2D Games Format – Pathfinding Tutorial

Contents

Introduction:

Welcome to a tutorial on creating a 2D game that will implement Pathfinding. My name is Steven Raven, and I will guide you toward creating this specific project.

Pathfinding is a critical skill for game development allowing the means to make a non-playable character navigate to a specific destination. Learning these skills would create a game project that features stealth with patrolling enemies. In the tutorial, we will be making a survival game where enemies will try to get to the player position with this project.

Regardless of programming experience, this guide is intended to teach anyone the fundamentals of making a simple 2D pathfinding game. If you have coding knowledge and enjoyed the tutorial, I have left a series of challenges you can attempt after completing the project if you wish to expand your coding skills.

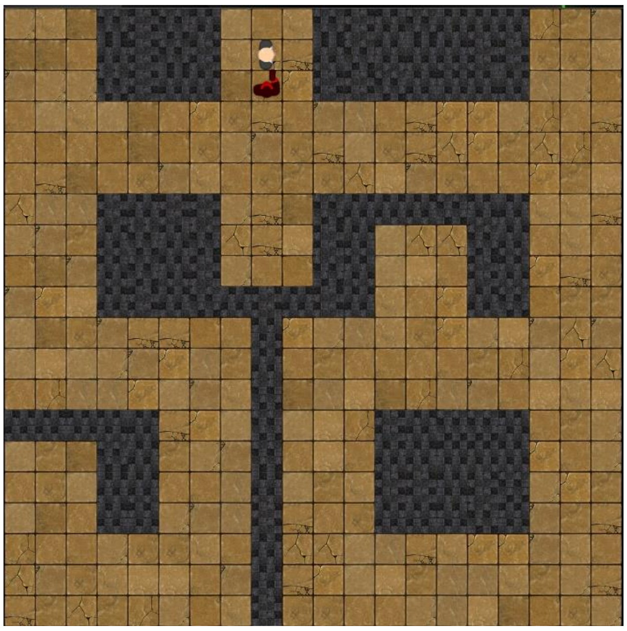
Figure 1 (Self Generated.)

What You Will Learn:

Throughout the project, you will learn several key skills such as:

- A complete understanding of Pathfinding of the A* algorithm.
- Spawning in Game Object through getting information from Tiled.
- Understanding how we can make use of classes.
- Making use of the EasyStar.js Script.

Before We Start:

First-time programming with Javascript? Check out W3 School (LINK) for more, as this is the language we will be using.

First-time using Phaser? Check out the API documentation (LINK) on the terminology used in the tutorial.

On a final note, this tutorial will not cover how to use Tiled or any other asset creation resources as the focus here is games programming in Phaser. We will only be extracting provided information from the levels JSON file.

Assets:

I have provided you with a base template with all assets already prepared for the project, including all necessary files to complete this tutorial. Please download a copy of the project template by clicking the LINK and then downloading the project's ZIP under “Code”.

If you are looking into making your sprites for this project, you can use Piskel (LINK), an online-based sprite editor. However, if you decide to do this, importing sprites is done using the below code within “preload()”.

```
// Load asset to the Scene.
this.load.image("example", "assets/example.png")
```

This code loads an image within a method by assigning a tag to where this can be called anytime in the code and the file to use. When naming, it is ideal that the tag name matches closely to the file so that you can easily remember what the object is for.

To use the A* Algorithm for Pathfinding, EasyStar* will be used through the guide. You can check and download a copy by clicking the (LINK).

Finally, if you want to change the whole level design of the project, you can download Tiled (LINK), the software used to create and edit the level in the template.

Step 1 - Understanding A*:

What is the A* Algorithm? A* is used to find the shortest path to the destination by calculating the vertices of the start and end points (see Figure 1).

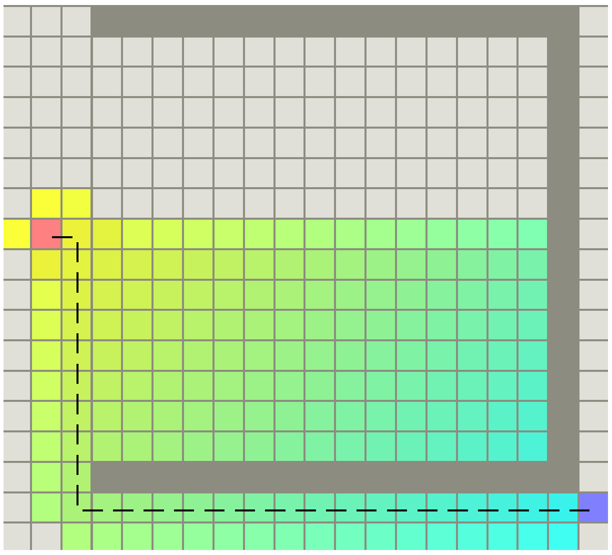

Figure 2 (Patel, 1997)

A line is drawn within the middle vertices of each grid space representing the most efficient path to the endpoint. The colours yellow to teal represent the distance between the start and end points.

In technical terms, h: yellow; g: teal and n: vertex; therefore, h(n) represents the exact cost of a path starting point and h(n) the estimated cost from its vertex goal.

Step 2 – Creating Level & Object Spawns:

Creating and spawning the levels and objects work by getting reference names of layers used and markers created within the provided level JSON file.

We need to make the correct JSON file to load and define the map. We do this by:

- Obtain the key name of the map
- Set the bounds of the physics
- Set the camera correctly
- Load the tilemap images
- Load the walkable path

This is done using the code below within the "create()" function to load the level scene to the game in "PathFindingScene.js".

```
// Create Tilemap.
this.map = this.make.tilemap({key: "tilemap"})
// Create Camera view.
this.cameras.main.setBounds(0, 0, this.map.widthInPixels,
this.map.heightInPixels)
this.physics.world.setBounds(0, 0, this.map.widthInPixels,
this.map.heightInPixels)
// Load Tileset to the Tilemap.
const tileset = this.map.addTilesetImage("tileset", "tileset")
//Load Tilemap Layers.
const groundAndWallsLayer = this.map.createLayer("groundAndWallsLayer",
tileset, 0, 0)
groundAndWallsLayer.setCollisionByProperty({valid: false})
```

Correctly coded, the scene in the HTML file should look like this (Figure 2).

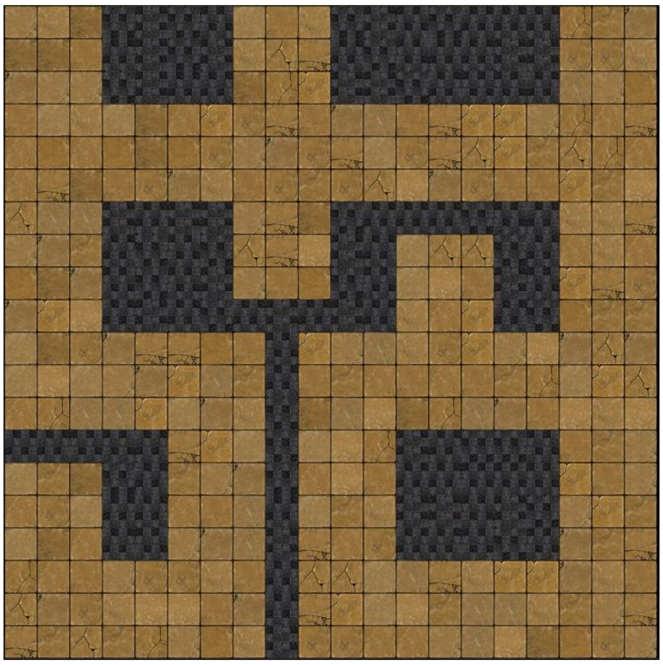
Figure 3 (Self Generated.)

To spawn in the player, enemies, and game objects, define a variable called “objectLayer” as “const” since this value never changes in the code. The value is:

```
// Get Tilemap Object Layers.
const objectLayer = this.map.getObjectLayer("objectLayer")
```

This code gets all the markers placed in the “objectLayer” in Tiled and then used as data objects. A for loop is used to load each data object to the scene based on its current location in Tiled.

```
// Load Object Layer Data Objects.
objectLayer.objects.forEach(function(object){
let dataObject = Utils.RetrieveCustomProperties(object)
if(dataObject.type === "playerSpawn"){
this.player = new Player(this, dataObject.x, dataObject.y, "player")
}else if(dataObject.type === "gunSpawn"){
// @ts-ignore
this.gun = this.physics.add.sprite(dataObject.x, dataObject.y, "gun")
}else if(dataObject.type === "enemySpawn"){
// @ts-ignore
this.enemySpawnPoints.push(dataObject)
}
}, this)
```

The for loop defines “dataObject” to be “Utils.RetrieveCustomProperties(object)”, which finds the marker points in Tiled JSON file in the asset folder and returns the locations. An if statement is used to check if “dataObject” correctly matches the exact data type name of the marker in Tiled. If met, It will then define the object on the exact location of the marker and load in a new sprite or class. Also, “// @ts-ignore” is used to ignore code errors below it when the code is intended to work.

Using the code below, sets the layers of what cannot be walked into during the game.

```
// Add Collision between the Player and the layer.
this.physics.add.collider(this.player.sprite, groundAndWallsLayer)
```

The code uses a “collider” method which adds collision between the player sprite and the “groundAndWallsLayer”. The Tiled JSON files use “valid” to define the walls and which ones are true can be walked on, such as the walkable brown tiles (Figure 2).

Step 3 - Creating Object Classes:

Classes are templates containing methods and variables in an object. When creating a class, these objects act as their own specific instance. Classes are a fundamental idea for object-oriented programming as they allow defining their methods and variables without changing the main scene.

Use classes to create and work on new JS files called “Player” and “Enemy”. Below is a basic template of a class.

```
// Here is a class example
class Example {
    /** @type {Phaser.Scene} */
    scene
    /** @type {Phaser.Physics.Arcade.Sprite} */
    sprite
    /** @type {number} */
    exampleNumber = 1
    /** @type {object} */
    keys
    /** @type {Phaser.Input.Keyboard.Key} */
    spaceBar

    constructor(scene, x, y, texture) {
    }
}
```

The class structure is first; you have the name (Example) used to reference and spawn into the main scene. The class holds data-types within its braces when creating the object. Standard data-type used for objects are sprites, numbers, and other classes for the project. Secondly, each class has a constructor, the initialization function to call on when using a new keyword to instantiate an object. Common code of the constructor will load the scene and the sprite.

To implement both classes, “constructor” and “update()”, toward giving the properties that the player holds, such as movement speed and controls; add in the start-up of setting our Pathfinding next with the enemy.

Insert below code in the “Player” and “Enemy” classes. Datatypes have already been entered.

Player:

```
    constructor(scene, x, y, texture) {
        this.scene = scene
      // Sprite to Load
        this.sprite = this.scene.physics.add.sprite(x, y, texture)
        this.sprite.setDepth(2)
        this.sprite.setCollideWorldBounds(true)
        this.spaceBar =  scene.input.keyboard.createCursorKeys().space
      // Player Controls.
        this.keys = scene.input.keyboard.addKeys({
            w: Phaser.Input.Keyboard.KeyCodes.W,
            a: Phaser.Input.Keyboard.KeyCodes.A,
            s: Phaser.Input.Keyboard.KeyCodes.S,
            d: Phaser.Input.Keyboard.KeyCodes.D
        })
        this.currentSpeed = this.standardSpeed
    }
    update(time, delta) {
      // Player Movement.
        if (!this.isDead) {
            if (this.keys.a.isDown) {
                this.sprite.setVelocity(-this.currentSpeed, 0)
                this.sprite.angle = 180
            }
            else if (this.keys.d.isDown) {
                this.sprite.setVelocity(this.currentSpeed, 0)
                this.sprite.angle = 0
            }
            else if (this.keys.w.isDown) {
                this.sprite.setVelocity(0, -this.currentSpeed)
                this.sprite.angle = 270
            }
            else if (this.keys.s.isDown) {
                this.sprite.setVelocity(0, this.currentSpeed)
                this.sprite.angle = 90
            }
            else {
                this.sprite.setVelocity(0, 0)
            }
            if (this.sprite.body.velocity.x != 0) {
                this.sprite.setSize(this.sprite.width, this.sprite.height)

            } else if (this.sprite.body.velocity.y != 0) {
                this.sprite.setSize(this.sprite.height, this.sprite.width)
            }
      // Player Fire Gun.
            if (this.hasGun && Phaser.Input.Keyboard.JustDown(this.spaceBar)){
                this.scene.events.emit('firebullet')
            }
        }
    }
```

Enemy:

```
    constructor(scene, x, y, texture) {
        this.scene = scene
      // Sprite to Load.
        this.sprite = scene.physics.add.sprite(x, y, texture)
        this.sprite.body.immovable = true

    }
    update(time, delta){
        // Time to Move?
        if(!this.pendingMove && this.sprite.x == this.targetX && this.sprite.y
== this.targetY){
            this.pendingMove = true
            this.scene.time.delayedCall(500, this.beginMove, [], this)
        }
        if(this.sprite.angle == 0 || Math.abs(this.sprite.angle) == 180){
            this.sprite.setSize(this.sprite.width, this.sprite.height)
        }else if(Math.abs(this.sprite.angle) == 90 ||
Math.abs(this.sprite.angle) == 270){
            this.sprite.setSize(this.sprite.height, this.sprite.width)
        }
    }
    beginMove(){
       this.scene.events.emit("enemyready", this)
       this.pendingMove = false
    }
```

Step 4 - Setting Up Pathfinding:

To set up the enemy's Pathfinding, we need first to load a file called “EasyStar.js” to the scene. EasyStar is the A* algorithm we covered earlier but written in Javascript to use for this project. The file is loaded by inserting the code within “create()” back in “PathFindingScene.js”

```
        // @ts-ignore
        this.finder = new EasyStar.js()
```

Before setting the Pathfinding, we need to spawn enemies on the map. First is setting an event on creating the enemy spawn mechanics and assigning their movement toward the player. Adding the below code within “create()”, whenever an active enemy is presented, they will move to the player. We will come back to setting “handleEnemyMove()” later.

```
        // Enemy Mechanics
        this.events.on("enemyready", this.handleEnemyMove, this)
        this.time.delayedCall(1000, this.onEnemySpawn, [], this)
```

The “delayedCalled” method is telling the scene to spawn a new enemy every 1000 milliseconds, which equals 1 second in “onEnemySpawn()”. Adding the below code in “onEnemySpawn()”, the scene loads a new enemy class and spawns one in one of the spawn points we had assigned back in step 2. Also, should the enemy overlap with the player, “collideEnemy()” function is called on to kill the player.

```
    onEnemySpawn() {
        let index = Phaser.Math.Between(0, this.enemySpawnPoints.length - 1)
      // Get Available Spawn Points.
        let spawnPoint = this.enemySpawnPoints[index]
      // Create New Enemy on Spawn Point.
        let enemy = new Enemy(this, spawnPoint.x, spawnPoint.y, "enemy")
        enemy.targetX = spawnPoint.x
        enemy.targetY = spawnPoint.y
        this.enemies.push(enemy)
      // Player Collides with Enemy.
        this.physics.add.overlap(this.player.sprite, enemy.sprite,
this.collideEnemy, null, this)
    }
```

Next, we need to load a for loop that contains a “grid” array of 2D map tiles and push this information to “EasyStar”. After this, we can then define the tiles based on the tilemap properties from “Tiled” to hold to the “acceptableTiles” array. These tiles are looked at and then added to that array. Insert the below code under “this.finder”.

```
        // 2D map tiles
        let grid = []
        for (let y = 0; y < this.map.height; y++){
            let col = []
            for(let x = 0; x < this.map.width; x++){
                let tile = this.map.getTileAt(x, y)
                if(tile){
                    col.push(tile.index)
                }else{
                    // If no tiles exist
                    col.push(0)
                }
            }
            grid.push(col)
        }
        // Get map info to EasyStar
        this.finder.setGrid(grid)
        // tileset Props
        let properties = tileset.tileProperties
        // Hold valid Tiles
        let acceptableTiles = []
        for(let i = tileset.firstgid -1; i < tileset.total; i++){
            // Look for tiles
            if(properties[i] && properties[i].valid){
                // Add Valid tile to list
                acceptableTiles.push(i + 1)
            }
        }
        // Which tiles can be used
        this.finder.setAcceptableTiles(acceptableTiles)
```

The “finder” we created at the beginning of this step will become essential for telling the enemy what tiles they can path. This becomes used in “findPath()”, which calculates the enemy's available path. We later call on “moveEnemy()” if there are any. Insert the below code in “findPath()”.

```
    findPath(point) {
        //  Point object of x and y to pixels
        let toX = Math.floor(point.x/this.map.tileWidth)
        let toY = Math.floor(point.y/this.map.tileHeight)
        let fromX = Math.floor(this.activeEnemy.sprite.x/this.map.tileWidth)
        let fromY = Math.floor(this.activeEnemy.sprite.y/this.map.tileHeight)
        let callback = this.moveEnemy.bind(this)
        this.finder.findPath(fromX, fromY, toX, toY, function(path){
            if(path === null){
                console.warn("No path found")
            }else{
                callback(path)
            }
        })
        // Execute Path Query
        this.finder.calculate()
    }
```

Any active enemy will move in any given direction based on the path length toward the player with a path found. The movement is done through a tween to bring smooth movement toward reaching the player and how fast they can move based on their class properties. The movement will stop whenever the player “isDead” is set to true. Insert the below code in “moveEnemy()”.

```
    moveEnemy(path) {
      // If Player is Dead, Do not Move the Enemy.
        if(this.player.isDead){
            return
        }
      // If Player is not Dead, Move Enemy of 4 different Direction.
        let tweenList = []
        for(let i = 0; i < path.length -1; i++){
            let cx = path[i].x
            let cy = path[i].y
            let dx = path[i + 1].x
            let dy = path[i + 1].y
            let a
            if(dx > cx){
                a = 0
            }else if(dx < cx){
                a = 180
            }else if(dy > cy){
                a = 90
            }else if(dy < cy){
                a = 270
            }
      // Use Above List to Active Enemies and Move Toward a New Tile.
            tweenList.push({
                targets: this.activeEnemy.sprite,
                x: {value: (dx * this.map.tileWidth) + (0.5 *
this.map.tileWidth), duration: this.activeEnemy.speed},
                y: {value: (dy * this.map.tileHeight) + (0.5 *
this.map.tileHeight), duration: this.activeEnemy.speed},
                angle: {value: a, duration: 0}
            })
        }
        this.tweens.timeline({
            tweens: tweenList,
        })
    }
```

Lastly, to ensure that the enemy can move to the player, we will need to look at using the “handleEnemyMove()”. Adding the code below in the function sets the active enemy to head toward the player current or last location on the map in the function. This is handled with the code we had written in “findPath()” to provide what tiles the enemy can walk on.

```
    handleEnemyMove(enemy) {
      // Tell active Enemy to find the Target. Target is the Player.
        this.activeEnemy = enemy
        let toX = Math.floor(this.player.sprite.x / this.map.tileWidth) *
this.map.tileWidth + (this.map.tileWidth/2)
        let toY = Math.floor(this.player.sprite.y / this.map.tileHeight) *
this.map.tileHeight + (this.map.tileHeight/2)
        this.findPath({x:this.player.sprite.x, y:this.player.sprite.y})
        enemy.targetX = toX
        enemy.targetY = toY
        this.findPath({x:toX,y:toY})
    }
```

And to finish setting the Pathfinding, we need to add the below code in the “update()” function, which will constantly update all information regarding the enemy and the players' current conditions.

```
    update(time, delta) {
        this.player.update(time, delta)
        for (let i = 0; i < this.enemies.length; i++){
            this.enemies[i].update(time, delta)
        }
    }
```

If all your code is implemented correctly, your game should look like this.

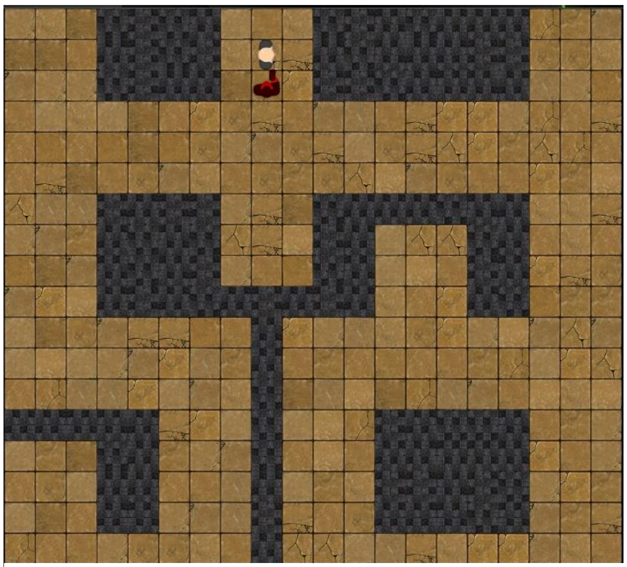
Figure 4 (Self Generated.)

What Did We Learn?

- A complete understanding of Pathfinding of the A* algorithm.
- Spawning in Game Object through getting information from Tiled.
- Understanding how we can make use of classes.
- Making use of the EasyStar.js Script.

Two questions to reflect on.

1. Which key skills did you learn the most from the project?
    a. If none, what did you hope/wish to learn in the future?
2. What are your future development plans after the project?

If there were any problems during the tutorial, please refer to "Where To Find Out More" for a link to a completed copy of the project.

Enjoy the Project? Try Taking These Challenges:

Improve your skills or expand the project by:

- Add a health system to the player.
- Add a bullet/ammo counter.
- Try adding in new weapons type the player could use during the game.
- Add in a scoring system where you earn points when defeating enemies.
- Try adding a wave system that makes the gameplay harder by featuring more enemies on the level.
- Add UI elements to tell how much health and ammo the player has.

Where To Find Out More:

Download a copy of the completed project here: LINK.

For more information on specific programming techniques:

1. Phaser 3 example – LINK – A large category of different examples of what has been made using Phaser 3. Some examples I recommend checking out is:
    a. Audio
    b. Physics/Matter
    c. Animation

2. Phaser API – LINK – A massive documentation on Phaser 3 that contains everything toward writing code around Phaser 3, ranging from “Namespaces”, “Classes”, “Events”, “Game Objects”, “Physics” and “Scene”.

3. W3 School Javascript – LINK – Phaser 3 uses Javascript, W3 School contain many Javascript writing tutorials.

Bibliography:

Neal, B (2022) EasyStar.js. https://easystarjs.com/ (Accessed 8 March 2022.)

Patel, A (1997) Introduction to A*. Available at: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html (Accessed 8 March 2022.)

Photon Storm LTD. (2021) Phaser 3 API Documentation. Available at: https://photonstorm.github.io/phaser3-docs/ (Accessed 8 March 2022.)

Photon Storm LTD. (2022) Phaser 3 Examples. Available at: http://phaser.io/examples (Accessed 8 March 2022.)

Piskel (2022) Free Online Sprite Editor. Available at: https://www.piskelapp.com/ (Accessed 8 March 2022.)

SRaven1994 (2022) pathfinding-template. Available at: https://github.com/SRaven1994/pathfinding-template (Accessed 8 March 2022.)

SRaven1994 (2022) pathfinding-template-complete. Available at: https://github.com/SRaven1994/pathfinding-completed-template (Accessed 8 March 2022.)

Tiled (2008) Flexible Level Editor. Available at: https://www.mapeditor.org/ (Accessed 8 March 2022.)

W3 Schools (1999) Javascript Tutorial. Available at: https://www.w3schools.com/js/ (Accessed 8 March 2022.)