which determine its location on the game screen.
- Gravity in the game pulls the sprites down, meaning that their Y coordinate increases.

Challenge 6

Goal: Students will try to launch the spaceship from a “heavy” planet (with greater gravity).

Means: Students need to increase the thrust force they apply on the spaceship, to be able to launch.

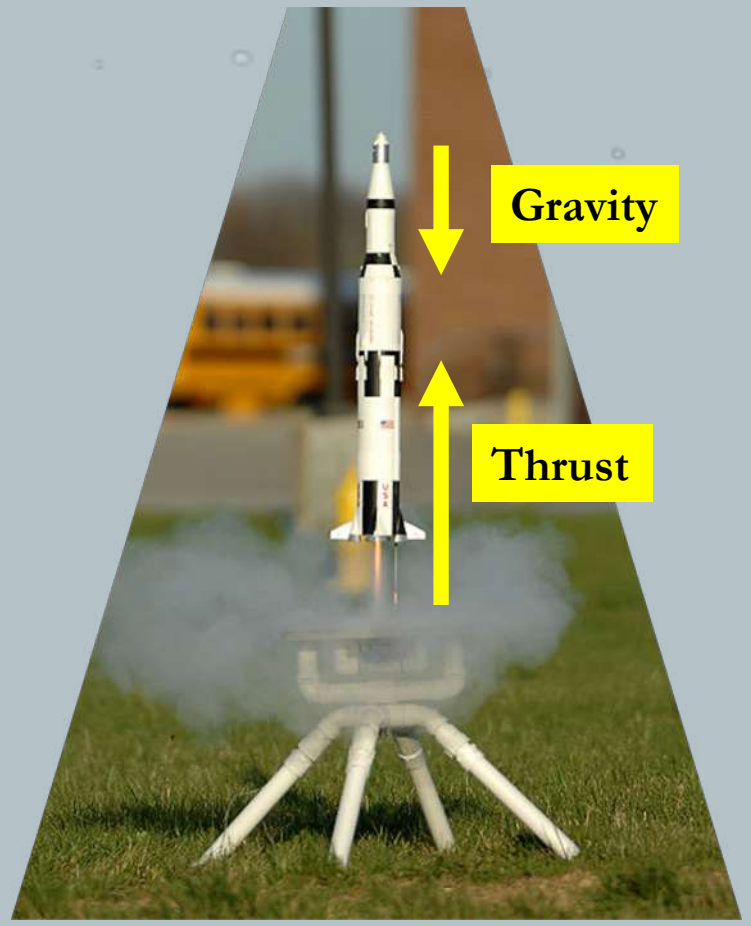
New coding tasks \ concepts:

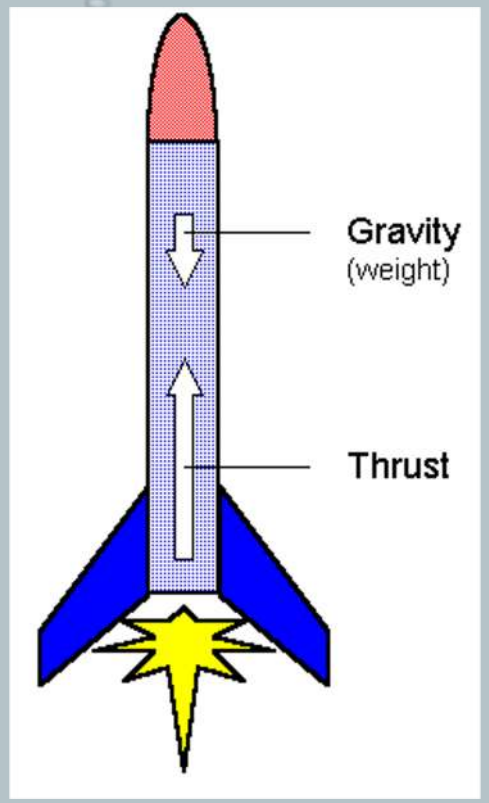
- **Gravity**: Before beginning this challenge, have students note the value of **Gravity** parameter on the GAME tab.

- `thrust()` argument: The argument of the `thrust()` function represents how much force the spaceship’s rockets are providing. The thrst parameter must be greater than gravity for the ship to get off the ground, but students should experiment to discover that for themselves.

TIP: Encourage your students to try different values of gravity and check what happens with different values of thrust!

Challenge 6 - Solution





PHYSICS REVIEW:

- The force of gravity and the spaceship's thrust force pull the ship in opposite directions. Whichever is greater "wins".
- Even when the engines are off, the spaceship doesn't fall off the bottom on the screen because the "ground" supplies the upward force that opposes gravity.
- Newton's law of gravitation says that gravitational force is proportional to an object's mass. Changing the gravity parameter of the game essentially changes the mass of the planet that the spaceship is on.

https://commons.wikimedia.org/wiki/File:JMS_0067Crop.jpg
https://space.stackexchange.com/questions/12889/how-does-the-roll-maneuver-allow-more-mass-to-be-lifted-into-orbit

Challenge 7

Goal: Students will add a landing surface and will navigate to land on it.

Means: Students will once more add a new sprite.

Concept review:

Adding sprites: New sprites can be added to the world using the SPRITES tab.

- Add a new sprite with the plus green button.
- Change its name to “surface”.
- Uncheck the **Allow Gravity** option, so that the star will not be affected by the gravity force in the game.
- Check the **Immovable** option, so that the star will not move when the spaceship runs into it.
- Set its location to coordinates to **X=200, Y=380**.

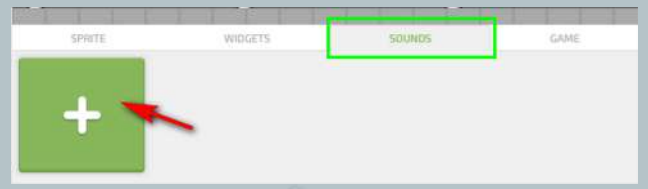
Challenge 8

Goal: Students will define what happens when the spaceship collide with the landing surface.

Means: Students will add a new sound to play on landing.

New coding tasks \ concepts:

- **Sounds:** Sounds can be added by clicking the plus green button on the SOUNDS tab.



The new sound should be named “win”

Sound objects have a **play()** method that can used to play them in code, e.g.. **win.play()**

Challenge 8 (continued)

New coding tasks \ concepts:

- **Collision handlers**: A collision handler is a function that is called when one sprite collides with another. The syntax for defining a collision handlers looks like this:

```
@onCollide spaceship, () =>
    # Collision handler code
```

The first argument to the `onCollide()` function is the name of the "other" sprite to check for a collision with, and the second argument defines the collision handler function itself.

Challenge 8 - Solution

Your students will add the code in red:


```
1.  @onCollide  spaceship, () =>
2.      win.play()
```

Lesson 2 – Land Safely, check velocity and fuel

In this lesson, students will:

- Check their spaceship velocity and try to land gently
- Implement a fuel system that limits how long thrust can be applied
- Add obstacles
- Solve challenges **9 - 18**

Challenge 9

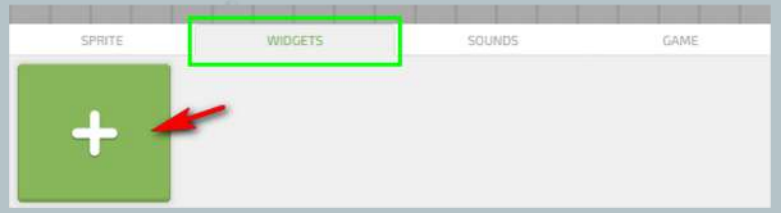
Goal: Students will display the spaceship's velocity in Y axis

Means: Students will add a new text widget and will use a function to get the spaceship's velocity.

New coding tasks \ concepts:

- **Text widget:** Text widget can be used to display text on the screen. The text displayed by a text widget can be changed by setting the widget's `text` property.

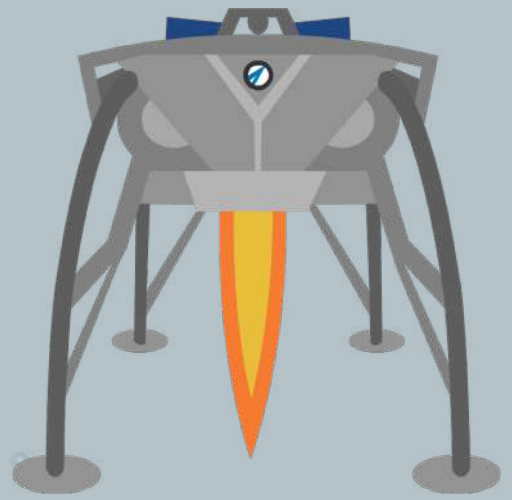
  In this challenge there is already a text widget called `textX`. Students will need to add a second widget and name it `textY`. This can be done from the WIDGETS tab.



Challenge 9 (continued)

New coding tasks \ concepts:

- **`onUpdate()`**: Every sprite has a property called onUpdate. If that sprite is set equal to a function, that function will be called every time the game redraws the screen, usually 50-60 times per second. An **`onUpdate()`** function is the perfect place to put code to update a text widget.
- **`getVelocityX()`, `getVelocityY()`**: These functions return the velocity of the sprite in the horizontal and vertical directions respective. In this challenge the students will use them to update the `text` property of the text widgets.



Challenge 9 - Solution

Your students will add the code in red:

spaceship

```
1.  @onKey = (key) =>
2.      if key == keyboard.up
3.              @thrust 130
4.          if key == keyboard.right
5.              @rotateRight 5
6.      if key == keyboard.left
7.          @rotateLeft 5

8.  @onUpdate = () =>
9.      # Add your code here to display the velocity on y axis
10.         textY.text = @getVelocityY()
```

TIP:

- The velocity is measured in pixels per second. It is displayed as a floating point number, with 14 decimal digits.
- In order to display it "nicer" we will convert it into a "string" in the next challenges and the red line code will change to:
  textX.text = "velocity.y = #{@getVelocityY().toFixed(4)}", which will only display 4 decimal digits.

Challenge 10

Goal: Students will try to land the spaceship safely (Y velocity < 50 pixels per second) on the surface.

Means: Students will add a condition to determine is the landing is successful or not.

New coding tasks \ concepts:

- **Safe landings**: This challenge starts to introduce an element of difficulty to the game. To "win", the player must not only land the spaceship on the pad but do so safely. In this context, "safely" means that the vertical velocity of the ship is less than 50. This can be checked with an `if` statement using the condition:

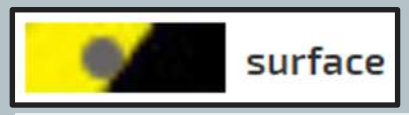
  ```
  spaceship.getVelocityY() < 50
  ```

  Note that it is `spaceship.getVelocityY()` not `@getVelocityY()` because this code will be associated with the surface sprite, not the spaceship.

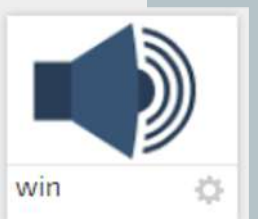
- **`game.reset()`**: This function does exactly what it says - resets the game to its initial state, just as if we had just clicked RUN!.

Challenge 10 - solution

Your students will add the code in red:




```
1.  @onCollide spaceship, () =>
2.      win.play()
3.      if spaceship.getVelocityY() < 50
4.          # Add your SUCCESS code here
5.              win.play()
6.              game.reset()
```



TIP:

- Remind your students that when running the game, they need to apply the thrust force (using the up key) to keep the spaceship's velocity low enough to land safely! (watch the velocity updating on screen and keep it under 50).
- After the game is reset, your students can play and try to land safely again and again.

Challenge 11

Goal: Students will determine what happens if the spaceship does not land safely on the surface.

Means: Students will add a “else” statement after the “if” condition.

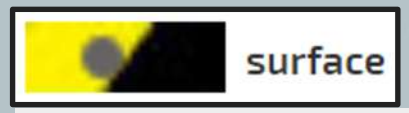
New coding tasks \ concepts:

- **Review**: To complete this challenge students will have to add a new sound and name it “crash”.
- **Failure is an option**: Most games include the possibility of failure. In the last challenge the players was given a way to win but not a way to lose.

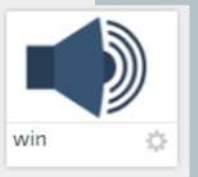
  The losing condition can be created by adding an `else` statement to the `if` from the last challenge. The code associated with the `else` will be executed if the ship’s vertical velocity is *not* less than 50 when it lands
- **`destroy()`**: The `destroy()` function removes a sprite from the game. The ship will be destroyed if it lands too fast. The function must be called as `spaceship.destroy()` since the call is inside the code for the surface sprite.

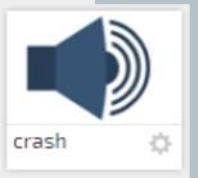
Challenge 11 - solution

Your students will add the code in red:




```
1.  @onCollide spaceship, () =>
2.      if spaceship.getVelocityY() < 50
3.          win.play()
4.              game.reset()
5.      else
6.              # Add your FAIL code here
7.              crash.play()
8.              spaceship.destroy()
```





TIP Reminder:

Remind your students that when they call a function in the code window of a sprite:

- If the function should act on the **same sprite**, then the sign **@** should be used before the function name.
- If the function should act on a **different sprite**, then the sprite's name should be written
- In this challenge we wrote the code in the **surface** sprite. Therefore, we used: **`spaceship.destroy()`**
- If we would have written the code in the **spaceship** code are, it would have been: **`@destroy()`**

Challenge 12

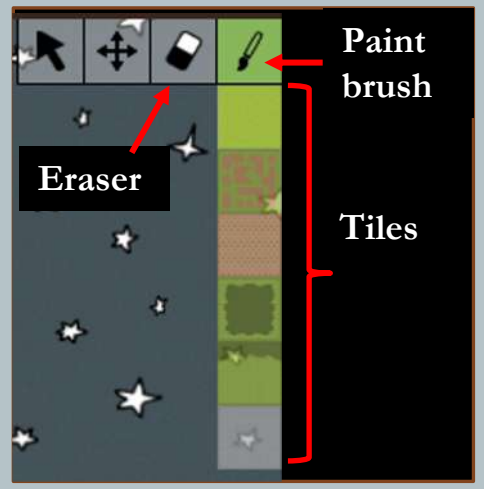
Goal: Students will add “tiles” to the screen and will try to avoid them before landing.

Means: Students will add a new function, which causes the spaceship to be destroyed when colliding with tiles.

New coding tasks \ concepts:

- **Tiles**: There is another kind of visual element that can be added to the screen besides sprites: tiles. Tiles are less dynamic than sprites. They are part of the game world but do not move. Collectively, all the tiles that are part of the game are called the tilemap.

  The tool bar in the upper right corner of the screen can be used to draw and and erase cards from the tilemap.

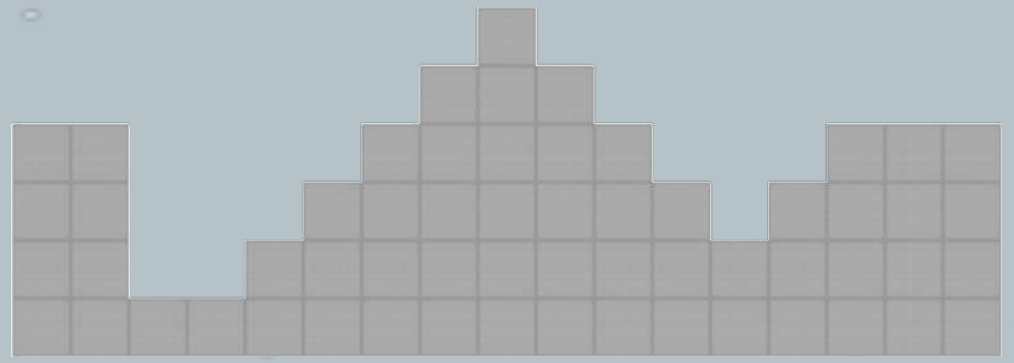


Challenge 12 (continued)

New coding tasks \ concepts:

- **`onCollideWithTilemap()`**: Since tiles are not sprites the `onCollide()` function can't be used to set up a handler for collisions with the tilemap. Instead, there this the `onCollideWithTilemap` property. When a function is assigned to this property it will be called when the sprite in question (the spaceship in this case) collides with any tile.

  For this challenge this `onCollideWithTilemap()` function should destroy the sprite and play the crash sound, just as with a too-fast landing. Since the code for `onCollideWithTilemap()` is part of the spaceship sprite, `destroy()` can be called with `@destroy()` rather than `spaceship.destroy()`.



Challenge 12 - Solution

Your students will add the code in red:


```
1.  @onKey = (key) =>
2.      if key == keyboard.up
3.                @thrust 130
4.           if key == keyboard.right
5.           @rotateRight 5
6.      if key == keyboard.left
7.           @rotateLeft 5
8.
9.  @onUpdate = () =>
10.      textY.text = "Y_velocity = #{@getVelocityY().toFixed(4)}"

11. @onCollideWithTilemap = () =>
12.     # Add your code here
13.     crash.play()
14.     @destroy()
```

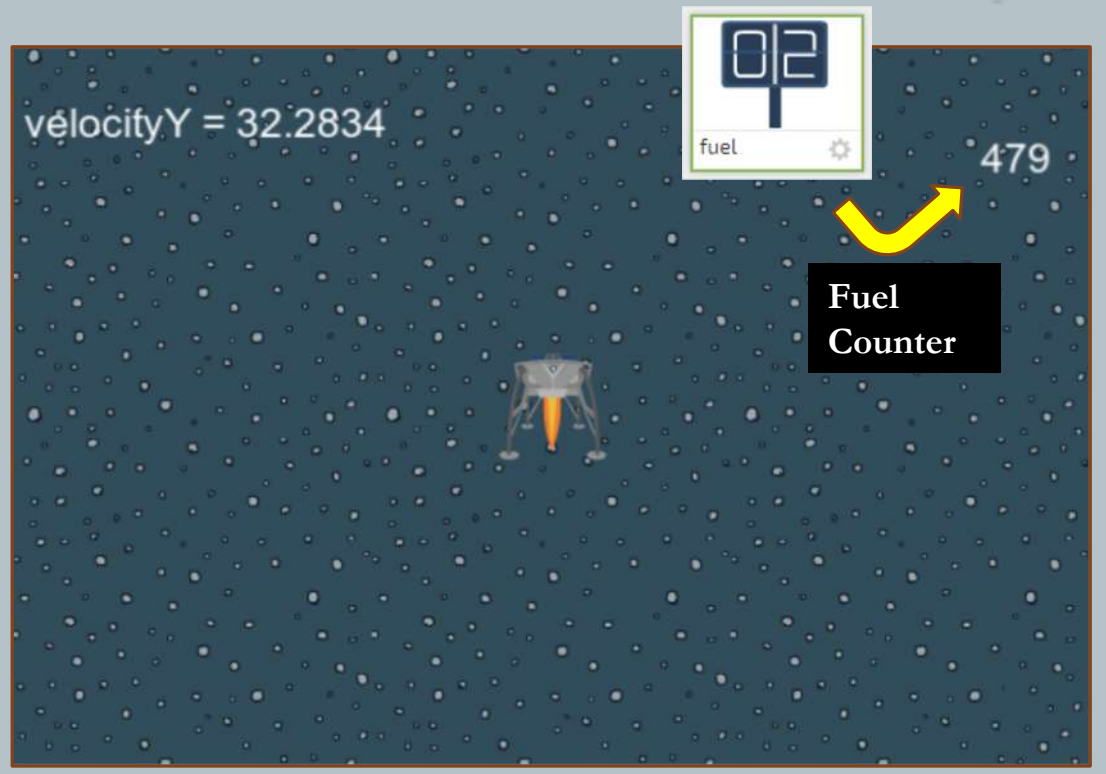
TIPS:

- When using the paint brush, your students can actually “paint” tiles on the screen continuously by pressing the mouse left click and keeping it pressed while moving on the screen. Same with the eraser.

Challenge 13

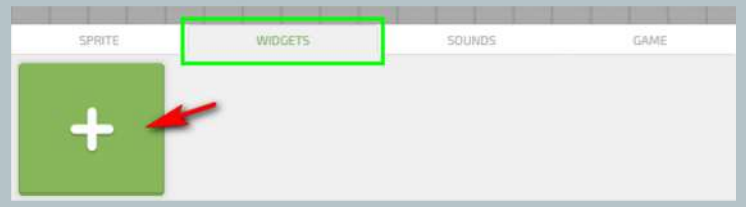
Goal: Students will add a counter to display the spaceship's fuel level.

Means: Students will add a new counter and reduce its value every time a thrust force is applied on the spaceship.



New coding tasks \ concepts:

- **Counter widgets**: A counter widget is similar to a text widget but is used to display a numerical value. Counter widgets can be added to the game from the WIDGETS panel. In this challenge students will create a counter called fuel to track the spaceship's fuel supply.



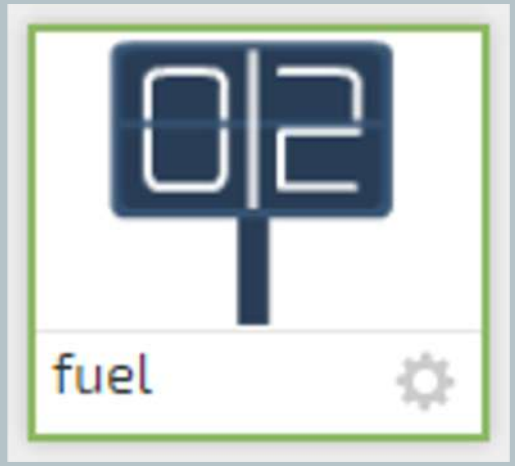
Counter widgets have a property called value. The initial value of a counter can be set in its own edit window using a statement like `@value = 500`.

Challenge 13 (continued)

New coding tasks \ concepts:

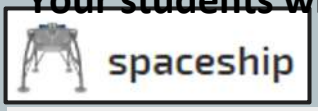
- `-=`: The counter in this challenge is intended to represent the spaceship's fuel supply. The spaceship should burn fuel each time thrust is applied. That means the code associated with `if key == keyboard.up` in the spaceship sprite needs to be changed so that it also decreases the `value` property of the fuel counter.

  The statement `fuel.value -= 1` does just this. It means subtract 1 from the current fuel.value and make that the new fuel.value. It is exactly equivalent to writing `fuel.value = fuel.value - 1`. The `-=` operator is just a shortcut provided by the Coffeescript language.



Challenge 13 - Solution

Your students will add the code in red:




```
1.  @onKey = (key) =>
2.      if key == keyboard.up
3.          @thrust 130
4.                      # Reduce fuel counter by 1
5.                      fuel.value -= 1
6.      if key == keyboard.right
7.          @rotateRight 5
8.      if key == keyboard.left
9.          @rotateLeft 5

10. @onUpdate = () =>
11.         textY.text = "Y_velocity = #{@getVelocityY().toFixed(4)}"

12. @onCollideWithTilemap = () =>
13.         crash.play()
14.         @destroy()
```

TIPS:

- Tell your students to watch the fuel counter as they fly the spaceship and apply thrust. The fuel counter may become negative (smaller than zero), because we subtract from fuel.value 1 each time they press the up key, without checking if there is still fuel left. This will be corrected in the next challenge.

Challenge 14

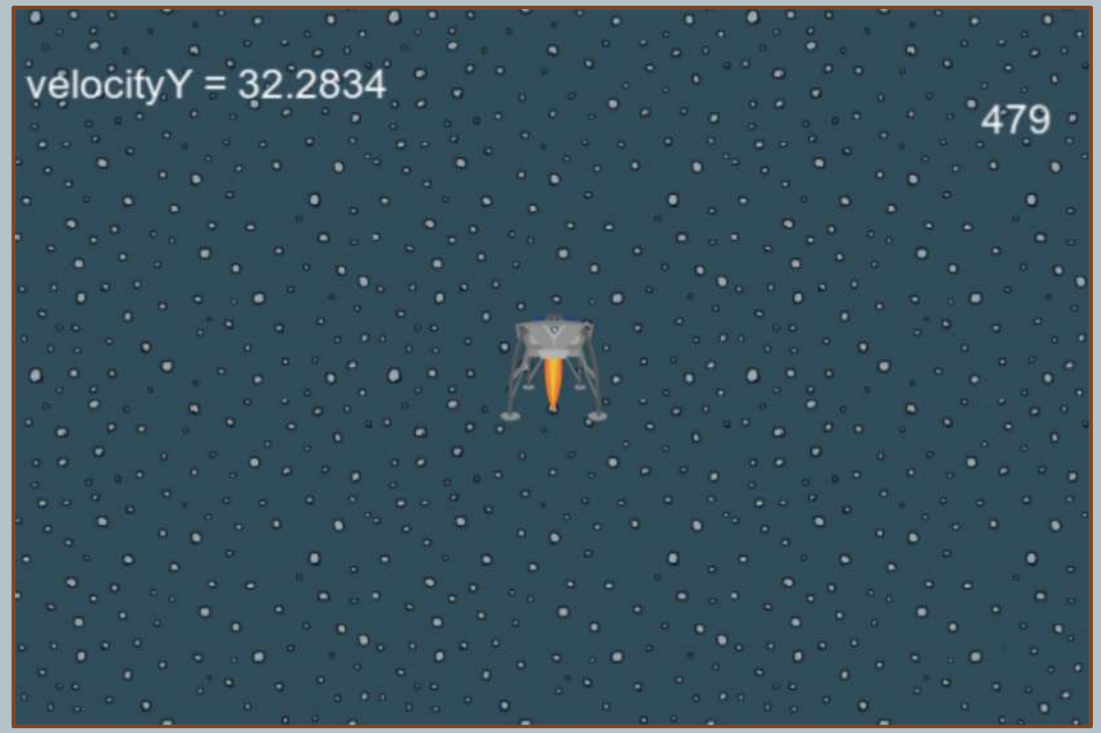
Goal: Students will make sure they land with enough fuel left in your spaceship.

Means: Students will add a condition that checks if you still have fuel left. If not- you cannot apply thrust force!

New coding tasks \ concepts:

- **Fuel matters**: While the last challenges added the concept of burning fuel when the spaceship uses its thrust function, the fuel level did not actually matter. All that happened when the ship ran out of fuel is the counter went negative.

  In this challenge we make things more realistic by making sure there is fuel before we allow the spaceship to thrust. We do this by adding an `if fuel.value > 0` statement inside the spaceship's `onKey()` function. This will make sure that we only provide thrust and subtract fuel if there actually is still fuel to burn.



Challenge 14 - Solution

Your students will add the code in red:


```
1.  @onKey = (key) =>
2.      if key == keyboard.up
3.          @thrust 130
4.          fuel.value -= 1
5.          if fuel.value > 0
6.              @thrust 130
7.              fuel.value -= 1
8.      if key == keyboard.right
9.          @rotateRight 5
10.     if key == keyboard.left
11.         @rotateLeft 5
12.
13. @onUpdate = () =>
14.     textY.text = "Y_velocity = #{@getVelocityY().toFixed(4)}"

15. @onCollideWithTilemap = () =>
16.     crash.play()
17.     @destroy()
```

Challenge 15

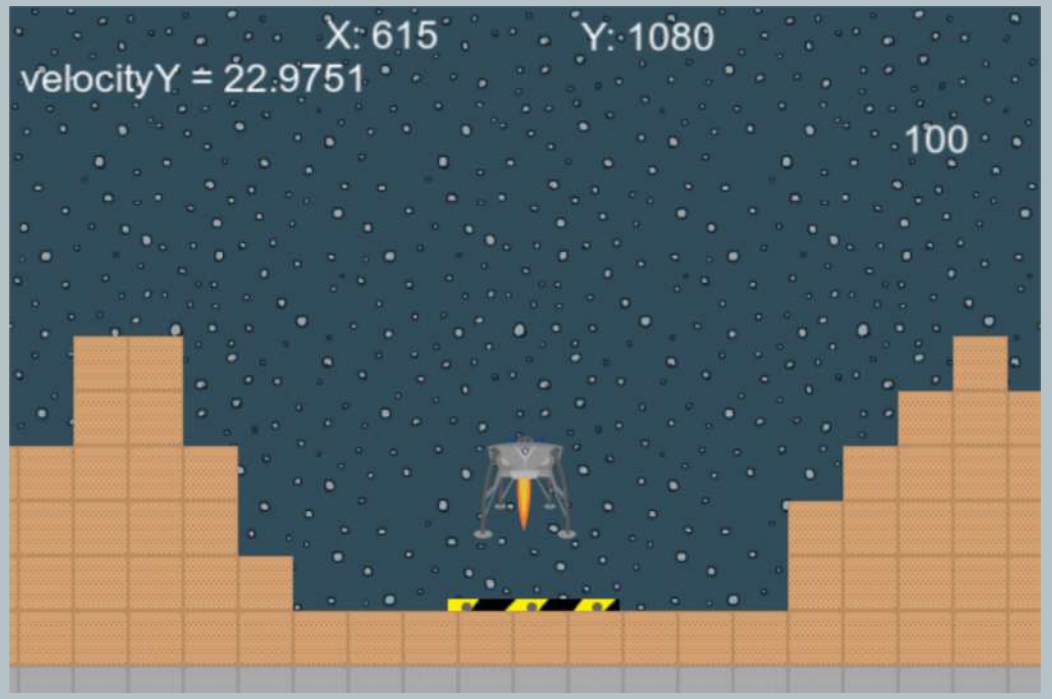
Goal: Students improve and organize the existing code for Moon Lander.

Means: Students will add a new function to encompass the tasks involved with providing thrust.

New coding tasks \ concepts:

- **Preliminaries**: To complete this challenge students will have to add a new sound and name it “force”. They also will need to go to the code editor for the fuel counter widget and change the initial value to 1000.
- **`burnFuel()`**: The main task of this challenge is to move the code from the `if key == keyboard.up` section of `onKey()` event handler into a new function called `burnFuel()` and replace that code with a call to `burnFuel()`. Grouping the tasks related to applying thrust (checking the fuel level, applying thrust, decreasing the fuel level) into their own function makes the code easier to read and understand.

  Students will also need to add a statement to the `burnFuel()` function to play the `force` sound.



Challenge 15 - Solution

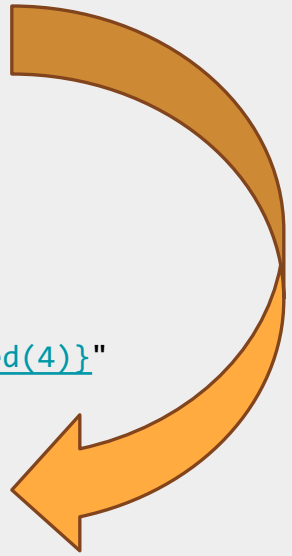
Your students will add the code in red:

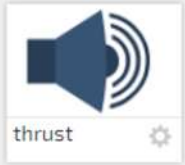
spaceship

```
1.  @onKey = (key) =>
2.      if key == keyboard.up
3.          @burnFuel()
4.          if fuel.value > 0
5.              @thrust 130
6.              fuel.value -= 1
7.      if key == keyboard.right
8.          @rotateRight 5
9.      if key == keyboard.left
10.         @rotateLeft 5
11.
12. @onUpdate = () =>
13.     textY.text = "Y_velocity = #{@getVelocityY().toFixed(4)}"

14. @onCollideWithTilemap = () =>
15.     crash.play()
16.     @destroy()

17. @burnFuel = () =>
18.     if fuel.value > 0
19.         @thrust 130
20.         fuel.value -= 1
21.         thrust.play()
```





TIPS:

- The sound that plays when the spaceship thrusts probably won't sound much like the sound the students' picked. That's because the sound keeps restarting for as long as the up arrow key is held down.
- Landing the ship safely is actually pretty challenging at this point. If students are having trouble, one way to "cheat" and make the game easier is to lower the **Gravity** parameter in the GAME tab.

Challenge 16

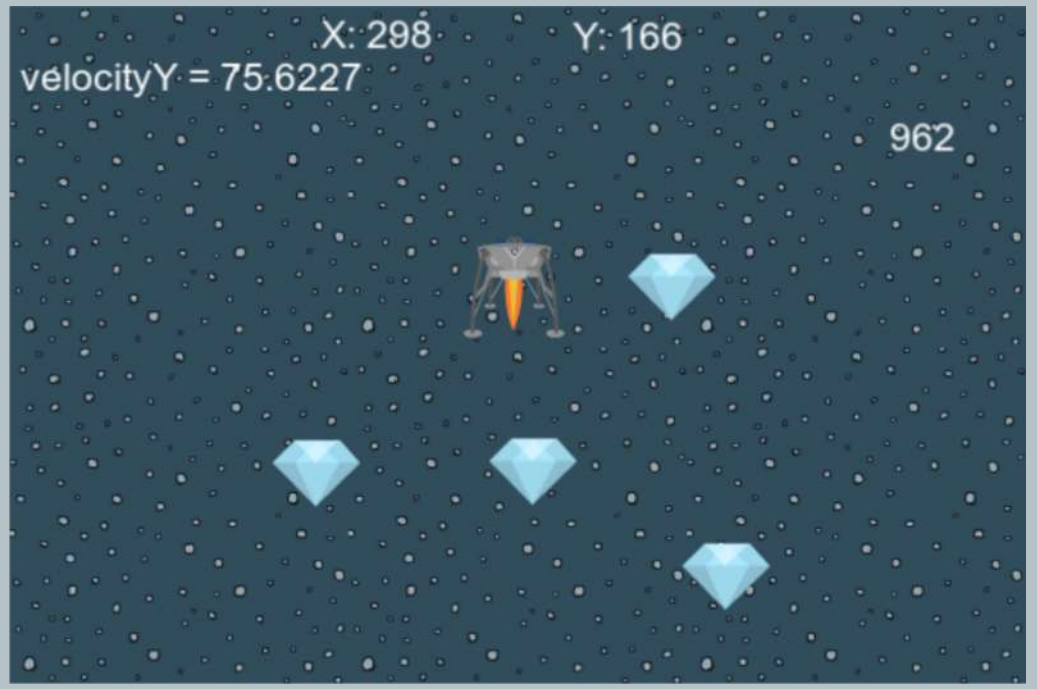
Goal: Students will collect diamonds on your their to landing to get more fuel.

Means: Students will add a new type of sprite, duplicate it, and write its code.

New coding tasks \ concepts:

- **Preliminaries**: To complete this challenge students will have to add a new sound and name it “collect”.
- **Diamond sprites:** Students will need to add a new type of sprite to their game. This sprite should have the name diamond and use the diamond image. Like the stars earlier in the course, the diamond sprite should have its **Allow Gravity** parameter turned off and **Immovable** parameter turned on.

  The code for the diamond sprite needs to call `onCollide()` to set up a handler for collisions with spaceship. That handler should destroy the diamond, play the collect sound, and 50 units to fuel.value (`fuel.value += 50`).

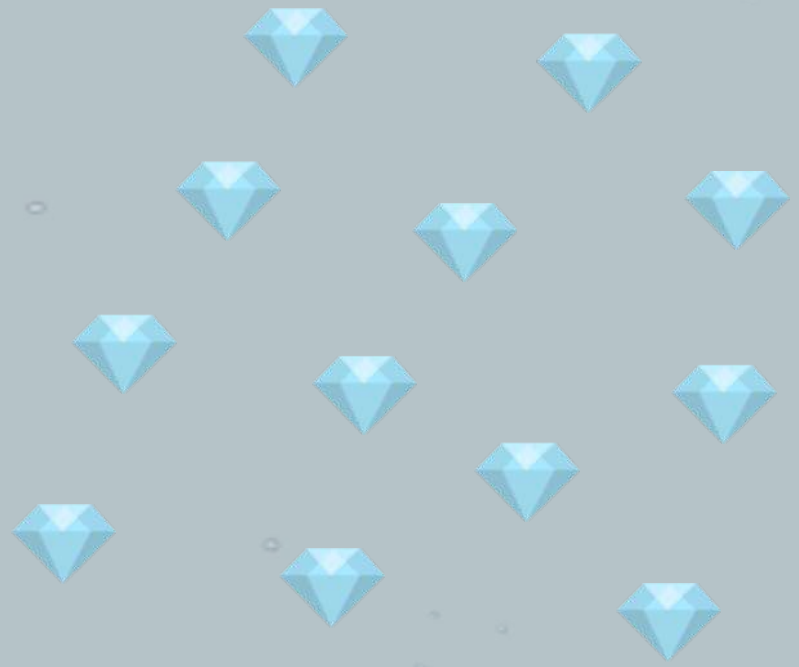


Challenge 16 (continued)

New coding tasks \ concepts:

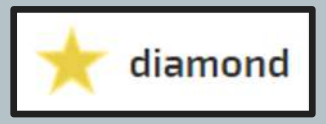
- **Duplicating sprites:** After the diamond sprite has been created and its code written, students will need to duplicate it several times. This can be done by first access the sprites properties ( ) from the SPRITES tab and then clicking on the Duplicate button.

  The copies of diamond are given new names like `diamond2`, `diamond3`, etc. Because of this, it is essential that the `destroy()` function be called as `@destroy()` inside the `onCollide()` handler. That way the specific diamond that was collided with will be the one destroyed.

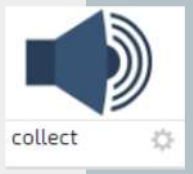


Challenge 16 - Solution

Your students will add the code in red:




```
1. @onCollide spaceship, () =>
2.     collect.play()
3.     @destroy()
4.     fuel.value += 50
```



Challenge 17

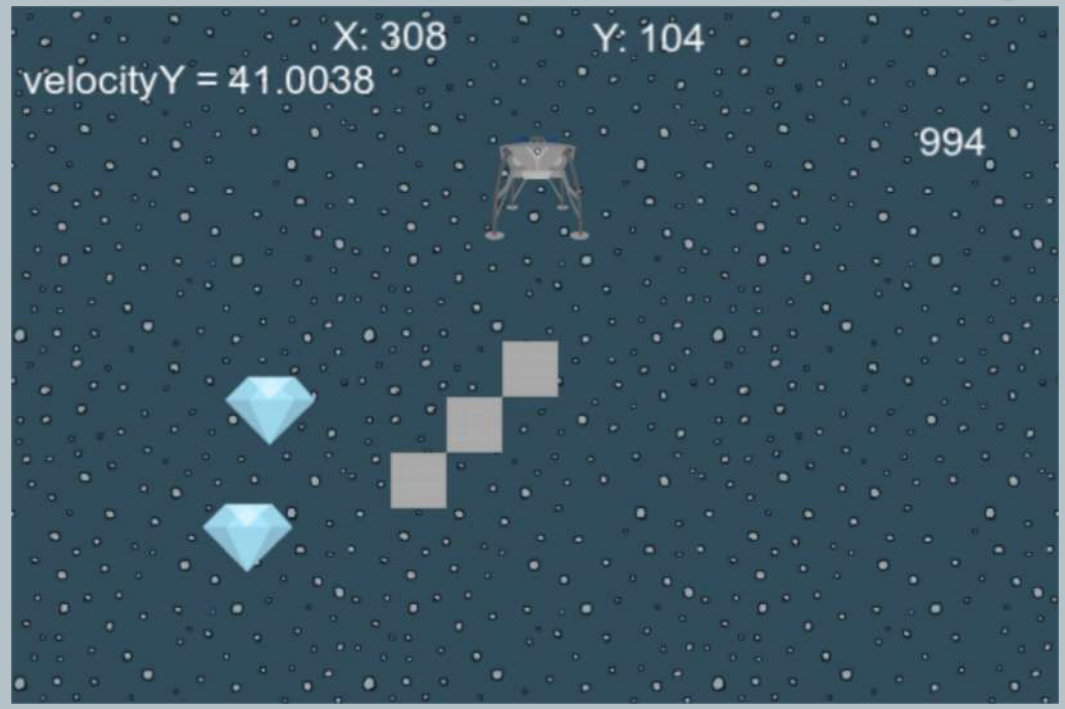
Goal: Students will customize the Moon Lander game,

Means: Students will add additional obstacles, power ups, and goals to make the game their own.

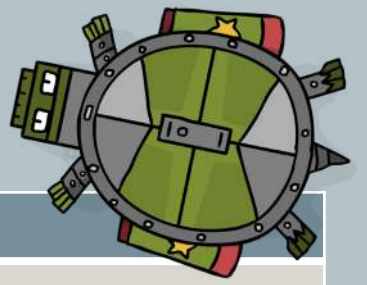
New coding tasks \ concepts:

There are no new concepts in this final challenge. The goal of this challenge is for students to take what they have learned and use it to customize the game.

At a minimum, to complete the challenge students must add at least one tile obstacle (as in Challenge 12) and at least two diamond power ups (as in Challenge 16). Ambitious students can add more - perhaps stars that award points and another counter to track the player's score.



Reference Card



Keyword/Function/Property	Applies To	Description
`==`	NA	Tests if the value on the left is equal to the value on the right. Not to be confused with the = (single equals) operator which assigns the value on the right to the variable on the left.

`-=, +=`	NA	Subtracts or adds to the current value of a variable.
		
		Example:
		
		`x = 10`
		`x += 5`
		`# Now is 15`

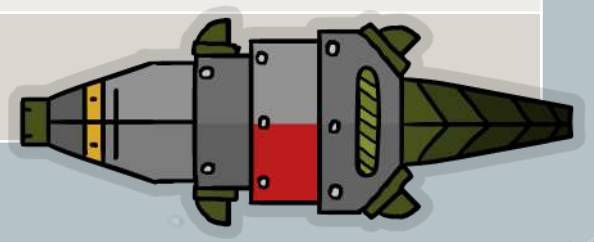
`thrust()`	sprites	Applies force that will cause the sprite to move forward in the direction it is facing unless some other force or obstacle stops it. Takes one numerical argument that represents the strength of the force to apply.
`rotateRight()`, `rotateLeft()`	sprites	Applies force that causes the sprite to spin in the appropriate direction unless some other force or obstacle stops it. Takes one numerical argument that represents the strength of the force to apply.
`onKey()`	sprites	Event handler function that is called when the player presses a keyboard key. Accepts one argument, a code that represents the key that was pressed.

Reference Card

Keyword/Function/Property	Applies To	Description

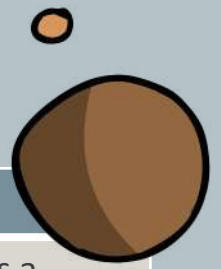
`onCollide()`	sprites	Defines a collision handler function for when this sprite collides with a specific other sprite. Takes two arguments. The first is the name of the other sprite. The second is a function to be called when the collision happens.
		
		Example:
		
		`@onCollide surface, () =>`
		`  win.play()`

`onCollideWithTileMap()`	sprites	Event handler that is called whenever the sprite collides with *any* tile. Takes no arguments.
`onUpdate()`	sprites	Called every time the game updates the screen, usually 50 - 60 times per second. Can be used to update counter and text widgets to reflect the game's state.
`destroy()`	sprites	Destroys the sprite and removes it from the game.
`play()`	sounds	Causes the sound to play.



TOP

Reference Card



Keyword/Function/Property	Applies To	Description
`getVelocityX()`	sprites	Returns the sprites current velocity in the horizontal direction. The velocity is returned as a floating point number. A positive velocity means the sprite is moving to the right and negative velocity means it is moving to the left.
`getVelocityY()`	sprites	Returns the sprites current velocity in the vertical direction. The velocity is returned as a floating point number. A positive velocity means the sprite is moving up and negative velocity means it is moving down.
`text`	text widgets	Represents the text displayed by the widget. Changing this property automatically changes what is displayed.
`value`	counter widgets	Represents the numerical value displayed by the widget. Changing this property automatically changes what is displayed.
`reset()`	game	Causes the game to reset to its initial state, just as if the RUN! button had just been clicked. **Example:** `game.reset()`

Sprite Review

Sprite	Description
	The player controls the **spaceship** sprite, using the left and right arrow keys to turn and the up arrow key to thrust forward in the direction the ship is facing.
	In challenges 1 through 6 the goal of the game is to navigate the spaceship to reach one or more **star** sprites.
	Starting with challenge 7, the goal of the game is to land the spaceship safely on the landing pad represented by the **surface** sprite.
	In challenges 16 and 17 **diamond** sprites are available as powerups to replenish to the spaceship's fuel supply.