ASP8 TOWER DEFENCE DOCUMENTATION

ASP8 Team members:

- Alex Chu
- Dimitri Vlachos
- Freda Xiaoyun Yu
- Jeremy Matthews

UNIVERSITY OF LONDON CM2020

Github route for codes: https://github.com/matthewsja/asp8_TowerDefense
Github route for files: https://github.com/FredaXYu/ASP_Group8
Link for out game: https://aspgrp8.z1.web.core.windows.net/
Link for final report:
https://github.com/FredaXYu/ASP_Group8/tree/main/after_midterm/finalterm_proposal
---

Table of Contents

audioManager

audioManager.mjs

The audio manager is used to manage the audio of the game. The music, SFX and so on.

Parameters

- *_soundManager* Phaser's default soundManager for the audioManager to manage.
- *_sfxEvents* A list of sfx 'events' to be called by the *playEffect()* function.
- *_musicEvents* A list of music 'events' to be called by the *playMusic()* function.
- *_musicSounds* A list of music 'sounds' to be called by the *playMusic()* function and looped.
- *_volume* (default: 1) The audible volume of all sounds.
- *_musicVolume* (default: 1) The audible volume of all music.
- *_sfxVolume* (default: 1) The audible volume of all sfx.
- *_currentMusic* (default: undefined) Music to be playing currently.

Functions

pause()

Pauses all sounds being played by the audioManager.

resume()

Resumes all sounds being played by the audioManager.

stopAll()

Stops all sounds being played by the audioManager.

*playMusic(*musicEvent*)*

Plays the *musicEvent*, adding it to the list *_musicEvents* if it is not already in it.

stopMusic()

Stops music from playing.

*playEffect(*sfxEvent*)*

Plays the *sfxEvent*, adding it to the list *_sfxEvents* if it is not already in it.

getVolume()

Returns the value of *_volume*.

*setVolume(*volume*)*

Sets the value of *_volume*.

getMusicVolume()

Returns the value of *_musicVolume*.

*setMusicVolume(*musicVolume*)*

Sets the value of *_musicVolume*.

getSfxVolume()

Returns the value of *_sfxVolume*.

*setSfxVolume(*sfxVolume*)*

Returns the value of *_sfxVolume*.

_getInstance()

Returns the *_instance*. Unless no instance exists, in which case it returns null.

Map Objects

enemy.js

This class extends the Phaser class Phaser.GameObjects.PathFollower, since enemies need to follow a settled path.

Functions

constructor (config)

Sets configurations for the enemy.

follow(self)

Let enemies follow the specified path.

destroyEnemy(self)

Remove the enemy when its Hit-Point is 0.

Play Objects

enemy.js

Same as mentioned above.

Below files, each is a class extended from Phaser.Scene, and each is set a keyword in the constructor with the format: `super('mapLogicTower');`

gameStats.js

It handles game statistics.

Create() method documentation:
https://newdocs.phaser.io/docs/3.55.2/Phaser.Types.Scenes.SceneCreateCallback

Within it, firstly reuse data from other scenes:

```
//these allow usage of attributes and functions from other scenes
        var gameStats = this.scene.get('gameStats')
        var resources = this.scene.get('resources')
```

Then import map data from resources scene:

```
//take the starting parameters from the resources scene and feed them to
//the code is a bit verbose, this shortens it for later use
        var params = resources.mapData['startParams']

        this.lives = params['lives']
        this.money = params['money']
        this.score = params['score']
        this.speedSetting = params['speedSetting']
        this.isPlaying = params['isPlaying']
```

Update() method documentatation:
https://newdocs.phaser.io/docs/3.55.2/Phaser.Scene#update

Within it, use attributes and functions from other scenes:

```
//these allow usage of attributes and functions from other scenes
        var map = this.scene.get('map')
        var gameStats = this.scene.get('gameStats')
        var gameRecords = this.scene.get('gameRecords')
```

Set the game over state when lives go to 0.

```
//in the case that the value of lives reaches 0, the game will change t
            if(this.lives <= 0){
                console.log('game over')
//record the current score to be displayed later
                gameRecords.score = gameStats.score
//call the function to update the high score if needed
                gameRecords.updateTopScore()
//record the number of waves survived to be displayed later
                gameRecords.wavesSurvived = map.origin.waveCounter
// this changes the scene
                map.timer = false
                this.scene.start('overState')
```

Set speed:

```
//change the speed at which different elements of the game run by mu
//as the whether the game is meant to be playing is set as a boolean
    this.playSpeed = this.speedSetting * this.isPlaying
```

hud.js

Describes functionalities in the User Interface when playing the game where the map displays.

Load data from other scenes:

```
//these allow usage of attributes and functions from other scenes
        var hud = this.scene.get('hud')
        var hudLogic = this.scene.get('hudLogic')
        var hudLogicTower = this.scene.get('hudLogicTower')
        var mapLogic = this.scene.get('mapLogic')
```

Import the tile logic which is how the map is built and is written in another file later:

```
//call function that creates the settings container at the start
//this is so the building site tiles could be correctly interacted with
    hudLogic.startHud()
```

Add the menu button at top right:

```
//create the image of the circle at the top right of the game window
    this.circle = this.add.image(850, 50, 'circle').setInteractive()

//when the circle is clicked on, the function that makes the settings is called
    this.circle.on('pointerdown', function(){
        hudLogic.makeSettings();
        AudioManager.playEffect(SFX.BUTTON_CLICK);
    })
```

Add the speed-changing button at the bottom:

```
//create the speed button at the bottom left of the game window
    this.speedButton  = this.add.image(50, 650, 'arrow1').setInteractive()

//when the speed button is clicked on, the speed of the game changes
    this.speedButton.on('pointerdown', function(){
        hudLogic.changeSpeed();
        mapLogic.updateSpeed();
        AudioManager.playEffect(SFX.BUTTON_CLICK);
    })
```

Within **update()** method, after loading data from other files, (1) add buy-tower functionality; (2) update different game statistics; (3) let the timer runs normally:

```
   update()
   {
//this allows usage of functions from other scenes
    var hudLogic = this.scene.get('hudLogic')
    var hudLogicTower = this.scene.get('hudLogicTower')

//call the function that gives the buy tower buttons a tint if the
    hudLogicTower.updateTint()
//call the function that displays different stats stored in the ga
    hudLogic.showStats()
//call the function that displays the countdown timer between wave
    hudLogic.showTimer()
```

hudLogic.js

Manipulates the logic of HUD elements.

Set menu with 3 buttons, save the current map to be resumed:

```
//this function creates the setting window which contains buttons
    this.makeSettings = function(){
//save the delay time of the origin and towers so they can resume
        if(map.origin.delay){

//ensure that most elements in the game stop while the menu is active
        gameStats.isPlaying = false
```

Then draw buttons and set interactions with specific outcomes. This step is difficult since it requires programmers to think about all the elements in each state comprehensively.

Then draw the speed-changing button and set options:

```
//function for when the speed button is clicked on
    this.changeSpeed = function(){
//uses a switch as the options are discrete
//the speed setting increases by one except for when it i
//these just draw the buttons and change the settings, th
        switch(gameStats.speedSetting){
            case 1:
```

Then show the game stats:

```
//display the stats of the current playthrough
    this.showStats = function(){
//strings that will contain the stats from the gameStats scene
        var wave = 'wave: ' + (map.origin.waveCounter+1)
        var lives = 'lives: ' + gameStats.lives
```

And show the timer:

```
//this function displays the timer for when a wave is finished
        this.showTimer = function(){
//remove any previous iterations of the display of these stats
            mapLogic.removePrev(hud.timerText)
//check if the timer is active
```

hudLogicTower.js

Mainly manipulates the towers shown in the HUD, i.e. buy tower, sell tower, tower upgrades, etc.

After reusing data from other files,

this.makeTowerMenu = function(tile)

Function that creates the tower menu. Takes in as a parameter the tile that was clicked on which caused the function to be called.

this.makeMenuBuy = function(tile)

Function that displays all the towers that could be bought. Takes in the tile the that was clicked on.

this.updateTint = function()

Function that would dim the buy button in the tower menu if the cost of the tower exceeds the amount of money at the moment. The button reverts back to normal when there is enough money.

this.clickBuy = function(button, tile, tower)

Function that allows the buy buttons in the tower menu to be interacted with interactions involve hovering the mouse over is and clicking it to buy the tower. Takes in as parameters 1)the button in question, 2)the tile the tower menu is for and 3)the tower the button is for.

this.makeMenuStats = function(tile)

Shows the stats of the tower in the building site that was clicked on. Takes as a parameter the tile that was clicked on.

this.removeTower = function(tile)

Removes a tower from building site. Parameter: the tile the tower is to be removed from.

map.js

Mainly summarises all of the interactions and logics done by other files.

Set all the towers, enemies, and bullets into groups:

```
//the group that would contain all the enemies
    this.enemyGroup = this.add.group()

//the group that would contain all the towers
    this.towerGroup = this.add.group()

//the group that would contain all the bullets
    this.bulletGroup = this.physics.add.group()

//the group that would contain all stationary targets for AOE bullets
    this.targetGroup = this.add.group()
```

Then import tile logic. Set enemies' path.

Then add the 'start' button and set interactions.

In **update()** method:

Update (1) speed changes; (2) remove bullets; (3) determine the attack speed of towers; (4) searches for any enemies that are in a tower's range; (5) checks the state of the origin and creates an enemy if the state is the right one; (6) determine how fast the enemies spawn in a wave; (7) handles the countdown between waves; (8) call the next next wave when the rush button is clicked; (9) when a building site tile is clicked on, it opens up the tower menu in the HUD.

mapLogic.js

Contains the functions and methods that are generally called by map.js.

extends from Phaser.Scene. Contains the functions and methods that are generally called by map.js.

After loading attributes and functions from other scenes, draw the map and set up the interactivity within it:

this.drawTiles = function(tileCoords, tileset, size)

Function for drawing the tileset to the map. Takes in as parameters 1)the array of indices indicating what tile goes where, 2)the tileset image and 3)the size of the tiles.

this.makePath = function(tiles, origin, destination, pathTiles, size)

Function for making the path which the enemies will travel on. Takes in as parameters 1)the array of indices showing what tiles are where, 2)the indices of the origin and 3)destination, 4)the array of indices indicating what tiles the enemies could travel on and 5)the size of the tiles in pixels.

Then find the coordinates in tiles units the origin and destination tiles.

Create a zone object which will have many parameters controlling the spawning process.

Create a physics image object which when enemies go over, a life is lost and the enemy is removed.

```
'/it has zero height and width as it isn't seen, just allows enemies to overlap with it
        map.dest = map.physics.add.image(destinationCoords[0]*size+size/2, destinationCoords
        map.dest.displayWidth = 0
        map.dest.displayHeight = 0
```

To find the path, the external library easystarjs is used to do an A* search.

this.findTile = function(tiles, type)

Function used to find a specific tile in a tile array. Takes in as parameters 1)the array of indices indicating where different tiles go and 2)the index of the tile to find.

General functions:

this.updateSpeed = function()

Function for changing the movement speed of the current enemies and bullets as well as the reload speed of waves and towers and the cooldown time of of the origin during play. Within it: update the movement speed of bullets; the movement speed of enemies; the cooldown speed of

the origin; the reload speed of the wave; the reload speed of the towers. Then add a ‘start’ button to let enemies in.

this.removePrev = function(object)

Function for removing the previous version of an object so there won't be multiple versions of the same object. Takes in as a parameter the object to attempt to remove.

mapLogicEnemy.js

Contains the functions and methods that are generally called by map.js relating to the enemy objects.

Functions

this.makeEnemy = function(key)

Function that makes the enemy object. Takes in as a parameter the key of enemy to make.

If the origin is ready to make another enemy: create a new enemy object; give it attributes for when it is moving and being attacked and its size; set how far the enemy has to travel; set its speed; set the HP bar (loads of codes); make the enemy follow the specific path at a specific speed; check if enemy reaches destination.

this.removeEnemy = function(enemy)

Function for removing the enemy. Parameter: the enemy to remove.

```
    this.removeEnemy = function(enemy){
//as the HP bar is made of graphics, its components need to be removed individually
        enemy.maxHPBar.destroy()
        enemy.HPBar.destroy()
//remove the enemy from the game
        enemy.destroy()
    }
```

this.debugEnemy = function(enemy)

Function for removing an enemy in case it is buggy. Parameter: the enemy to remove.

this.updateHPBar = function(enemy)

Function for updating the HP bar of an enemy. Parameter: the enemy whose HP bar to update.

this.startGame = function()

Function for starting the game proper by sending the first wave.

this.rushWave = function()

Function for starting the next wave prematurely when the rush button is clicked on while the countdown is active.

this.makeWave = function(waveData)

Function for changing the parameters of the origin so it is ready for another wave. Parameter: the data for the next wave.

this.updateWave = function()

Function for determining the speed at which the enemies are made. Also stops making enemies when enough have been made.

this.reloadWave = function()

Function for changing the state of the origin so more enemies could be made.

this.coolDown = function()

Function for seeing if the origin is ready for another wave and if it is, call a function to change the origin parameters to fit this state. Also determines if the winning conditions have been met in which case the game state will change to the game complete state.

this.callNextWave = function()

Function for calling the next wave.

this.enemyReachDest = function(enemy, dest)

Function for when an enemy reaches the destination. Parameters: 1)enemy; 2)destination.

mapLogicTower.js

Contains the functions and methods that are generally called by map.js relating to the tower objects.

Functions

this.clickSite = function()

Function for when a building site tile is clicked on.

this.makeTower = function(tile, key, size)

Used to create a tower at a specific point. Parameters: 1)the tile to make the tower in, 2)the key of the tower and 3)the size of the tiles.

this.clickTower = function(tower)

Function for when the tower is clicked on. Parameter: the tower that was clicked on.

this.searchEnemy = function(enemy)

Function for searching for an enemy within a tower's range. Parameter: the enemy being searched.

this.enemyFound = function(enemy, tower)

Function for when an enemy has been found. Parameters: 1)the enemy that was found and 2)the tower that found it.

this.makeBullet = function(enemy, tower)

Making the bullet. Takes in as parameters 1)the enemy being targeted and 2)the tower that the bullet originated from.

this.updateTower = function(tower)

Regulating the attack speed of the tower. Takes in as a parameter the tower being affected.

this.reloadTower = function(tower)

Changing the state attribute of a tower so it may fire another bullet. Parameter: the tower being affected.

this.moveBullet = function(bullet)

Moving the bullet. Takes in as a parameter, the bullet that is affected.

this.hitTarget = function(bullet, target)

Function for when the bullet reaches its target. Parameters: 1)the bullet and 2)the target.

this.damageEnemy = function(bullet, enemy)

Responsible for damaging the enemy. Parameters: 1)the bullet that will damage the enemy and 2)enemy itself.

this.updateBullet = function(bullet)

Function for when a target enemy is destroyed before a targeting bullet has made contact with it. Takes in as a parameter the affected bullet.

states

Each file is one class which is extended from Phaser.Scene, and each is set a keyword in the constructor with the format: `super('bootstrapState');`

bootstrap.js

Handles what happens when the winning conditions for the game are met. This file loads and initialises all of the Phaser objects for the audio manager in order to facilitate testing as the mocha.js testing suite crashes when it tried to load Phasers objects, meaning we had to remove this section from that code in order to run our tests.

First we import Phaser and the audioManager itself:

```
import Phaser from 'phaser';
import AudioManager, { MUSIC } from '../audiomanager/audioManager.mjs';
```

Then we launch all of our scenes necessary to run the game:

```
this.scene.launch('resources')
this.scene.launch('checkJSON')
this.scene.launch('gameRecords')
this.scene.launch('mapLogic')
this.scene.launch('mapLogicEnemy')
this.scene.launch('mapLogicTower')
this.scene.launch('hudLogic')
this.scene.launch('hudLogicTower')
```

Then we create our scene loader/switcher, allowing us to change between scenes:

```
update(time, delta) {
    if (this._loaded && this._elapsedTime > MIN_DISPLAY_TIME)
    {
        this._switchScenes();
    }
    this._elapsedTime += delta;
}

_switchScenes() {
    console.log('Loading Complete... Switching to Menu State');
    this._progressBox.destroy();
    this._progressBar.destroy();
    this.scene.stop('bootstrapState');
    this.scene.start('menuState');
    AudioManager.playMusic(MUSIC.MAIN);
}
```

complete.js

Handles what happens when the winning conditions for the game are met.

First we stop the other scenes to prevent them from overlaying this one:

```
this.scene.stop('gameStats')
this.scene.stop('hud')
this.scene.stop('map')
```

Load the gameRecords:

```
var gameRecords = this.scene.get('gameRecords');
```

Create new variables and show them:

```
//this string will be displayed to show the score from the current playthroug
    var score = 'Score: ' + gameRecords.score;
//this string will be displayed to show the highest score from the current pl
    var topScore = 'Top Score: ' + gameRecords.topScore;
//this string will be displayed to show how many lives remain from the currer
    var lives = 'Lives Left: ' + gameRecords.lives;
```

Add logic to trigger the 'happy ending' page if the condition is 'instant win'.

Display:

```
//display the previously made strings giving a new line to each
this.endResults = this.add.text(textX, textY, score + '\n' + topScore + '\n'
```

Create 3 buttons: 'menu', 'level', 'restart'.

level.js

This is the level selection scene for selecting levels.

We close other scenes to prevent them from overlaying the menu:

```
this.scene.stop('gameStats')
this.scene.stop('hud')
this.scene.stop('map')
```

Add a title bar:

```
this.bar = this.add.image(450, 100, 'level_selection_bar')
```

Create the 'back' button at top left:

```
//creates an image that could be clicked on
    var back = this.add.image(50, 50, 'left').setInteractive()
//when the image is clicked on the game would be the menu state
    back.on('pointerdown', function () {
        this.scene.scene.start('treasureState');
        AudioManager.playEffect(SFX.BUTTON_CLICK);
```

Then add the functionality:

this.levelButtons = function()

This function positions the buttons which when clicked on would take the player to the level that was clicked on. Create a for loop to build level buttons.

this.makeButton = function(level, x, y)

This function is used to make a button for each level that could be played. Takes in as parameters 1)the key of the level and the 2)x and 3)y coordinates of where the button would be located.

this.clickButton = function(level)

Function to handle what happens when a level button is clicked on. Takes in as a parameter the name of the level.

Call the functions.

menu.js

Handles the main menu of the game.

First we stop other game scenes.

Add background image:

```
this.background_jungle = this.add.image(450, 350, 'background_jungle')
```

Then we create the 'start' button at bottom, and set interactions and sounds:

```
//create an image that does something when clicked on
        var start = this.add.image(450, 620, 'start').setInteractive()
        start.displayWidth = 200
        start.displayHeight = 160

//when the image is clicked on, the scene changes to the 'treasureState' sce
//code to change scenes from https://www.thepolyglotdeveloper.com/2020/09/sw
        start.on('pointerdown', function () {
            // Please debug using the below sentence. Try change it btw 'lev
            this.scene.scene.start('treasureState');  //changed by XYu Mar5.
            AudioManager.playEffect(SFX.BUTTON_CLICK);
        })
```

over.js

Handles what happens when the defeat conditions for the game are met.

After stopping other scenes, we display the results of the game:

```
//display the previously made strings giving a new line to each
        this.endResults = this.add.text(350, 0, 'Game Over \n' + score + '\n' +
        topScore + '\n' + waves, { font: '32px Arial' })
```

We then display 3 buttons:

```
//display some images, make them interactable and certain sizes
        var menu = this.add.image(450, 250, 'menu').setInteractive()
        menu.displayWidth = 200
        menu.displayHeight = 100
        var level = this.add.image(450, 400, 'level').setInteractive()
        level.displayWidth = 200
        level.displayHeight = 100
        var restart = this.add.image(450, 550, 'restart').setInteractive()
        restart.displayWidth = 200
        restart.displayHeight = 100
```

And set interactions and sounds.

play.js

Handles what happens when the game is being played.

Start all the scenes that make the game up itself:

```
this.scene.launch('gameStats')
this.scene.launch('map')
this.scene.launch('hud')
var gameRecords = this.scene.get('gameRecords');
```

Make sure 'instant win'= False, since users haven't set 'the Law' yet:

```
gameRecords.instantWin = false;
```

treasure.js

This is the 'treasure box' page which shows lists of characters.

Firstly, stop interactions from other pages:

```
    this.scene.stop('gameStats')
    this.scene.stop('hud')
    this.scene.stop('map')
```

Then, add the background picture from the resource.js:

```
this.background_treasure = this.add.image(450, 350, 'treasure_bg')
```

Create a 'start' button and set interactions.

utilities

checkJSON.js

This utility extends Phaser.io's default Phaser.Scene. It allows for the application to validate JSON files, ensuring their integrity and compatibility with the application.

Functions

checkDict(dict, keys)

Checks that every relevant value of a given dictionary is the right data type.

It takes, as input:

- *dict*, the dictionary to check.
- *keys*, a second dictionary that contains the data types for all of the relevant values for any dictionary used in the game. Only those covered by the key dictionary are checked.

checkLevel(dict, data, key)

This function is used specifically for checking the inner arrays of the level dictionaries.

It takes, as input:

- *dict*, the dictionary to add the data to.
- *data*, the dictionary containing all of the data to check.

- *key*, the key of the dictionary to check.

By default, all dictionaries pass, therefore, the data may be added to the larger level dictionary. If any of the tests fail, then it will not be added to the larger level dictionary.

gameRecords.js

This utility extends Phaser.io's default Phaser.Scene. It allows for levels to be selected

Parameters

- *levelSelect* (default: -1) This is how the levels are selected and the selection is preserved.
- *wavesSurvived* This records how many waves the player has survived.
- *lives* This records how many lives the player has remaining.
- *score* This records the highest score the player has attained during the current game session.
- *topScore* (default: 0) This records the highest score the player has attained during the current game session.
- *instantWin* (default: false)

Functions

updateTopScore()

Updates the top score by comparing it to the current score. If the current score is higher, then the top score is changed to the value of the current score.

resources.js

This utility pre-loads all of the necessary assets for the game.