Delta-Zone Game Summary

Delta-Zone is a fast paced, top-down 2D arcade game where the player plays the part of a lone US secret agent battling waves of bomb-loaded undead with nothing but his trusty fully-automatic pistol.

The rules are simple:

* Stay alive by killing the zombies before they get too close and explode their bombs
  + Killing a zombie grants you 100 points
  + Being hit by a zombie however makes you lose 10 Hit Points/100
  + It takes 3 hits to knock an enemy down, so make sure to give yourself enough time to kill them!
* Staying alive can be tricky, so you’ve been provided with 3 health pills to keep you going. Use these sparingly as they’re all you’ve got!
* Watch out, things get harder after wave 10, where now it’ll take 4 shots per zombie, and after wave 15 they’ll be even faster!
* Surviving a round grants you 500 points. If you’re up for the challenging hard mode, then just remember you’ll be missing out on these bonus points, but maybe reaching higher waves!
* Use the walls to trap your enemies to lower the number of zombies chasing you, or just to take a little break.
* Don’t let your health reach 0, else that’s the end of the road for you this time, but don’t fret your high score will be saved, so you can keep going to reach a new high score!

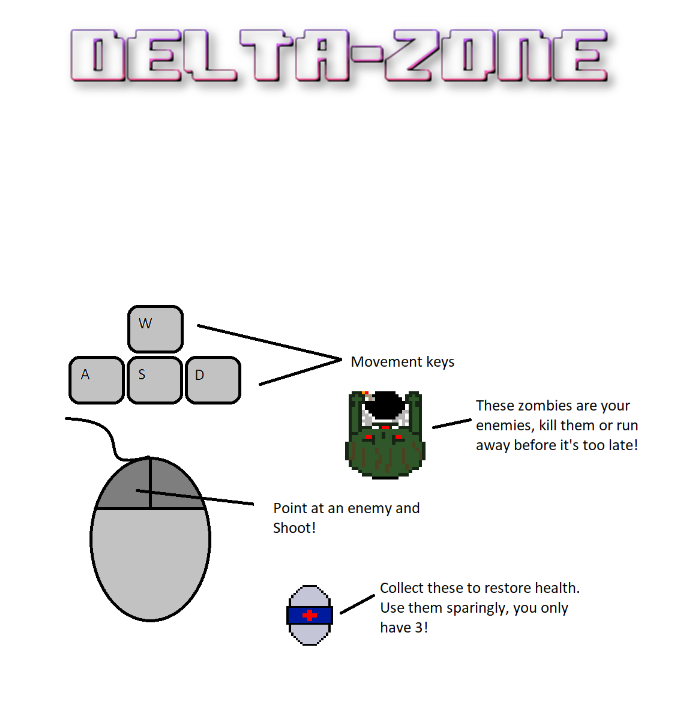
Control your agent using your mouse to aim and shoot, and keyboard to move out of the way. You can also play on your mobile device, though movement has been disabled for the time being.

Game Structure

******Preload Screen:**

Loading bar

Delta-Zone logo

**Main Menu Screen:**

Instructions

Delta-Zone logo

Difficulty settings

Local user high score

Instructions overlay button

Delta-Zone logo

**Instructions Overlay:**

**Main Game:**

Number of enemies remaining/wave

Current wave number

Current user score

****

Zombies (Enemy)

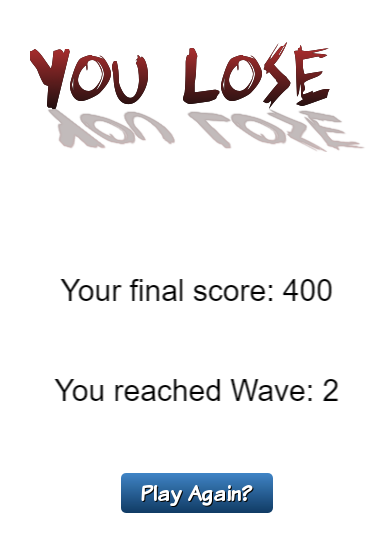
Health Pill

Wall

Health Bar

Player

**Lose Screen:**

****

“You Lose” icon

Final score upon death

Waves reached upon death

“Play again?” button, returns to main menu giving opportunity to save score with name if exceeds high score

Implementation Specification

The main data structures used in this game are made using Phaser’s Group and Sprite structures. The Groups were important for storing numerous amounts of Sprite or Objects, as they acted as the container to hold all of the Sprites, in this case my Enemies, Bullets and Pills. What’s great about using Phaser Groups for this is that they have been given support for fast pooling and object recycling, so performance is maximised for the recycling of Sprites, especially bullets in my case which fire at a rapid pace and are then destroyed and reused either on collision or on out of bouts detection.

The different stages in the game (Boot, Preload, Main Menu, Game, Lose) are stored as states, and are controlled using the Phaser state manager to conditionally switch between states.

The functions of each JavaScript file will be listed below, alongside their line number in the code for reference.

**Main.js**

Whilst this file does not contain any functions, it is one of the most important files as it is in charge of not only creating a new game, but also creating all the states and adding them to the game’s state library for use by the state manager. Once this is complete, the Boot state is started.

**Boot.js – Boot State**

The Boot State uses two functions, preload() and create().

*preload() – Line 6*

This function takes no parameters.

The preload function loads 2 assets for use in the loading screen, the logo and the preload bar. Only these two are loaded here so that they will be visible whilst the Preload State loads all the other assets for the game.

*create() – Line 11*

This function takes no parameters.

The create function is responsible for initialising the primary settings for the game. In this case it sets the default background colour, sets scaling to ‘SHOW\_ALL’, centres the game in the page, updates the screen size, and initialises the physics system before moving on to the Preload State.

**Preload.js – Preload State**

The Preload State uses two functions, preload() and create().

*preload() – Line 6*

This function takes no parameters.

The first thing this function does is create sprites from 2 assets loaded in the boot stage, and pastes them at the top and bottom of the screen, using the Phaser function ‘game.load.setPreloadSprite’ to make the preload bar act properly when loading the assets. After this the map, all the sprite assets, spritesheets and audio are loaded and given a name ready for use in the other states.

*create() – Line 39*

This function takes no parameters.

After the preload() function has completed, the create() function is run and moves directly to the MainMenu State.

**MainMenu.js – MainMenu State**

The MainMenu State uses 3 functions, create(), init(score), and update().

*create() – Line 11*

This function takes no parameters.

Firstly the create function resets the world size, as originally it would be much larger after the game had run once. Next:

* Loads logo sprite.
* Initialises menu music.
* Add invisible instructions sprite (visible when instructions button is pressed).
* Adds buttons for Easy, Medium, Hard and Instructions. Instructions button is created last so that it can disabled the other buttons whilst the instructions sprite is shown, otherwise they could be pressed when in the background.
* Adds the highest user score, with data retrieved from locaStorage.

*init(score) – Line 66*

This function takes a score (integer) parameter passed from the Lose State.

This function sets the score to 0 or the score from the last game, then retrieves the stores highest score (if there is no highest score, it is set to 0). Then it compares the two, and if the new score is higher than the highest score, it calls up a prompt to take the player’s name and saves both the name and the score to localStorage.

*update() – Line 79*

This function takes no parameters.

This function checks if the pointer has be pressed, and if so makes the instructions sprite invisible, and re-enables all the buttons. This is primarily used to hide the instructions sprite.

**Game.js – Game State**

The Game State uses 16 functions, listed below in order as they appear.

*create() – Line 23*

This function takes no parameters.

This function is in charge of creating the starting components of the game:

* Adds the map to the game and creates its layers, setting layer collision and resizing the world.
* Creates the Objects, Bullets, Player and Enemies by calling their respective functions.
* Adds the UI Health Bar, and all UI text components including score, wave number, and wave enemies.
* Creates local variable names and adds the bullet spawn point to the player.
* Initialises game music audio

*update() – Line 94*

This function takes no parameters.

This function is called every frame, and checks important components of the game:

* Updates local variables (for example the player).
* Resets player movement to nullify any acceleration.
* Makes the player face the pointer.
* Fires the gun if the pointer is pressed (for touch screen also controls movement.
* Detects keyboard movement and moves player.
* Controls enemy AI movement.
* Checks for any collisions, and calls the correct functions to handle them.

*createObjects() – Line 144*

This function takes no parameters.

This function creates a group, then creates and positions all pills on the map before being added to the group for easy manipulation.

*collectPill(player, pill) – Line 155*

This function takes the first parameter of the overlap check, in this case the player, and the pill it has overlapped with, passed through using ‘this’.

This function handles the overlap between the player and the collectible pills, by resetting the player’s health to 100, updating the UI health bar, playing the collection audio and killing the Pill.

*enemyCollide(player, enemy) – Line 161*

This function takes the first parameter of the collision check, in this case the player, and the enemy it has collided with, passed through using ‘this’.

This function handles the collision between the player and an enemy, by:

* Reducing player health and score by 10.
* Checking to see whether the player has died (if health = 0, start Lose State).
* Playing explosion audio.
* Destroying the enemy
* Updating the score, active wave enemies, and player health.
* Checking to see whether the wave has ended, and if so start a new wave given set parameters (including increasing difficulty of enemies after certain waves).

*killEnemy(bullet, enemy) – Line 199*

This function takes the first parameter of the collision check, in this case a bullet, and the enemy it has collided with, passed through using ‘this’.

This function handles the collision between a bullet and an enemy, by:

* Reducing enemy health by 20.
* Checking to see whether the enemy has died (if health = 0, play splat sound, explode and destroy enemy, reduce enemy count, and update score and enemy count).
* Checking to see whether the wave has ended, and if so start a new wave given set parameters (including increasing difficulty of enemies after certain waves).
* Kill the bullet.

*killBullet(bullet, wall) – Line 231*

This function takes the first parameter of the collision check, in this case a bullet, and the wall layer it has collided with, passed through using ‘this’.

This function handles the collision between a bullet and a wall by killing the bullet.

*createPlayer() – Line 234*

This function takes no parameters.

This function is in charge of creating the player in the game.

* Creates the player sprite and adds it to the world, enabling physics and enabling collision with world bounds.
* Sets the anchor and body rotation of the player.
* Sets the player health to 100.
* Tells the camera to follow the player.
* Creates the cursor keys for the game.

*createBullets() – Line 248*

This function takes no parameters.

This function is in charge of creating all the bullets to be used in the game. Firstly a new physics group is made, which is then given a body with ARCADE type physics. Then using Phaser’s ‘createMultiple’ function I create 50 bullets within the group, and set all of them to be killed if they go out of bounds, and set their anchors.

*createEnemies(n) – Line 259*

This function takes one parameter, n (Integer), which is the number of enemies to create.

This function is in charge of creating new enemies every wave. Firstly a group is added to the game to hold all the enemies, then a ‘for’ loop is run up till the number of enemies (n) has been reached. In this ‘for’ loop a random x and y coordinate is generated within the world bounds, and a new enemy is created at these random coordinates. Then physics are enables, and the enemies are added to the group. After the loop, the enemies are given health and speed properties.

*fire() – Line 278*

This function takes no parameters.

This function fires a bullets from the gun. First an ‘if’ statement is run to compare the time with the nextFire variable to see whether the player is allowed to fire yet, and if it is:

* Play shooting sound.
* Update nextFire variable.
* Get the first dead bullet and reset its location to the spawn point at the end of the player’s gun.
* Rotate the bullet to the pointer, then fire the bullet towards the pointer.

*explode(target) – Line 290*

This function takes a target parameter which acts as the centre point to start the emitter.

This function:

* Creates an emitter in the centre of the target.
* Makes particles from the emitter from a single pixel particle sprite.
* Sets the minimum and maximum speed of the particles.
* Sets gravity to 0.
* Starts the emitter to create 100 particles from the centre of the target in ¼ of a second.
* After ¼ of a second, destroy the emitter.

**The last bullet point is very important. When I first started using an emitter, I didn’t realise that by default the particles are ‘killed’ rather than ‘destroyed’, which meant that when the player got to high levels, and therefore higher numbers of enemies, there was considerable lag on the system as it was constantly adding more dead particles to the field but never actually disposing of them.**

*updateScore() – Line 301*

This function takes no parameters.

This function updates the UI score to the current running score.

*updateWaveN() – Line 304*

This function takes no parameters.

This function updates the UI wave number to the current running wave number.

*updateWaveE() – Line 307*

This function takes no parameters.

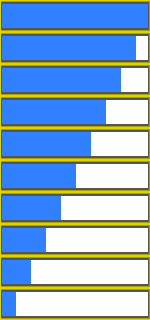
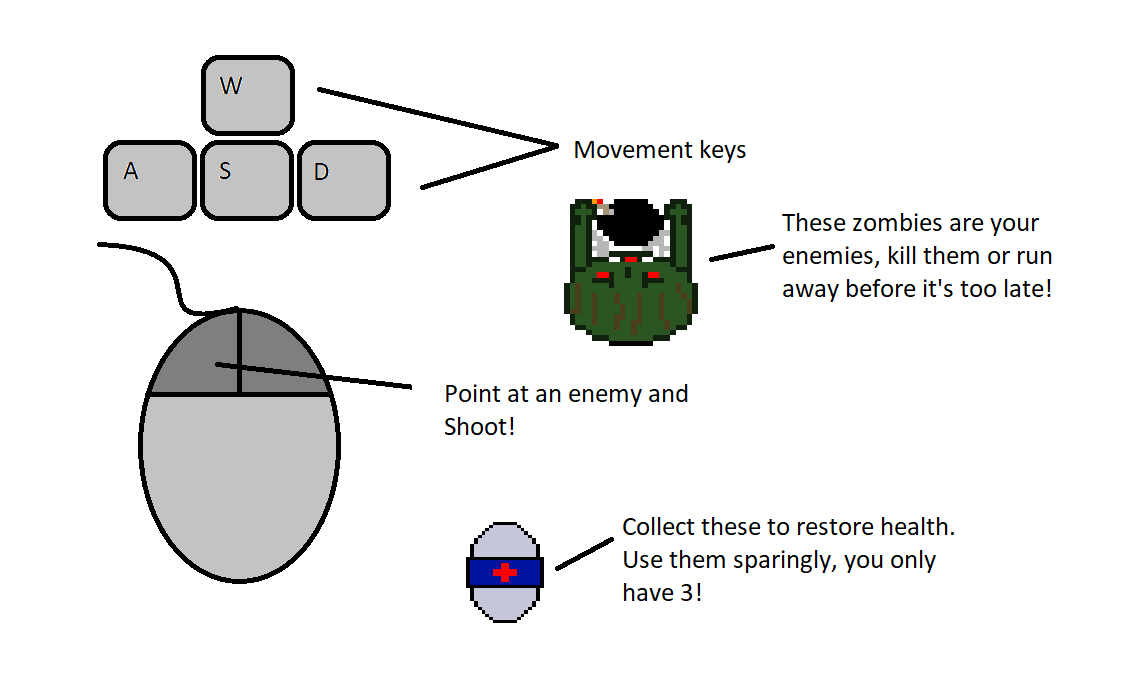
This function updates the UI enemy count to the current running enemy count.

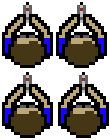
*updateHealth() – Line 310*

This function takes no parameters.

This function updates the UI health bar to the corresponding frame depending on the health of the player.

# Assets

Many of the visual assets used in the game were made by myself using an online editor called piskelapp. All of the sprites shown below were created using single frames in piskelapp, and spritesheets made using multiple frames exported as spritesheets. They originally began as 32x32 images, however the widths and heights were modified in order to clear out as much alpha as possible. One thing to note however is that I originally made the player sprite to be a spritesheet where the player moves from side to side, however it looked a bit strange when put into practice, so the decision was made just to use it as a normal sprite stuck to one frame.



Other than these assets, the audio in the game was acquired from royalty free services Zapsplat and TeknoAXE; The buttons were made using an online html button maker, then appended into a spritesheet using an online spritesheet maker; Logo and Lose screen were made using an online text logo maker; Map tiles were acquired from an open source website with permission from the creator, all of which have been referenced in the references.

# Critical review

## Good

* Ordering of states allows for a proper flow of data throughout the game, which helps it run smoothly.
* Many functions involving updates to UI were instead run within collision handlers to prevent the function calls to be taking place constantly when there has been no change to data. For instance, there is no need to update the score every frame, when it only changes upon killing an enemy or colliding with an enemy. With the collision check running every frame anyway, it’s better to update the UI when there has actually been a collision rather than have it checked every frame.
* The recycling of bullets was emphasized so that they only need to be created once and then repositioned and spawned when required again, rather than creating a new sprite every shot which simply wastes cpu time, and may hinder performance on mobile devices.

## Improvements

* Whilst the game does run on a mobile device, movement can be a little frustrating, as the player always moves the opposite way to which he is shooting (inability to strafe) and must be shooting to move. Creating a joystick and fire button would be a way around this, though some optimisation would have to be made to not hinder performance.
* Even though much of the code has been moved around to increase performance, there is still some lag when played on mobile devices, so potentially disabling the particle emitter or spawning less enemies if on a mobile device could be a way around this.
* Currently the highest score is saved using localStorage. Whilst this does work for keeping personal scores within a user’s browser, it would be nicer if the information were saved to a file/database server side so that the highscore is seen by anyone who accesses the game.

References

* Lecture Slides by Robin Heath:
  + Week 5 – Physics
  + Week 7 – Phaser Physics
  + Week 9 – Tile Based Worlds 2
* Game templating: Farias Navarro, Pablo (Unknown). *HTML5 Phaser Tutorial - SpaceHipster, A Space Exploration Game*. Zenva, GameDevAcademy. Retrieved from <https://gamedevacademy.org/html5-phaser-tutorial-spacehipster-a-space-exploration-game/>
* Menhennett, Ashley (2014, June 16). *Building HTML5 Mobile Apps with Local Storage and Topcoat*. binpress. Retrieved from <http://www.binpress.com/tutorial/local-storage/106>
* Phaser 2.6.2 API Documentation. Retrieved from <https://phaser.io/docs/2.6.2/index>
* Phaser Docs Search. Retrieved from <https://phaser.io/learn/chains>

### Assets

* Button maker: Da Button Factory. Retrieved from <https://dabuttonfactory.com/>
* Logo maker: flamingtext. Retrieved from <http://eu2.flamingtext.com/>
* Spritesheet maker: ZeroSprites. Retrieved from <http://zerosprites.com/>
* Pixel editor: Piskelapp. Retrieved from <https://www.piskelapp.com/>
* Tiled. Retrieved from <http://www.mapeditor.org/>
* Brackets. Retrieved from <http://brackets.io/>
* Map tiles: Hyptosis (2012, December 2). *Lots of free 2D tiles and sprites by Hyptosis*. OpenGameArt. Retrieved from <https://opengameart.org/content/lots-of-free-2d-tiles-and-sprites-by-hyptosis>
* Chiptune throne Room by TeknoAXE under a Creative Commons Attribution 4.0 International License. Retrieved from <http://teknoaxe.com/Link_Code_4.php?q=1202&Genre=Eight>
* Crimson Nights Track 02 by TeknoAXE under a Creative Commons Attribution 4.0 International License. Retrieved from <http://teknoaxe.com/Link_Code_4.php?q=1086&Genre=Eight>
* Body impact, splat by Blastwave FX under a ZapSplat Standard License. Retrieved from <https://www.zapsplat.com/music/body-impact-splat/>
* Retro 8-bit game explode or lose life tone by ZapSplat under a ZapSplat Standard License. Retrieved from <https://www.zapsplat.com/music/retro-8-bit-game-explode-or-lose-life-tone/>
* Retro 8-bit game, pickup or collect object 04 by Little Robot Sound Factory under a Creative Commons Attribution 4.0 International License. Retrieved from <https://www.zapsplat.com/music/retro-8-bit-game-pickup-or-collect-object-04/>
* Retro 8-bit game, shoot 01 by Little Robot Sound Factory under a Creative Commons Attribution 4.0 International License. Retrieved from <https://www.zapsplat.com/music/retro-8-bit-game-shoot-01/>