Tile-Based Platformer: Cave Climber

COS 426 Final Report

5/10/2021

Raiden Evans and Bharat Govil

Abstract

When deciding on a graphics project to work on for our final assignment, we were both interested in exploring the challenges associated with more traditional two-dimensional platformers. Tile-based platformers, such as the original Super Mario Bros. released in 1985, were a staple in early console gaming. There are many advantages associated with designing and simulating levels in a tile-based manner, including computational efficiency, ease of adding features, and ease of level design. One aspect that set the original Mario games apart from their competitors was their use of intricate character simulation. Instead of directly moving the player across the screen when a button is pressed, the player is physically simulated to include movement acceleration, friction, and variable jump height, all parameterized to change depending on if Mario is on the ground or in the air. It is our goal to explore the design challenges associated with creating such an interactive graphics system in order to learn the design challenges faced by early platformer developers. Before jumping into much more modern, complex video game implementations, it may be wise for us to first understand the fundamentals.

Key Components

To create a project that satisfies our learning interests, there are a few key components that we would like to demonstrate. A list of the desired features can be found below, along with a short description:

1. Levels stored as tiles: Each level should be stored as a grid of tile types, not individual objects.
2. Ease of development: Tiles should be easy to modify, and it should be easy to add new types of tiles.
3. Tile collisions: The player's object should appear to collide with some tiles, but not others.
4. Level swapping: A new level should be loaded when the player reaches the objective.
5. Smooth player control: The player should feel good to control through simulated physics.
6. Player animations: The player should be animated based on its movement, enhancing the game feel.
7. Visual effects: Visual effects should be explored, e.g. dust particles from the player sliding.

For our project, the minimum viable product (MVP) should include features 1, 3, 4, 5, and 6. Ease of development primarily assists in making stretch goals easier, such as adding interesting blocks like deadly spikes, bouncy blocks, blocks that prevent jumping, etc. Additional visual effects, while pleasing, are not necessary for a functioning game. Nevertheless, it is our intent to provide these features for the final demonstration. Player animation may also be viewed as not entirely necessary, but without a properly animated character, much of the attachment the human player has with the virtual player is lost. We set player animation as our requirement for the MVP in order to prioritize it, as more benefit would be gained from its addition than by the addition of polish-like visual effects.

Starting Out

We first wanted to understand the framework behind tile-based games. To do so, we looked through online tutorials and decided to work with example code that used a tile-based architecture on base HTML5 and javascript (link in Works Cited). Since we could not find any

tile-based architecture that was created using Three.js, we adapted the concepts that were used to render a game in base HTML5 and javascript, to a framework involving Three.js methods. The sections below expand upon each of the key components we implemented.

Levels Stored as Tiles

The primary framework for any tile-based game is how a level is stored and rendered through tiles. For our approach, we created a simple tile map using a 2D array to store the physical details of our levels. Here, each position within the 2D array represents a tile, and the value at each position represents a specific type of tile. In the image below, for example, 0 represents a walkable tile while 1 represents a wall.

```
var map = [
  [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
  [1,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
  [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
  [1,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
  [1,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1],
  [1,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,1],
  [1,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,1],
  [1,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
  [1,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1],
  [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1],
  [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
];
```

Figure 1. 2D Array representation of a level

Once we have a tile structure in place, the next step is rendering this structure on our screen. We do this by using a tilemap. Tilemaps are image files that contain the many different tile textures that are used to render different types of tiles. We associate each unique value in the 2D tile array with a specific tile texture on a tilemap. Then, when we need to render the level, we iterate through each tile in the map 2D array, checking which texture from the tilemap is associated with the given tile, and adding a new sprite at the corresponding location on the scene with the associated tile texture.

Ease of Development

Tile-based games naturally lend themselves to having an easy workflow, so long as the code is organized properly and hardcoded checks in the form of if-statements and switch-statements are avoided. We decided to organize the code in a development-friendly

manner by creating important singleton classes that can be accessed by any section of the code. These classes serve as global data structures, where all of the important logic relating to new features can be found in one place. Singletons that we have created so far include a TileData class, a Keyboard class, and a SceneManager class.

The TileData class stores all of the features relating to different tile types. The materials of the tiles can be swapped out here, as well as the collision types (solid vs. no collision). Future features such as lethal spikes or bouncy blocks should have new data structures added to this class that map each tile type to a feature type. This class also contains public functions that request feature information from a tile, such as its material or type of collision.

The Keyboard class simply stores the states of each usable button on the keyboard. When "keydown" or "keyup" events occur, the Keyboard class is directly modified to update which keys are being pressed at any given time. This way, only one event listener is needed for each type of event, and the state of the keyboard can be known by any piece of code at any time.

The SceneManager class was created to assist with the goal of having level swapping. Instead of having the main render loop keep track of which scene is currently displayed, the SceneManager class has a function runScene() that can be called. runScene() updates the currently loaded scene and renders it. If a level needs to be switched out, the switchScene() function can be called, with the number of the level specified as an argument. Each scene also has a load() and unload() function, loading in all of the objects of the scene and cleaning up the scene for unloading, respectively.

Tile Collisions

There are several ways tile collisions can be implemented to create the appearance of an object bumping into a tile. The first important implementation is that each tile should have a way of storing what type of collision behavior it has. For our implementation, we currently only have two collision behaviors: no collision and solid collision. The tiles that the player can move through are representative of "empty" or "air" tiles. The tiles that make up the physical world should be "solid." Another type of tile that is possible are platform tiles, where the player can jump up through the tile, but collides when falling back down. This type of collision is not currently implemented.

One simple method of colliding with the world can be understood by collecting positional and collision information on a grid of neighboring tiles, and simulating collision for each tile's bounding box. A graphic of such an example can be seen below in Figure 1, where a 4x4 grid of tiles is found around the player.

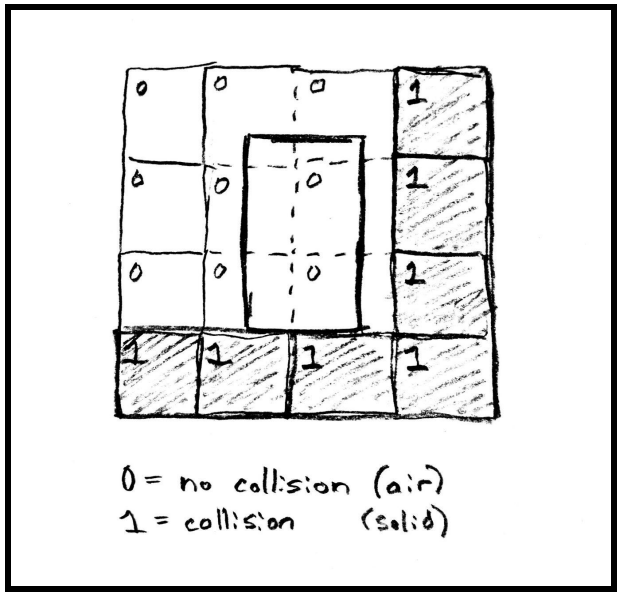

Figure 2. Simple collision implementation, where a set grid of tiles are found around the player and collisions are simulated with each tile's bounding box.

A more efficient method, however, involves sampling tiles at preset offsets from the player's centroid. There are four different types of collision checks: bottom checks, top checks, right checks, and left checks. Each "check" simply finds the tile at its given position and calculates whether or not the player should be offset either up, down, left, or right, depending on if we're checking the bottom, top, right, or left side of the player, respectively. An example set of collision checks is drawn below in Figure 2. Note that check positions must not be further than one grid-width apart, otherwise gaps could form in the collision checks.

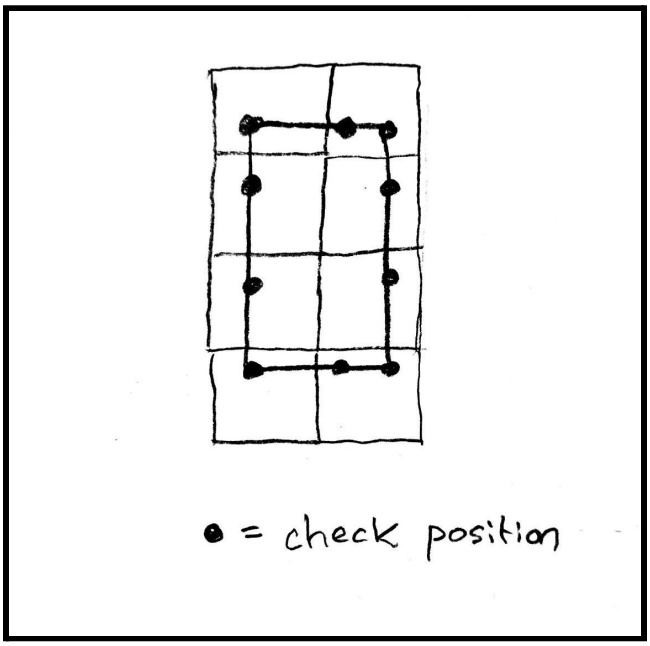

Figure 3. Efficient collision checks, where preset offsets around the perimeter of the player's bounding box are used to sample tiles from the level's tile grid.

Because we're checking the bottom, top, right, and left edges of the player independently, we must make sure that any repositioning of the player is indeed towards the closest edge of the tile. In other words, if the player intersects with a tile both horizontally and vertically, we need to make sure that the player is pushed towards the smaller of the two intersections. A visual example of this test along with pseudocode can be found in Figure 3. An example of a collision that should not be acted upon is found in Figure 4.

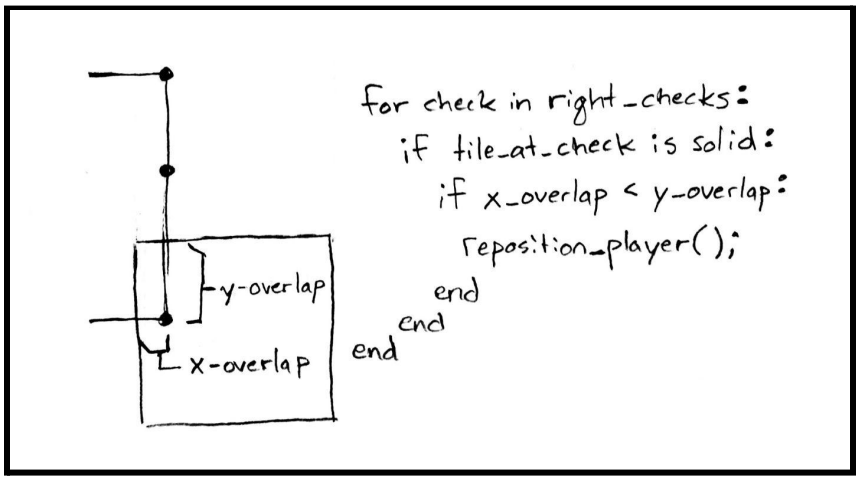

Figure 4. An example of the collision process for the right edge of the player's bounding box. This process is also performed for the bottom, top, and left edges of the player's bounding box.

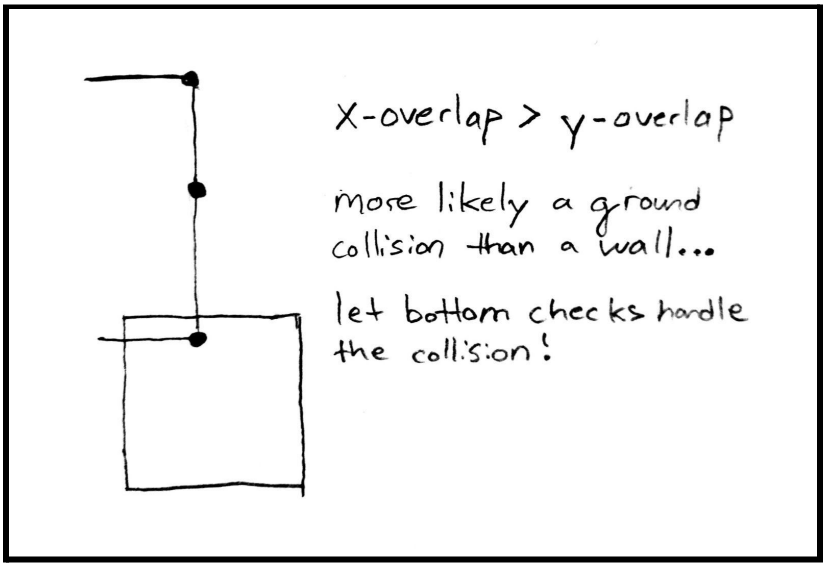

Figure 5. An example of the when the right collision checkers should not update the player's position. In this case, the collision is likely a floor collision instead of a wall collision, so the right collision system should do nothing.

One final note on the collision system is that there are instances where the player may be moving so fast that the player's bounding box intersects too far in one direction and the collision code pushes the player to the wrong side of a tile. This is easily fixed by simulating multiple collisions each frame. This is discussed in more detail in the Smooth Player Control section.

Level Swapping

If not careful, one may find that hard coding everything for a single playable level may make it difficult later on to add new levels. As mentioned before, the SceneManager class assists with making level swapping easy. In order to switch from one level to the next, a goal condition is checked. As a simple implementation, we have a line of code that checks whether or not the player has made it to a certain point in the first level. If the player has, then SceneManager.switchScene() is called, specifying that the second level should be loaded. This allows us to add custom goal conditions to each level, so long as each scene properly implements its load() and unload() functions.

Smooth Player Control

Due to our robust tile collision system, smooth player controls were relatively straightforward to implement. The player class has several attributes and functions, such as a sprite, a bounding box, and many state variables to determine whether the player is grounded, colliding with a wall, in the middle of a jump, etc. At its core, however, the player is defined by a Vector3 position on the currently loaded scene, and all user controls manipulate that one position. This player can be moved left or right with the left or right arrow keys, and can be made to jump with the up arrow key or spacebar. All of these interactions are based on a velocity and acceleration system, where player positions are updated according to these variables once every frame (60 times every second). At every frame, the player's current velocity is first updated according to the player's acceleration, which is determined by player state and user controls. Then, we use the collision system to check if the bounding box around the player's projected new position (when updated according to velocity) will collide with a solid block. We divide the displacement between the intended new position and original position of the player at a given frame into five increments, and check for collisions at each added increment to the player position. In this manner, we simulate a 300Hz collision check environment. If a collision occurs, the bounding box for the player's position is kept strictly out of the overlapping solid block, and the player's velocity is updated to reflect the collision.

To promote smooth gameplay, the player's acceleration and velocity are often modified according to the player's state. When the player is grounded, they have a horizontal acceleration of 1. That is to say, for each frame that the user holds down the left or right arrow keys when the player is grounded, the player's velocity in the horizontal axis increases or decreases by 1 tile per second. Similarly, this value is 0.5 when the player is not grounded. We set the maximum value for horizontal velocity at 8 to limit the player's maximum speed, and also add air resistance and friction scaling factors so the player will slow to a stop when the user is not pressing an arrow key. We simulate gravity by assigning a constant negative vertical acceleration of 1, with a maximum negative velocity of 20. When the player is made to jump with the up arrow key or spacebar, they are given an immediate initial vertical velocity of 20.

We have also expanded on our basic motion system by implementing additional movement capabilities, such as variable jump height, wall-drag, and wall jump. To implement variable jump height, if the user lets go of the jump button within 300 milliseconds of starting the jump, the player will have an immediate additional deceleration applied to themselves, which scales with the duration of the jump so far. The shorter the jump button is held, the larger this

additional deceleration value. In this manner, the player can jump to different heights without the implementation of variable jump height feeling abrupt. The wall-drag and wall jump movement capabilities are possible due to our robust player state system, which can determine whether a player is colliding with a wall either on their left or right side. Wall-drag is a movement mechanic where the player falls down through the air at a slower velocity than they normally would, by virtue of 'dragging' themselves down along the wall. To implement this, we simply check if the player is at a wall, and if their horizontal velocity is negative, we apply a diminishing scaling factor of 0.88 to the velocity each frame. In this manner, a player dragging against a wall will be subject to both increasing velocity from gravitational acceleration, as well as the diminishing scaling factor. This will eventually reach an equilibrium at a falling speed of around 7.3 tiles per second. For wall jump, we check which edge of a wall a player is touching, and if they press the jump button, we begin a new jump from that wall in a direction away from the wall.

Player Animations (work in progress)

Having proper player animations that match player movement is the only requirement missing from having a working MVP. Currently, we have demonstrated that a crude animation system works, where the sprite texture is offset along the sprite sheet at a specified framerate to create an animation. Example code was observed from the texture animation tutorial listed in Works Cited. The current animation only renders the character (currently a cat) walking. For a final animation system, a state machine will need to be implemented.

The animation state machine will keep track of which movement state the player is currently in. Example states include idling, walking, jumping, falling, wall sliding, and more. Certain states, like jumping, naturally flow into others, like falling. We still hope to use a free premade sprite sheet for the character, similar to the free cat sprite sheet we're currently using. The sprite sheet would need to contain relevant actions or must be easily modifiable to include new actions. If such a sprite sheet cannot be found, we may spend a little bit of time creating our own very simple cave exploring character, most likely in stickman styling.

Challenges ahead mostly lie in the design of the state machine so it isn't too disorganized, but it also appears to be a little bit tricky to flip the sprite along the y-axis. Being able to flip the sprite in code would save needing to manually modify the sprite sheet to include duplicates of each animation, one facing left and one facing right. Nevertheless, it should be possible to overcome these challenges in the coming days to make sure we meet the self-imposed definition of a minimum viable product.

Visual Effects (work in progress)

As of right now, our progress in visual effects has been limited to designing tile textures that fit our theme of cave exploration. The spikes, background wall, and icy platforms have all been independently created. The player sprite and chest texture, however, have been taken from freely available texture assets. In the future we are planning on adding more independently created tile textures, as well as implementing a particle engine based on Three.js architecture. This particle engine will allow us to create particle effects that link with player animations and

movement options, such as dirt particles that will erupt when a player jumps, lands, drags themselves down a wall, etc. We also intend to create particle effects for when the player finds a collectible at a chest, falls onto spikes, or any other such tile interaction we add in the future. Finally, particle effects throughout the overall environment are also being considered, such as ambient lighting or drifting particles.

Individual Contributions

Raiden Evans:

Project Code: Organized the classes that provided ease of use (SceneManager, Keyboard, TileData), developed the collision code, developed the level swapping system, and will be working on the player animation.

Written Report: Wrote the Abstract as well as the Ease of Development, Tile Collisions, Level Swapping, and Player Animations sections.

Bharat Govil:

Project Code: Worked on player motion control, level tile architecture, intro html code, gameplay mechanics such as spikes and collectible chests (see demo). Will be working on visual effects.

Written Report: Wrote Starting Out, as well as the Levels Stored as Tiles, Smooth Player Control, and Visual Effects sections.

Works Cited:

HTML5 + Javascript Platformer Tutorial:
https://www.creativebloq.com/html5/build-tile-based-html5-game-31410992

Three.js Texture Animation Tutorial:
http://stemkoski.github.io/Three.js/Texture-Animation.html