



# A Coding-Only Guide to Making the Game

## Creating the Files

---

- If using DrRacket:
  1. Make a new folder for each team, naming them after the students.
  2. Into each folder, download the [DrRacket file](#) and the [unzipped] [teachpacks](#)
- If using WeScheme:
  1. Start a new program for each team, and click "save". Name the projects after the students.
  2. Into each file, copy and paste the code from the [online template](#), then click "save". **Note:** for this and all other curriculum files, please make sure you are copying the WeScheme code and NOT the DrRacket code. They are very similar, but not interchangeable.
- We suggest using student names (e.g. - "janelle+chris") for the file/WeScheme name, but anything will do. If you're in WeScheme, we assume you've logged in using a class-wide username and password, so that every student can access their games.

## Running the Game

---

- Near the end of the game file, you'll see an expression that defines a level, ( `define game_level ...` ), passing in animation functions and images. You can define other levels if you wish, with different images or animation functions.
- The last line, ( `play game_level` ) will run the game using the level created above. By default, this line will be executed every time the student clicks "Run". You can stop this by adding a semicolon at the beginning of the line: `;(play game_level)`. This can make it easier for student to experiment with their code and test cases while they're developing, and they can always run the program manually by typing ( `play game_level` ) in the Interactions window.

## Adding Images to the Game

---

- Open the Game Template file for a team of students.
- Towards the top of the file, you'll see some variables defined for the `BACKGROUND`, `PLAYER`, `DANGER` and `TARGET`.
- At the moment, they are defined to be simple images. For example:
 

```
(define BACKGROUND (rectangle 640 480 "solid" "black"))
```
- **In DrRacket**, replace the image-producing expression with one that pulls in the image file for your background, using the relative filepath of the image as a String:
 

```
(define BACKGROUND (bitmap "tami+chris/water.png"))
```
- **In WeScheme (or DrRacket 5.1)**, replace `bitmap` with `bitmap/url`, and the full URL of their images as a String:
 

```
(define BACKGROUND (bitmap/url
  "http://exampleschool.edu/tami+chris/water.png"))
```
- Time to test it out! Click "Run", and then evaluate those variables in the Interactions window. Typing `BACKGROUND`, for example, should show you the image you used for the Background. Once you see all images loading successfully, click "save".
- You can rotate, scale, or flip the images using Racket functions:
 

```
; rotate : Number Image -> Image
; scale : Number Image -> Image
; scale/xy : Image Number -> Image
; flip-horizontal : Image -> Image
; flip-vertical : Image -> Image
```
- If the image has a solid background that you'd like to remove, you can also use the custom function `clipart/url` to import the image. This will remove the background from any image, using whatever color is found on the first pixel. [This is not supported on IE 7 or 8](#)
- You can also export the individual pixels from an image using `image->color-list: Image -> Listof Color`. These pixels can then be modified programmatically, and then restored to an image using `color-list->image : Listof Color -> Image`. See [this example](#).
- If you wish to perform more complex manipulations, you can also download the image and modify it with the editor of your

choice (Paint, GIMP, etc). However, you will need to upload the new file to someplace web-accessible, so it can be imported with `bitmap/url` or `clipart/url`.

## Making Characters Move

---

- Every Bootstrap game includes at least three "characters": the `Player`, the `Target` and the `Danger`.
- Each is controlled by an `update-` function, named for the character it updates. These updates are like the pages of a flip-book: given a page, the update function generates the next logical page in the book.
- These functions take the current coordinate as input, and return the next coordinate. If the `Target` moves to the left by 20 pixels each frame, then the body of the function should return `(- x 20)`.
- You'll note that there is also a function called `update-mystery`. This function is actually for animating projectiles, if students insist on having them! (In the curriculum, this is kept a secret early on in order to encourage students to think beyond violent games.)
- The `update-player` function is slightly different from the rest:
  - It takes *two* inputs: the y-coordinate of the player (a number) and the key that was pressed (a string, such as "h", "left" or "q").
  - It is called only when a key is pressed, instead of each time the screen is redrawn
- If you'd like the player to move differently depending on which key is pressed, you'll need to use `cond`. For example:

```
; update-player : Number String -> Number
; given the y-coordinate and a direction, output the NEXT y
(define (update-player y key)
  (cond
    [(string=? key "up")    (+ y 10)]
    [(string=? key "down") (- y 10)]
    [else y]
  ))
```

- Notice the `else` clause at the bottom! Without it, pressing a key that is *not* "up" or "down" will end the program with an error!

## Making the Target and Danger Come Back

---

- The teachpack will automatically regenerate the characters if the `onscreen` function returns `false` when given the character's x- and y-coordinates.
- By default, the function returns true - meaning that the characters are always considered onscreen. Once they get moving, they fly off the screen, never to return!
- The "onscreen" logic should also take into account that the characters have width and height! For example, a 200px-wide character will still be partially onscreen even when it's x-coordinate is -50. For this reason, we recommend a "buffer" of roughly 50 pixels on all sides.
- In the lesson plans, students write functions that check the left and right sides of the screen separately:

```
; protect-left? : Number -> Boolean
; Determines if the x-coordinate is greater than -50
(define (protect-left? x)
  (> x -50))

; protect-right? : Number -> Boolean
; Determines if the x-coordinate is less than 690
(define (protect-right? x)
  (> x 690))
```

- Then they write a single function, called `onscreen?`, which is only true if their characters are protected on the left AND the right:

```
; onscreen? : Number Number -> Boolean
; Determines if the coordinates are within 100 pixels of the screen
(define (onscreen? x)
  (and (protect-left x)
       (protect-right x)))
```

- If you have an unusually wide or tall character image, you may wish to use larger buffer values.

## Detecting Collisions

---

- To detect whether or not two characters have collided, the game code needs to know two things: how close they are, and

how close they are allowed to be before a "collision" occurs.

- You can illustrate this by using the `*distances-color*` variable. Setting it to any color will draw triangles to show the distance between the `player`, `target` and `danger`. Setting the value to the empty string `""` will hide them again.
- The lengths of each the legs is calculated by the `line-length` function. Modify the body so that it correctly calculates the distance between `a` and `b`. (You can do this using `abs` or `cond`):

```
; line-length : Number Number -> Number
; the distance between two points on a number line
(define (line-length a b)
  (abs (- a b)))
```

- The hypoteneuse length is calculated by the `distance` function. Students will code the distance formula here, using the `line-length` function:

```
; distance : Number Number Number Number -> Number
; The distance between two points on screen:
; We have the player's x and y, and a character's x and y.
; How far apart are they?
(define (distance px py cx cy)
  (sqrt (+ (sq (line-length px cx))
           (sq (line-length py cy)))))
```

- Finally, `collide?` starts out always returning false, so nothing collides even when you have the distances all working. Use the distance formula, with some threshold:

```
(define (collide? px py cx cy)
  (< (distance px py cx cy) 50))
```

## Exploring the Teachpack

---

The bootstrap-teachpack lets you do more -- try the [Supplemental Lessons](#) for some ideas and challenges: play with 2d motion, random numbers, a different `collide?` function, or harder play based on `*score*`.

Bootstrap by [Emmanuel Schanzer](#) is licensed under a [Creative Commons 3.0](#)

[Unported License](#). Based on a work at [www.BootstrapWorld.org](http://www.BootstrapWorld.org). Permissions

beyond the scope of this license may be available at [schanzer@BootstrapWorld.org](mailto:schanzer@BootstrapWorld.org).

