This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Games

What to Do

1. After watching all of the Pong videos, submit Pong.
2. After watching all of the Mario videos, submit Mario.

When to Do It

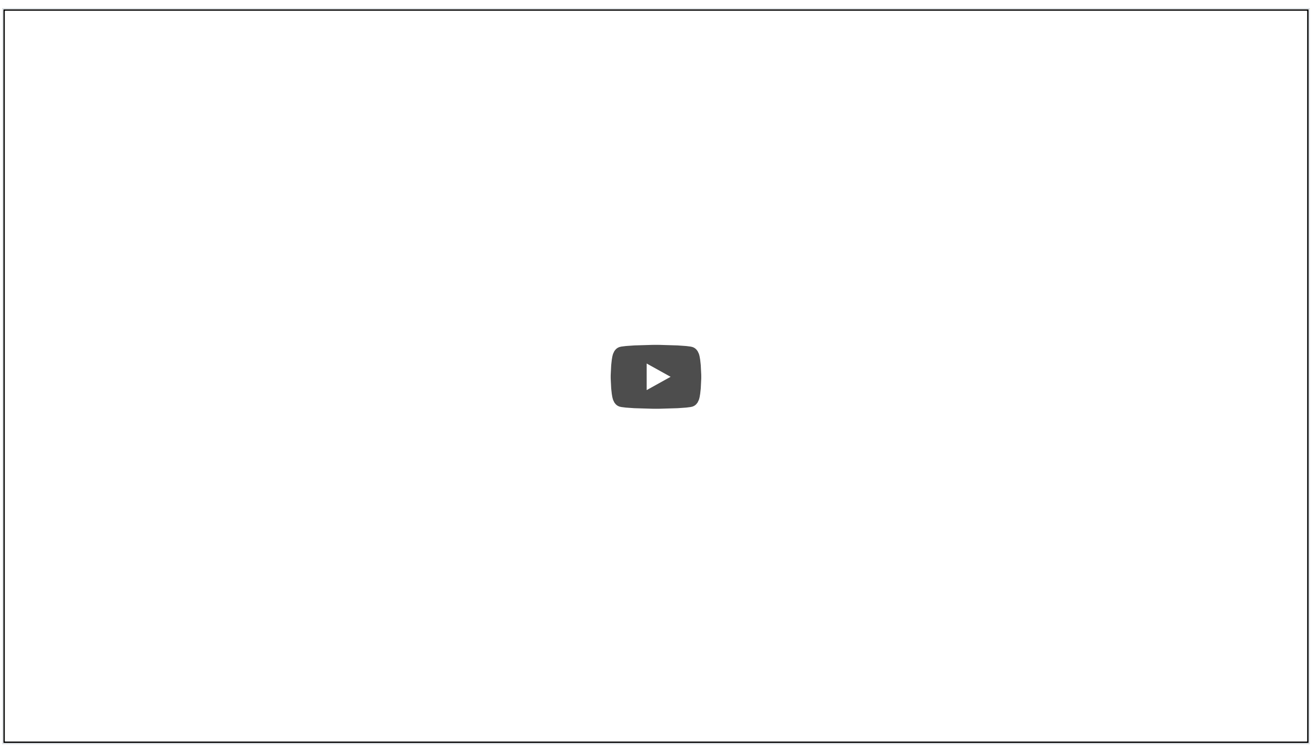
By 11:59pm on 31 December 2020.

How to Do It

▶ **Introduction**

Pong

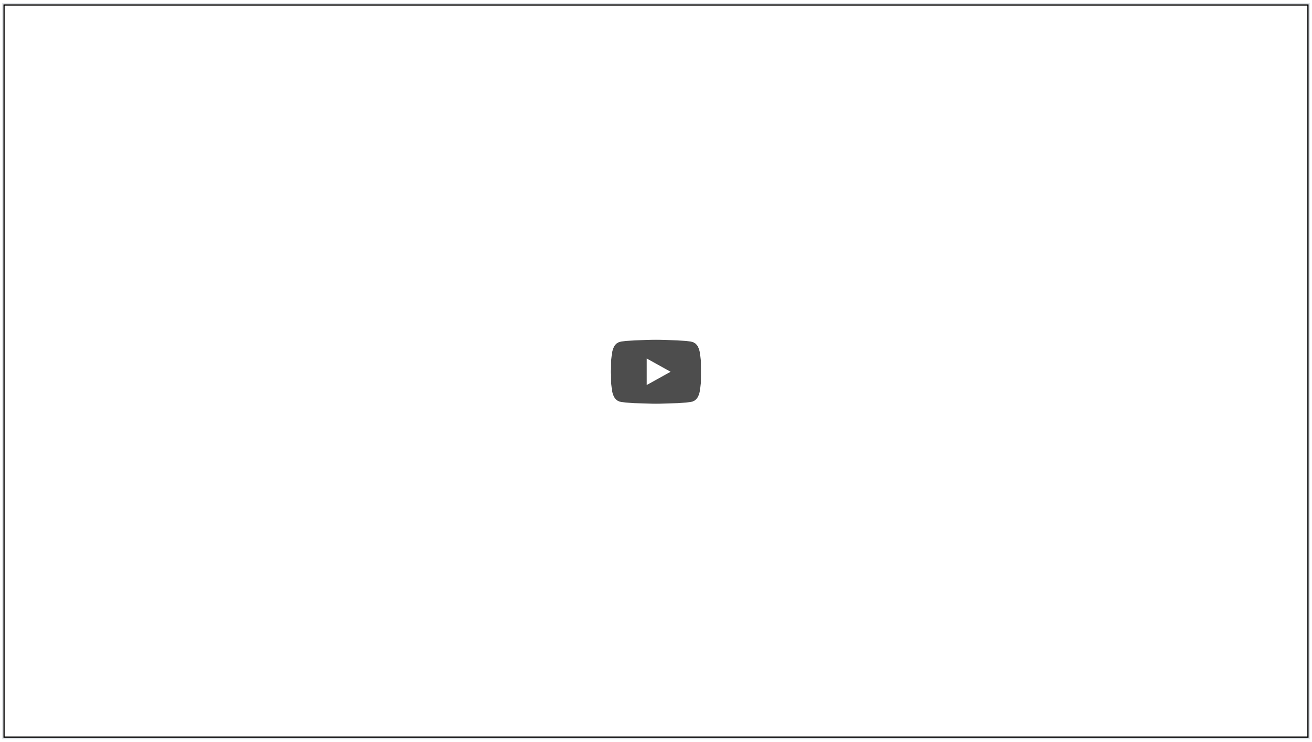
▼ **Pong**



- We'll be recreating Pong, one of the first video games ever made, where we have a ball bouncing between two paddles.
- [1:00] We'll be using a programming language called Lua, similar to JavaScript, focusing on "tables", which are like objects in JavaScript or dictionaries in Python, with key-value pairs.
- [2:35] LÖVE is a 2D game development framework written in C++, but allows us to use Lua to write games that run on the framework, with

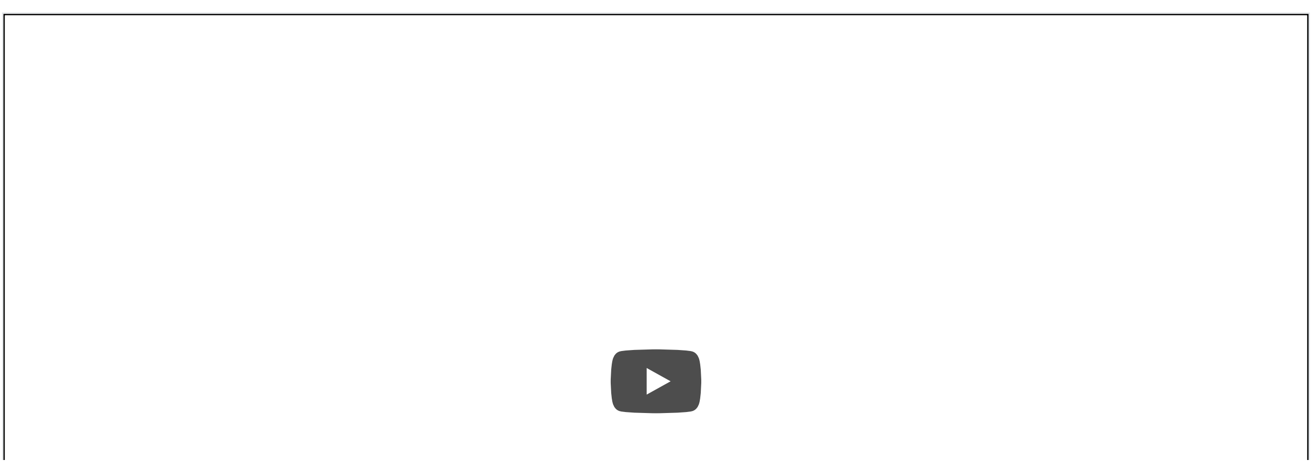
- [2:55] LÖVE is a 2D game development framework, written in C++, but allows us to use Lua to write games that run on the framework, with built-in features like graphics, keyboard input, math, audio, physics, and more.
- [4:10] We'll need to install LÖVE from [https://love2d.org/ (https://love2d.org/)](https://love2d.org/), where we can also find its documentation and wiki. The wiki also has instructions for installing it easily for our operating system. We'll need some kind of code editor, such as Visual Studio Code, Atom, or Sublime Text.
- [6:15] We can open VS Code, and create a new folder for our project with a file called `main.lua`. This is the main file that Lua will run. We'll add a small function and then drag our folder on top of the Lua application, which will run our code for us. In VS Code, we can also add an extension to do this with just a keyboard shortcut.
- [10:30] In our games, we'll rely on the 2D coordinate system, where our top-left corner is 0, 0, and x is the horizontal distance from left to right, and y is the vertical distance from top to bottom.

▼ **Pong 0**



- We'll start by looking at how a game works. The game loop represents the basic form of what our program will be doing: first, processing input from the player; second, updating the game's state, perhaps multiple times depending on how much time has passed; third, rendering the graphics for the player; and finally, repeating.
- [2:40] LÖVE will expect us to implement some functions, `load()`, which will set up the initial state of our game; `update(dt)`, which will be called to update the game state; and `draw()`, to update the screen.
- [4:10] We'll make a new folder, and open it in VS Code, and write our `main.lua` file. Our `load()` function will set the size of our game screen after we set some constants for ourselves, and also use a table, or a set of key-value pairs, to specify more details about the game window.
- [7:25] Next, we'll implement `draw()`, where we can use `printf` to center a welcome message.

▼ **Pong 1**



- Our text size was small, so we'll actually decrease the resolution and stretch it out in our window, which will also give us a retro, pixelated look.
- We'll need a third-party library, [push (https://github.com/Ulydev/push)](https://github.com/Ulydev/push), which will give us a `push.lua` file we can include in our project.
- [2:00] We'll see some new functions: `setDefaultFilter()` , to increase the size of images without smoothing; `keypressed()` , to handle what to do when a key is pressed; `quit()` , to end our application.
- [3:25] We can see the difference with filtering on textures, or images, as we change their scale. Point, or nearest neighbor, just increases the size, and other types smooth out the edges, possibly adding blurriness.
- [3:45] Now, we'll write `keypressed()` to quit our game if the escape key was pressed.
- [4:35] With the `push` library, we'll be able to set up the screen with a virtual width and height of smaller dimensions, scaled up to the actual size we want our window to be. In our `draw()` function, we'll also need to set a flag to use the `push` library to draw each time. We'll also set the filter to use the `nearest` (neighbor) filter. Now, we can run our program and see everything as we expected.

▼ **Pong 2**

- We'll add rectangles to represent two paddles and a ball, as well as use a new font for our title.
- [0:50] We'll need some more functions from the `love.graphics` library: `newFont` , `setFont` , and `clear` , which clears the entire screen with some color.
- [2:00] Finally, we'll use `rectangle` , which will draw a rectangle at some poistion and size.
- [2:15] https://dafont.com will have many fonts we can use, and we'll put one font in our game project directory. We'll open the font file, and set the font to use.
- [5:15] Before we start drawing anything else, we can use the `clear` function to clear the window to a color, and we'll set colors with decimal values from 0 to 1 for each of red, green, and blue, and we can also pass in values out of 255 since we might be more familiar with that format elsewhere.
- [6:50] We'll draw rectangles by positioning and setting their sizes by calculating their coordinates.

▼ **Pong 3**

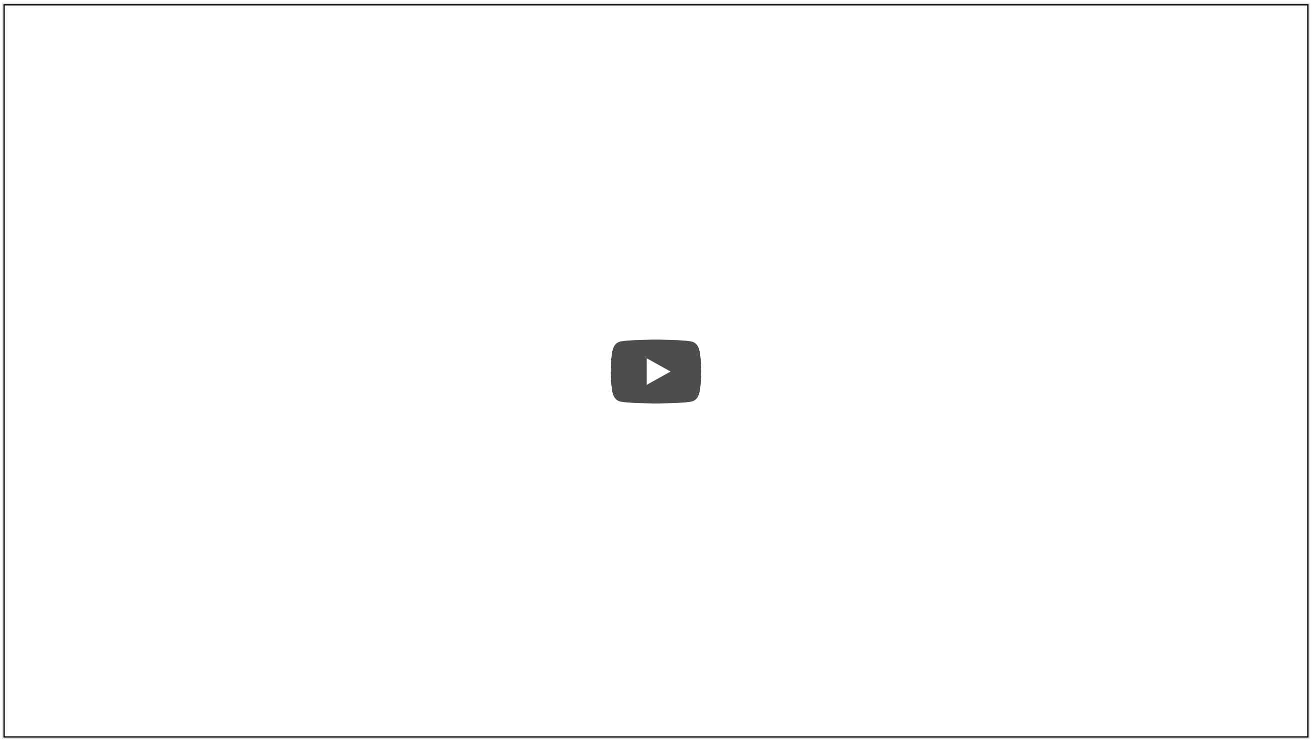
- We'll add the score, and check if a key is being held with `isDown`, to move our paddles.
- [2:00] We'll set our font size to be bigger when we're drawing the score, and track them in variables.
- [4:50] The next thing we'll need to do is track the horizontal position of our paddles, so we can use them in our `draw()` function and change them in our `update()` function. `dt` is the amount of time that has passed since the last frame, or rendered screen, so we want to multiple our changes by that so our changes appear smooth and consistent to the player.
- [7:35] Now we'll check if certain keys are down, and set the y coordinates by some change based on the `dt`, delta time, so that the speed of the paddles are fixed.

▼ **Pong 4**

- We'll also want to change the game state based on the position of the ball, and move it on the screen automatically.
- [1:00] We'll need functions like `math.randomseed`, `os.time`, and `math.random`. By passing in a seed, or starting value, to a random number generator, we can get the same series of "random" numbers. Alternatively, we can pass in a unique value, like the current time in seconds, to make sure the sequence is (highly likely) unique. We'll also use `math.min` and `math.max` to get the smaller or larger of two numbers.

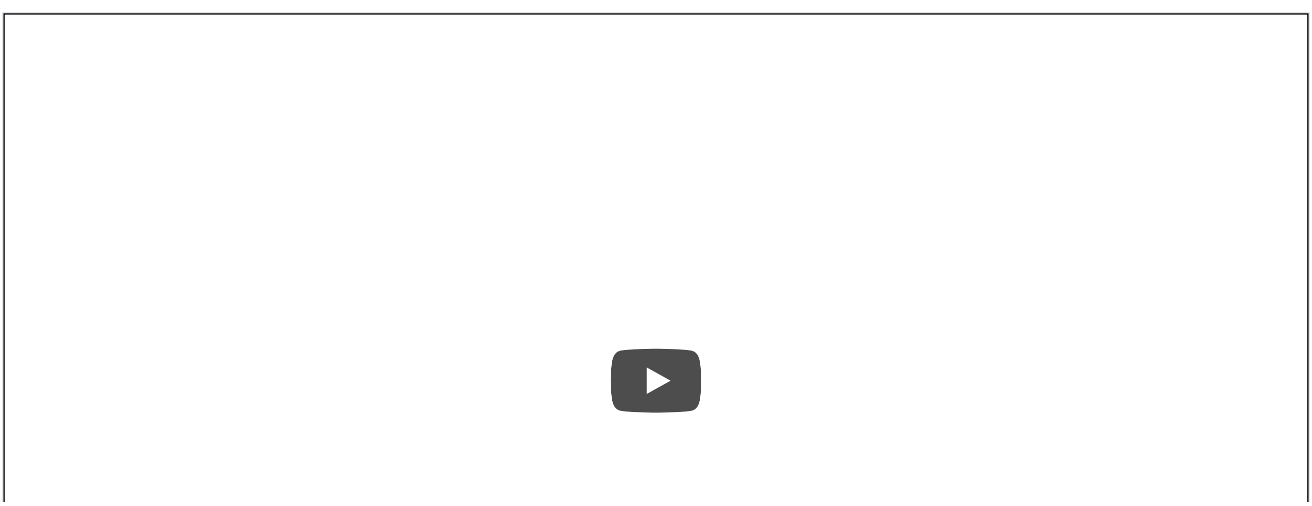
- [3:25] We'll make sure that our paddles' y coordinates are not too low or too high, so they can't go off the screen.
- [5:45] Now let's move our ball with some random starting velocity, and store its current position as variables we can change every time we `update` . We'll also track the game's state, where we don't want the ball to start moving until the game state is in `play` mode. We'll add a `keypress` case for the enter key to change the game state from `start` to `play` . And after we're in the `play` case, we can reset the ball's position too, so we can play over and over again.

▼ **Pong 5**



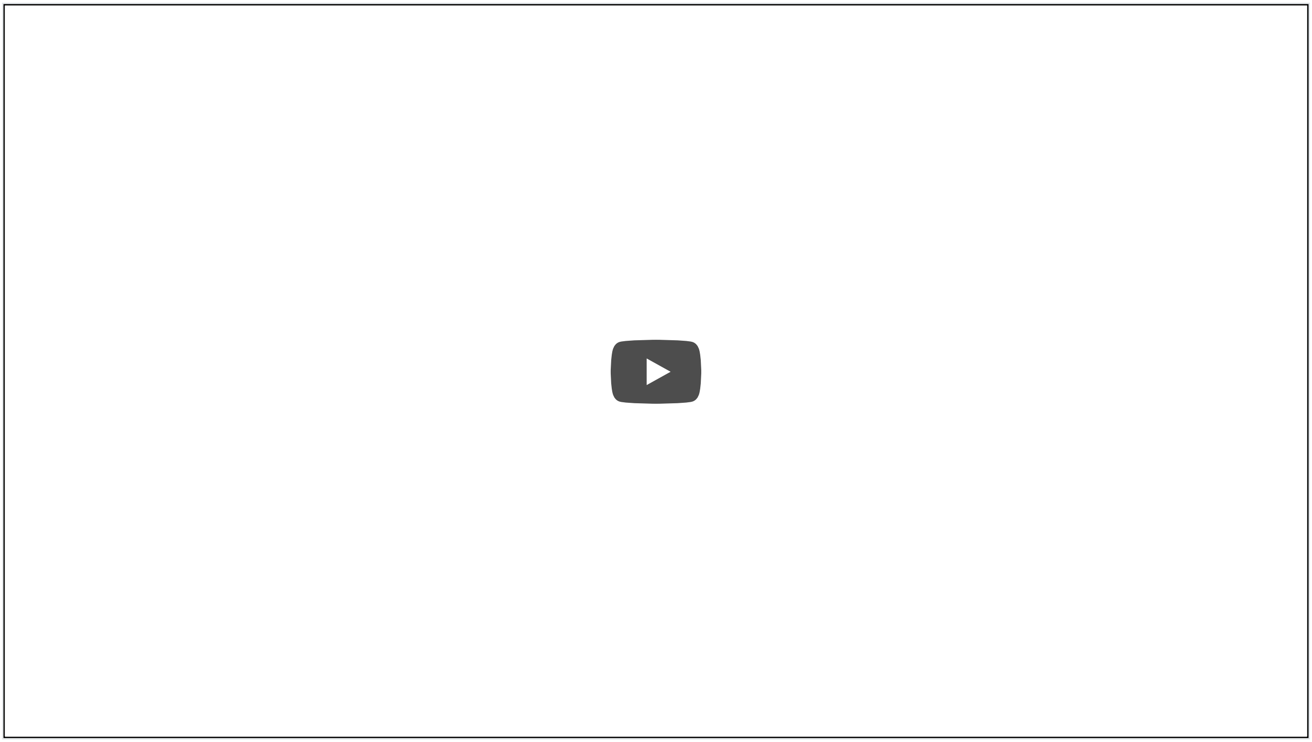
- Now that our game is getting more complicated, we'll start refactoring our code so that it's better designed and compartamentalized.
- Object-oriented programming centers on the concept of classes, where we have objects of a particular class that are like structs with a group of variables and functions, or methods, that acts on each object. In other words, a class is like a blueprint, from which we can create objects, or instances, each with their own state.
- [3:50] We'll use a third-party library for its cleaner implementation of classes on top of what Lua supports. We'll create `Paddle.lua` and `Ball.lua` , where we create our classes and define an `:init()` function that describes how to create new instances of each class. The colon indicates that this function will run on each individual object when they are created.
- [7:45] In our `init()` function, we'll take in some variables about its original position, and set them on the `self` variable, or the object's reference to itself. Then, we can define a `update(dt)` function on our Paddle object for it to move itself. We'll also need to define a `render()` function on our Paddle, so it can draw itself by being called in our main `draw()` function. And we'll need to construct each Paddle with the right initial parameters.
- [16:40] Our Ball class will be similar, with `init()` , `update()` , and `render()` methods that have the behaviors we want for our ball. We'll also add a `reset()` method so our ball is centered again when the game is restarted.

▼ **Pong 6**



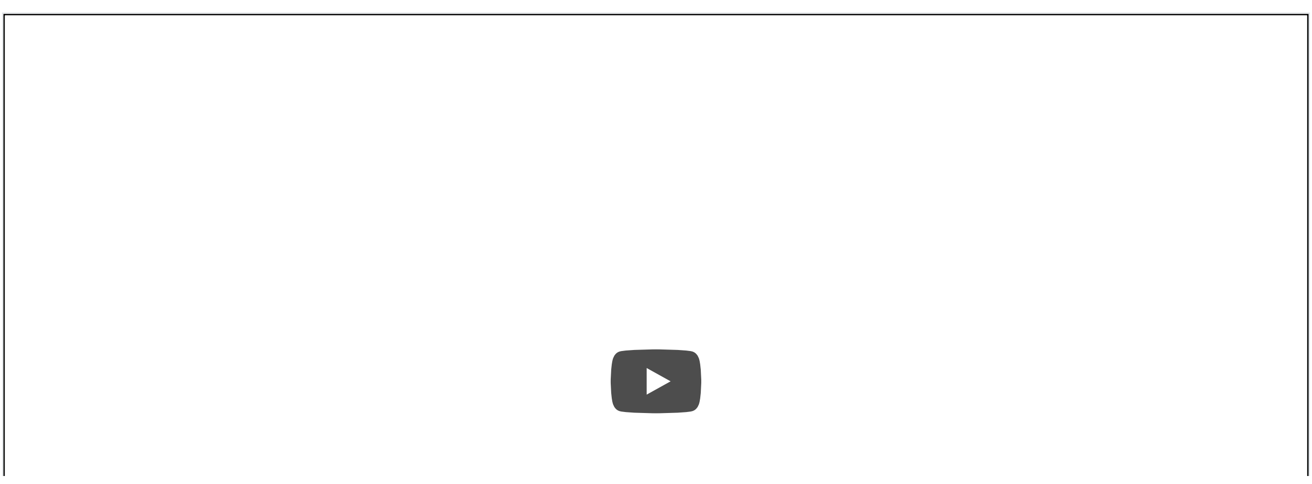
- We'll add an FPS, or frames per second, counter to our window so we can get a sense of how our game is performing. We can use a built-in `getFPS()` function, and at the end of our main `render()` function we'll also call a `displayFPS()` function that writes the value from `getFPS()` to the screen.

▼ Pong 7



- Now we'll implement collision detection, where the ball can actually bounce from the paddle or the edges of the screen, instead of passing through them.
- We'll use a simple system, "axis-aligned bounding boxes", which just means that everything in our world has a rectangular shape, and we can test for collision by just comparing the boundaries of the coordinates, as opposed to having to calculate curves or diagonals.
- [2:05] Our Ball class can have a `collides()` function that takes in some box, and caculates if any of its own edges are overlapping with the given box. In our main `update(dt)` function, we can pass in the paddles and reflect the ball's x velocity. And at the top or bottom of the screen, we can also reflect its velocity.

▼ Pong 8



- Now we can add a score to our game state, increment it if the ball reaches the left or right edges of the screen, and then reset the ball. We'll also clean up the game state so we can only go from the start state to the play state by pressing enter.

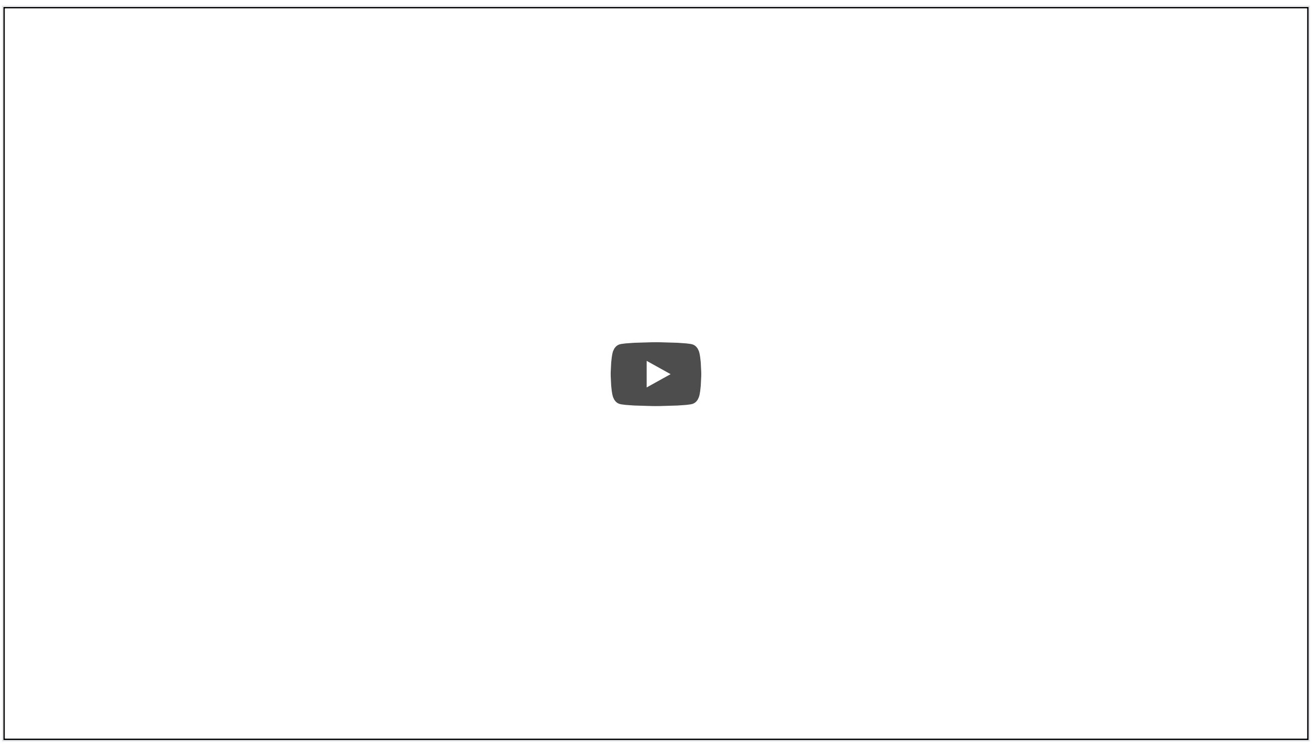
▼ **Pong 9**

- After the ball reaches the left or the right side of the screen, the other player scores. So on the next turn, we want the ball to start by going towards that opposing player, as though the player was serving the ball.
- State machines are like a map of different states and the transitions between them. For example, key presses might take a character in a game from a standing state to a jumping state.
- [2:15] So we'll need to have three states in our game, one that indicates we haven't started yet, one that indicates a player is serving, and one that indicates the ball is in play. We'll also want to update the ball's x velocity to be towards the right player, after one of them has scored.
- [9:25] We need to make sure that the starting velocity matches the randomly chosen player at the very beginning, too, so we'll set that on the ball too.

▼ **Pong 10**

- Now let's track the victory state, and reset the game entirely once a player has a certain number of points. We'll need to add another state to our game overall, and call it `victory` . We'll print a message when we get to that state, and check whether the score has reached some threshold every time a player scores.
- [2:10] We'll make a new variable to store the winning player when we check the score, and then in our `draw()` function we can check whether we're in the `victory` state and print a message with the winning player's number.
- [3:55] When the enter key is pressed, and we're in the `victory` state, we also want to reset the state to `start` and reset the scores to 0.

▼ Pong 11



- To add sound, we'll use the `newSource()` function in the `love.audio` library, to open an audio file. We'll also use [bfxr (https://www.bfxr.net)](https://www.bfxr.net), a program that lets us generate sound effects.
- [1:30] We'll save some `.wav` sound files from bfxr, and save them with names in our game project directory.
- [3:20] In our game program we'll create a variable, `sounds` , which is our first table that stores key-value pairs of strings of audio names to audio objects from `newSource()` .
- [4:55] And at each point of our game where we want the sound to play, we can just access the audio object and call the `:play()` function on it.

▼ Pong 12

- We'll add the ability to resize the window, and just call the `push:resize()` function when our main `resize()` function is called by the LÖVE framework.
- Our assignment will be to add some AI, or computer player, to play as player 2.

Mario

▼ Mario

- We'll be building a game inspired by Mario, with a blue alien as our main character, a colorful world, and blocks we can see and interactive with.

▼ Mario 0

- Let's start by adding some tiles. We'll need a sprite sheet, which has 16x16 tiles laid out in a grid in one image. We'll split this in our game program.
- We'll need a tile map, or a representation of what tiles to be rendered to the screen graphically. This might be stored in a file or generated programmatically. For example, we might have an array of numbers where each number stands for a particular type of tile.
- [2:25] We'll create a new folder for our game project, and set up the basics with `class` and `Push`. We'll need a Map object that we can create, update, and render. First, we'll set up our window, clear it to be a solid sky blue, and write some text to make sure everything works.
- [7:10] We'll make a `graphics` directory in our game project directory, and put in our sprite sheet with graphics. We'll give each tile a number, from left to right and top to bottom, and LÖVE has a helper we can call, `love.graphics.newQuad()`. We'll make a `Util.lua` file for this utility helper, and write a `generateQuads` function that will take some atlas, or spritesheet, and split it. Here we see local variables and a table as an array, into which we'll store each sprite.
- [16:00] In our Map class, we'll open our spritesheet file and create the individual sprites (quads) from it. We'll initialize the map to some default layout of blocks and empty sprites to represent open space. And we'll need a `setTile()` function in our map to set the value in our `map` table at the right index.
- [23:50] Our `Map:render()` function will draw the tile at the right index in our spritesheet by looking at our tile map and drawing the image at the right coordinates on the screen.

▼ **Mario 1**

- We'll add some scrolling from left to right with a default velocity, and we'll need to simulate a camera with `love.graphics.translate`, which shifts the coordinate system for us.
- In our Map class, we'll write an `:update()` function that our main update function will call, and inside it we'll just change the position of the camera at some speed, by storing and updating the coordinates of our camera. And we have to convert the offset to an integer so there aren't artifacts from decimal values.

▼ **Mario 2**

- We’ll learn to control scrolling in our game with the keyboard, and we’ll do this by add conditions for whether certain keys are being held down in our map’s `:update()` function.
- We’ll also need to prevent the camera from going too far off the map, so we can use `math.max` or `math.min` with the right boundaries.

▼ Mario 3

- Now let’s generate a map procedurally, or programmatically with the help of random values. We’ll generate each column as we go from left to right, since our game scrolls from left to right.
- [1:35] We’ll go over the distribution code that’s already been written, and in our Map class we’ll have a function that generates each column, or horizontal offset on our map with random values. We need to generate blocks representing the floor, and in our assignment we’ll extend this to build a pyramid.

▼ Mario 4

- Now we’ll add an avatar of a blue alien which will be the character we can later control.
- In our Map class, we’ll need a Player class, so we’ll create one and initalize it to some defaults. Our Player class will also open the spritesheet for itself, and render the standing sprite for now.

▼ **Mario 5**

- We’ll add support to move our character with the keyboard, and have our camera track our character by positioning the camera so the character is always in the middle of the screen, unless the camera is already at the edge of the map.
- In our Player class, we’ll check for keyboard input, and move our character left and right. We aren’t checking for any other tiles, so we’ll be floating on the screen, but our camera follows our character as expected.

▼ **Mario 6**

- Now we'll add animations, where our character can face different directions. Our sprite sheet has a quad for each frame, and we can specify how many frames each quad is shown, giving us the illusion of an animation.
- We'll also need to represent the character's state, where we know whether they are standing, moving, or jumping, and the transitions between them will have different animations.
- [2:35] We'll create an Animation class, and in it we'll store the texture, frames, and intervals from the parameters it was created with. We'll write functions that get the current frame to show, to restart the animation, and to return the frame that should be rendered after some `dt` , delta time, has elapsed.
- [9:10] Now, in our Player class, we can create Animations with the spritesheet of all our frames as the texture, and the array of the frames that will make up each individual animation.
- [11:35] We also want to maintain the state of the character by using a `behaviors` table, which will map some state to a function that will update our position depending on what states we're in. This is the same for checking the keyboard for movement while standing or walking, but it will let us easily trigger different animations depending on the direction we're moving.
- [16:30] Our frames have our character facing to the right, so we need to track the direction we're moving and pass that into the `Player:render()` function to flip the frame from the animation. We need to set the origin point to the center as opposed to the top left, so the image is flipped in place.

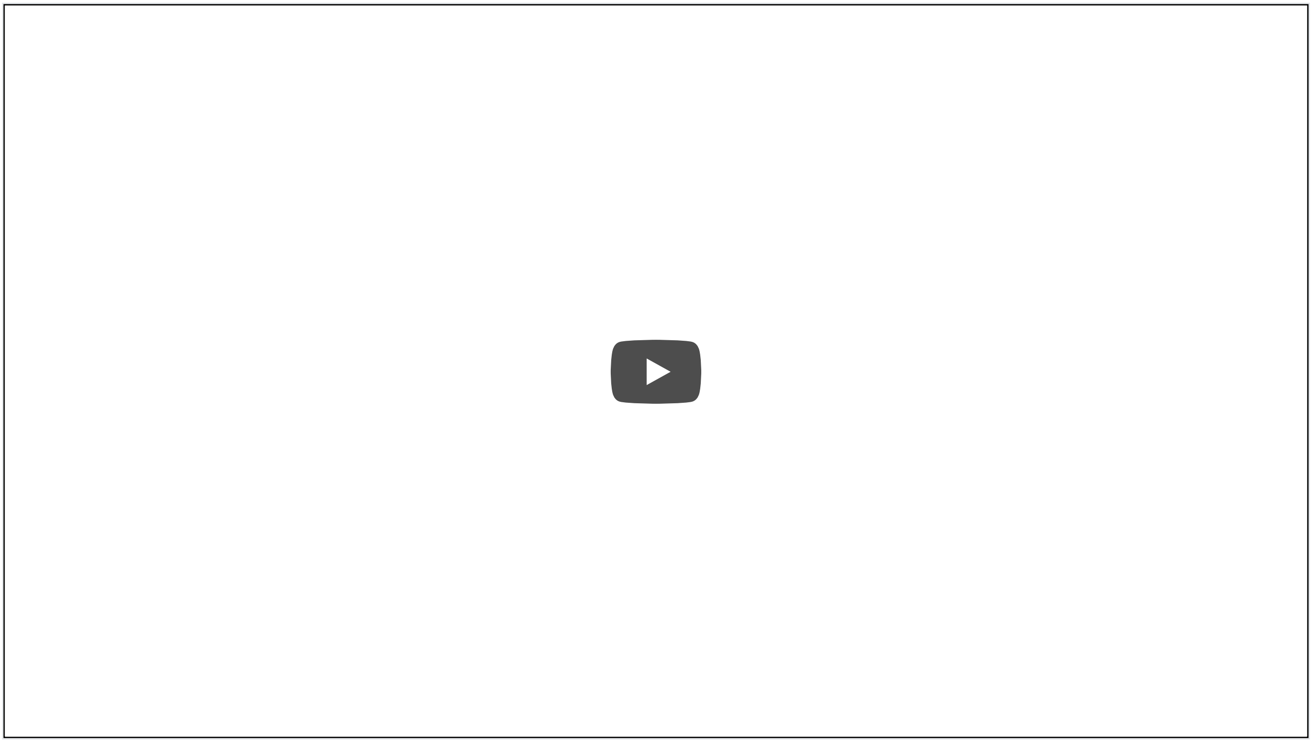
▼ Mario 7

- We'll add another state, jumping, which will require us to model gravity by changing the y velocity of our character over time;.
- [1:15] We'll create a JUMP_VELOCITY variable and GRAVITY variable as constants, and add `jumping` as a new state with its own behavior. From the idle and walking states, we'll check for whether the space key was pressed, and then change our state and animation if so. And in

From the idle and walking states, we'll check for whether the space key was pressed, and then change our state and animation if so. And in our `jumping` state, we'll also allow for horizontal movement by checking for the same keys for moving left and right. We'll need to stop falling once we're at our original vertical height, too.

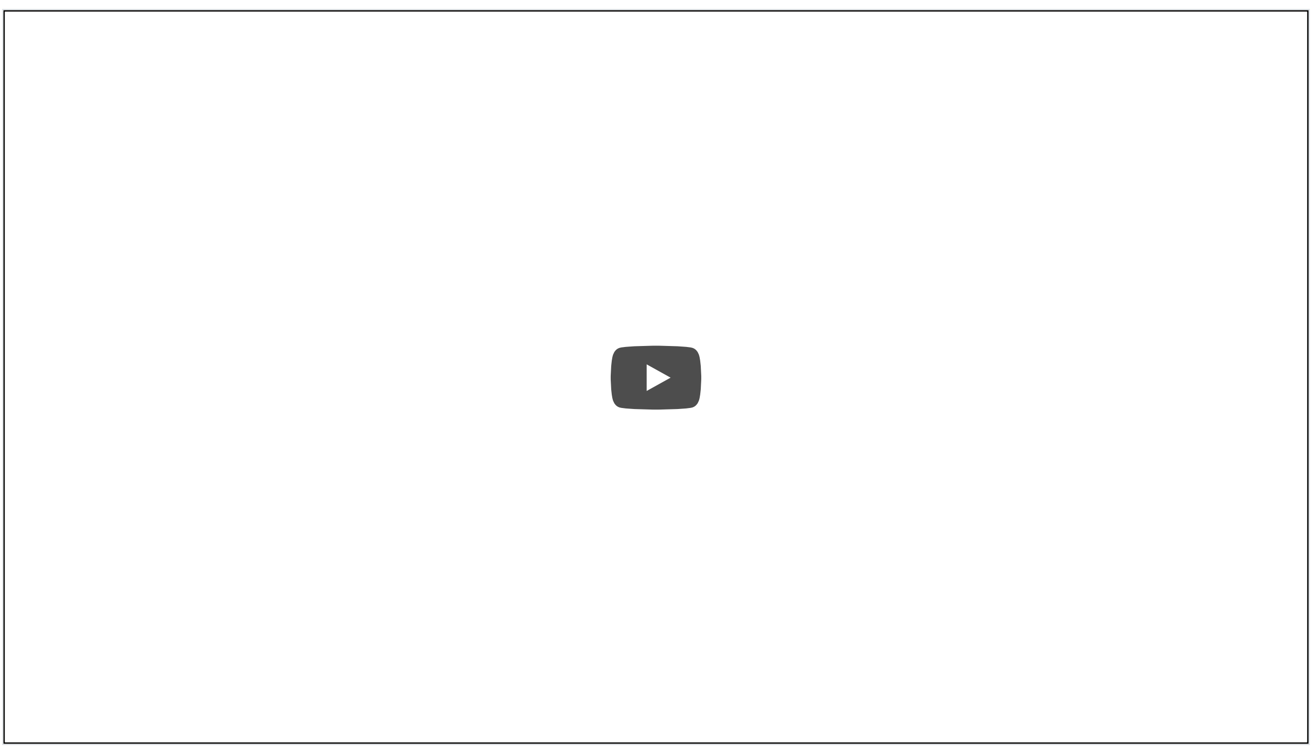
- [9:00] It turns out that we need to implement a `wasPressed` function to check whether a key was pressed, since that doesn't come with LÖVE, so we'll add a table that will store the keys that was pressed since the last frame. After each `update` , we'll clear the table, but during our `update` , we'll be able to see what keys were pressed since the last `update` .

Mario 8



- We'll add collision checking by checking if the bounding box, or the coordinates of the rectangle around our character's sprite, overlaps with the bounding boxes of any block tiles. Since we can only hit a tile from below, we only need to check the top corners of our character.
- We'll review the code that's been written, where we start by getting the tile at some location, and if there is a tile at the coordinates of the top left or right corners of our character as our player is jumping, we want to reset our vertical velocity and set the tiles to another value.

Mario 9



- Now, we'll extend our collision detection to include the left, right, and bottoms of our character, so we can fall into gaps or stop at a mushroom column.

- [1:15] Our update function will need to check for left and right collisions, and to do that we'll check if the tile we are touching is "collidable", or in a list of tiles that we can't walk through.
- [3:35] And in our Player class, we'll check for tiles that we're moving into or below us. If we're walking and there is no tile beneath us, we'll need a special case to put us in the `jumping` state, with negative vertical velocity. If there is a tile below us, we can stop our falling state by setting our y velocity to 0, so we can effectively jump on top of tiles. And if there is a tile to our left or right, we'll also need to make sure our position is outside of the tile, and set our x velocity to 0 as well.

▼ Mario 10

- Like we did in Pong, we'll add some sound effects and music playing on a loop. We'll use `love.audio.newSource` , and we can use built-in functions to ensures our music is on a loop and plays when the game starts. We'll have a `sounds` table, and play them at the right moments in our Player's `:update()` function too.
- In our assignment, we'll add a pyramid and flag to our map signifying the end of our level.

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Pong: The AI Update

Objectives

- Download LÖVE.
- Read and understand all of the Pong source code from the last track lesson.
- Implement a basic AI for either Player 1 or 2 (or both!).

Distribution Code

Download this project's [distribution code (https://cdn.cs50.net/2019/fall/tracks/games/pong/pong.zip)](https://cdn.cs50.net/2019/fall/tracks/games/pong/pong.zip).

It's Game Time

Your first assignment in this track will be a fairly easy one, since the dive into game programming can be deep enough as it is without having to implement an entire code base from scratch! Instead, we'll take the Pong example we covered and extend it in a small but fun way by giving one of the paddles (or perhaps both!) logic for playing the game so that you don't always need a buddy to play the game with you!

Of course, the code won't run if you don't have the LÖVE framework installed, so we'll have to tackle that in addition to grabbing the code; choose the version for your system here:

[Download LÖVE (https://love2d.org/)](https://love2d.org/)

For further information on how to actually run games, please visit the following page:

[https://love2d.org/wiki/Getting_Started (https://love2d.org/wiki/Getting_Started)](https://love2d.org/wiki/Getting_Started)

Additionally, if you're new to the Lua programming language (what LÖVE expects you to write in order to work), check out the following online reference manual:

[Lua Programming Manual (https://www.lua.org/manual/5.3/)](https://www.lua.org/manual/5.3/)

Once the code and LÖVE have been downloaded and installed, the actual change you'll be making to the code base is small, but it will require you to understand what many of the pieces do, so be sure to watch each of the track's lessons and read through the code so you have a firm understanding of how it works before diving in! In particular, take note of how paddle movement works, reading both the `Paddle` class as well as the code in `main.lua` that actually drives the movement, located in the `update` function (currently done using keyboard input for each). If our agent's goal is just to deflect the ball back toward the player, what needs to drive its movement?

Your goal:

- Implement an AI-controlled paddle (either the left or the right will do) such that it will try to deflect the ball at all times. Since the paddle can move on only one axis (the Y axis), you will need to determine how to keep the paddle moving in relation to the ball. Currently, each paddle has its own chunk of code where input is detected by the keyboard; this feels like an excellent place to put the code we need! Once either the left or right paddle (or both, if desired) try to deflect the paddle on their own, you've done it!

How to Submit

To submit your code with `submit50` , you may either: (1) upload your code to CS50 IDE and run `submit50` from inside of your IDE, or (2) install `submit50` on your own computer by running `pip3 install submit50` (assuming you have [Python 3 (https://www.python.org/downloads/)](https://www.python.org/downloads/) installed).

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/tracks/games/pong
```

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Mario: The Goal Update

Objectives

- Read and understand all of the Mario source code from the last track lesson.
- Add a pyramid of blocks to the generated level.
- Add a flag at the end of the level that either loads a new level or simply displays a victory message to the screen.

Distribution Code

Download this project's distribution code (https://cdn.cs50.net/2019/fall/tracks/games/mario/mario.zip).

It's Game Time

Your second assignment in this track will be a step more difficult than the last, but charmingly tied back into an earlier problem set in the course; this time, rather than constructing a pyramid using hash marks, you'll be creating a pyramid of tiles in a procedurally generated level. Additionally, you'll create the characteristic end-of-level flag that delineates one level from another, choosing to either end the level with a message or transition to a brand new one!

Your goal:

1. Add a pyramid of blocks to the generated level. Taking into consideration the column-based generation we discussed in the track, find a way to generate a Mario-style pyramid like the below, placed directly atop the ground (ASCII flag to the right shown as well):

   ```
      #      ~
     ##      |
    ###      |
   ####      |
   ###############
   ```

   You may choose to alter the pyramid such that it is symmetrical, but avoid a pyramid going the opposite direction, for Mario therefore won't be able to climb it! Also be careful to avoid starting the generation too close to the end of the level :)

2. Add a flag at the end of the level that either loads a new level or simply displays a victory message to the screen. Also tied to generation, this time take the flag and flagpole sprites included in the distro's sprite sheet and create a flagpole at the end of the level that, upon Mario's collision, triggers either a victory message or a reloading of a brand new procedurally generated level.

How to Submit

To submit your code with `submit50`, you may either: (1) upload your code to CS50 IDE and run `submit50` from inside of your IDE, or (2) install `submit50` on your own computer by running `pip3 install submit50` (assuming you have Python 3 (https://www.python.org/downloads/) installed).

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/tracks/games/mario
```