Kloster Software Major Part B

Software Specifications

- OS: Windows 7 SP1+, macOS 10.11+, Ubuntu 12.04+, SteamOS+
- Graphics card with DX10 (shader model 4.0) capabilities.
- CPU: SSE2 instruction set support.
- iOS player requires iOS 8.0 or higher.
- Android: OS 4.1 or later; ARMv7 CPU with NEON support or Atom CPU; OpenGL ES 2.0 or later.
- WebGL: Any recent desktop version of Firefox, Chrome, Edge or Safari.
- Universal Windows Platform: Windows 10 and a graphics card with DX10 (shader model 4.0) capabilities

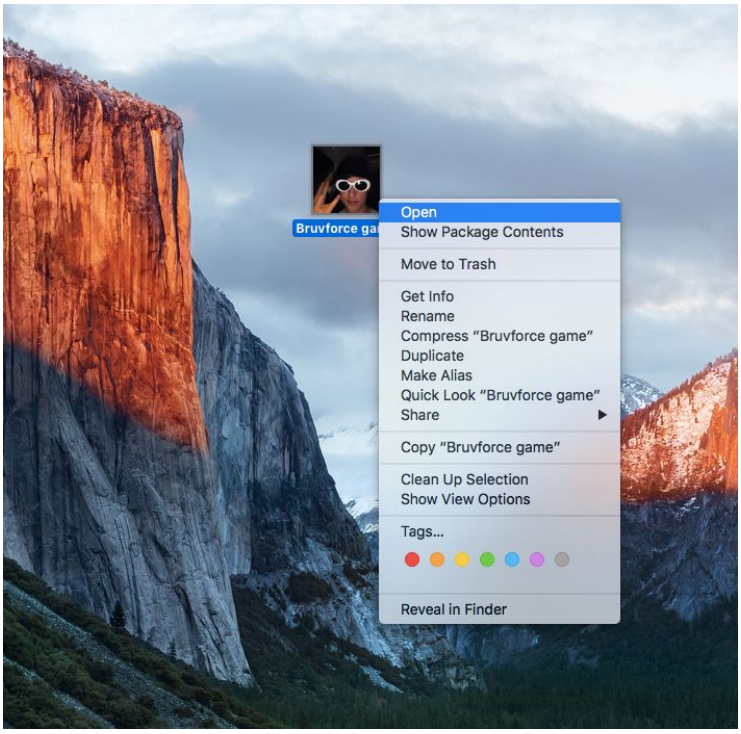
Hardware Specifications

- Processor: Intel Core 2 Duo or above
- Memory: 1 GB RAM
- Graphics: Integrated Graphics from Intel CPU
- Storage: 100 MB available space
- Keyboard
- Mouse

User Manual

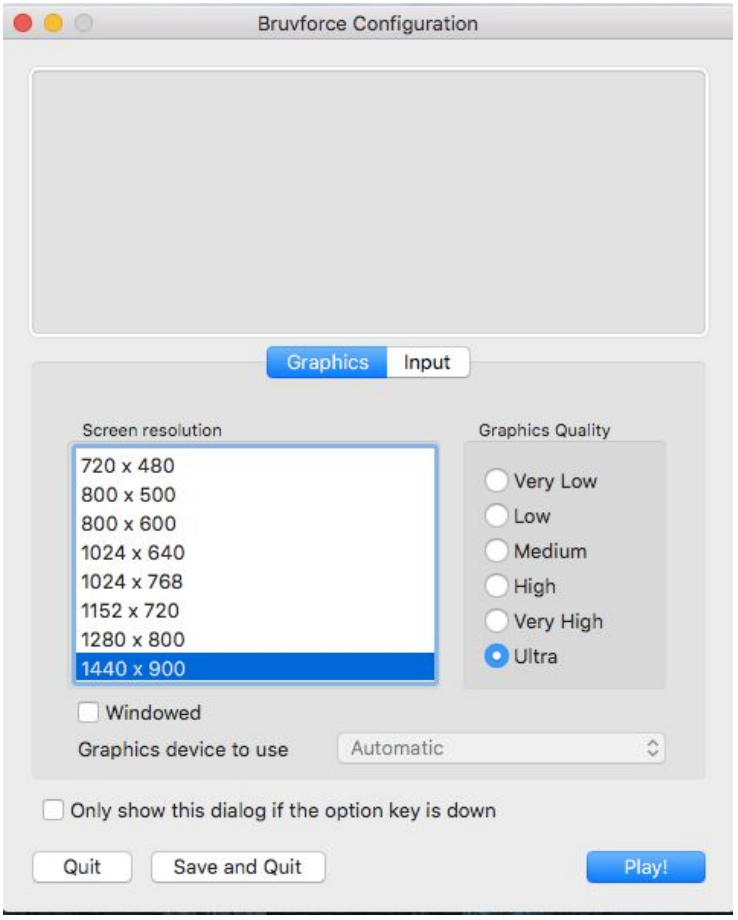
Installation Instructions:

Upon Downloading the submission, open up the “Bruvforce game” app. No other installation is required and the game will run as intended from there.



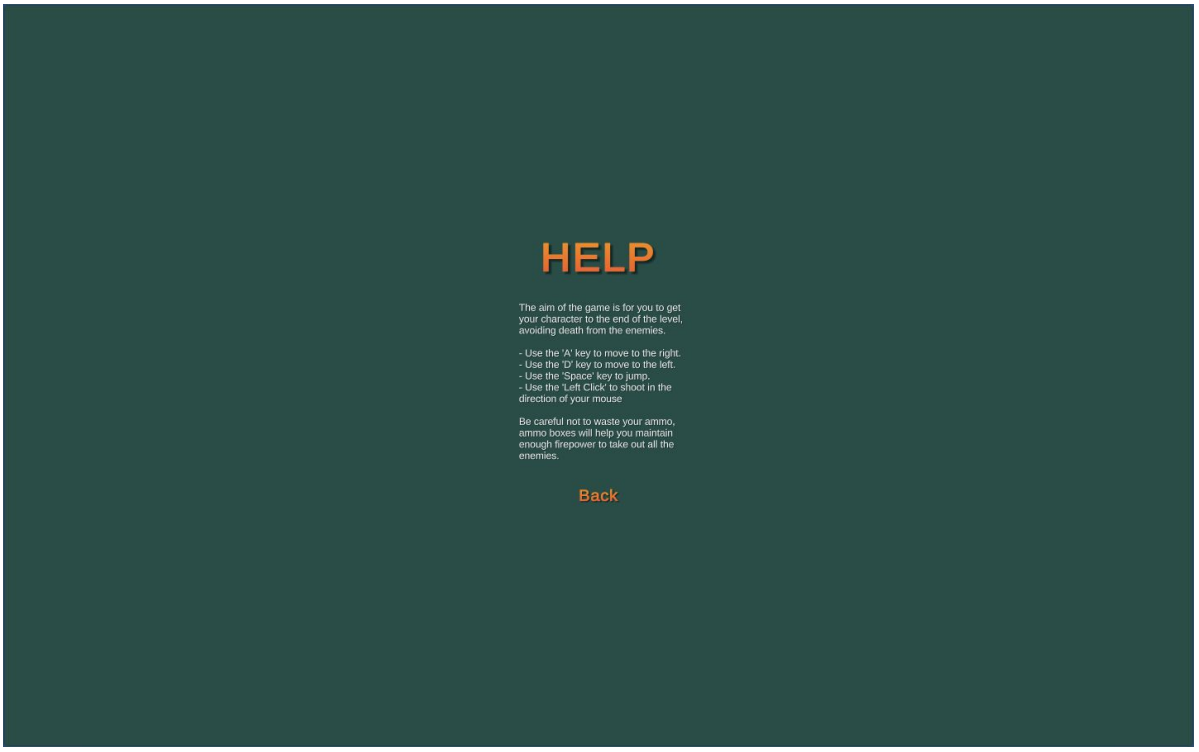
User Processes:

Upon opening Bruvforce, the user will be presented with a screen outlying graphics settings and input options, it is best to leave this as default or to check the windowed mode for multitasking. Press ‘Play!’ to continue.

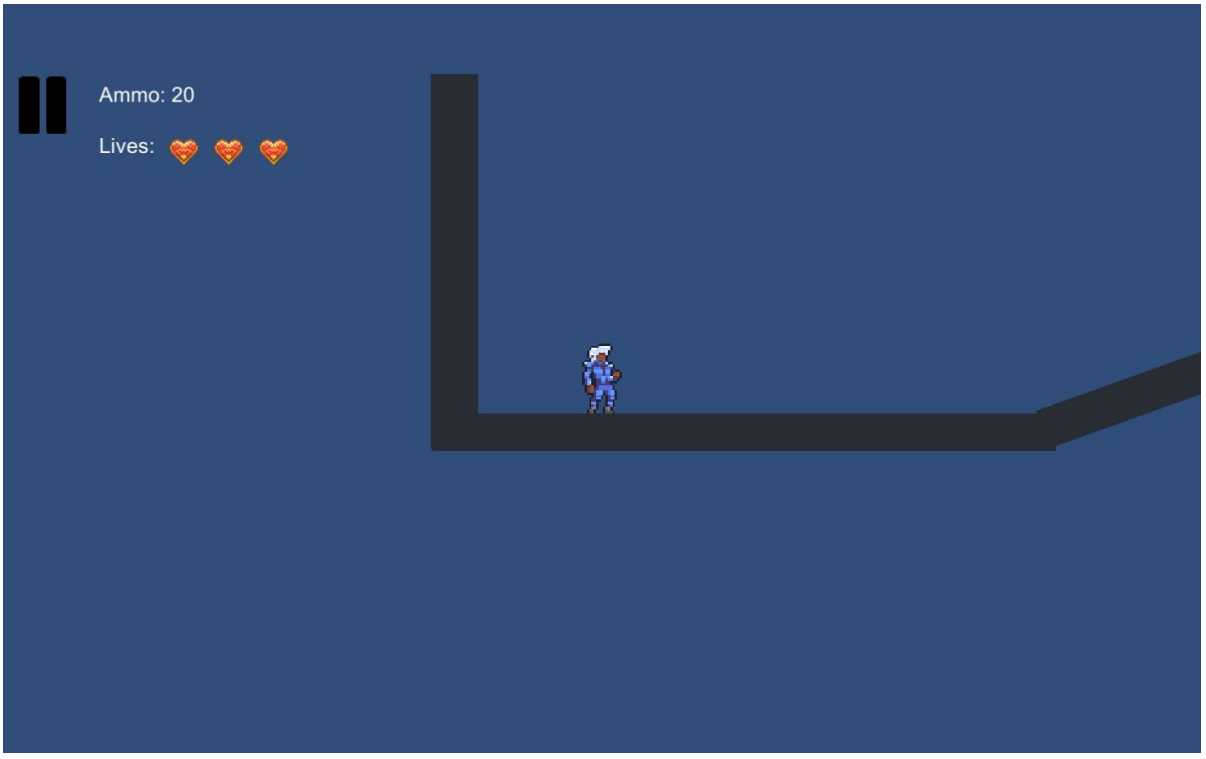


Once the user has selected ‘Play!’ they will be taken to the main menu, here they can access help, start the game or quit the app.

The help screen outlines all the most important aspects of the game, once the user has read everything, they can click ‘Back’ to be returned to the main menu.

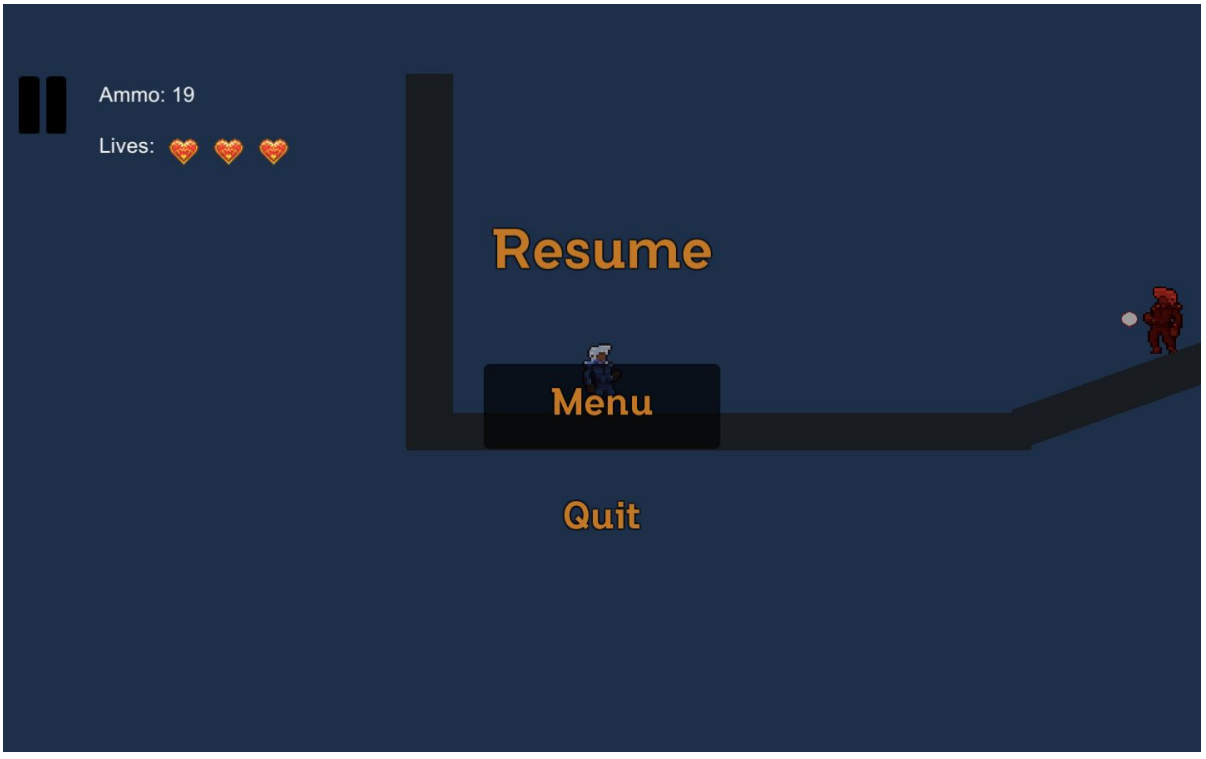


Once the user has pressed play, they are put into the level. Pressing the ‘A’ and ‘D’ keys for left and right movement along with ‘Space’ for jumping and a ‘Left Click’ for shooting are all the controls they must know.



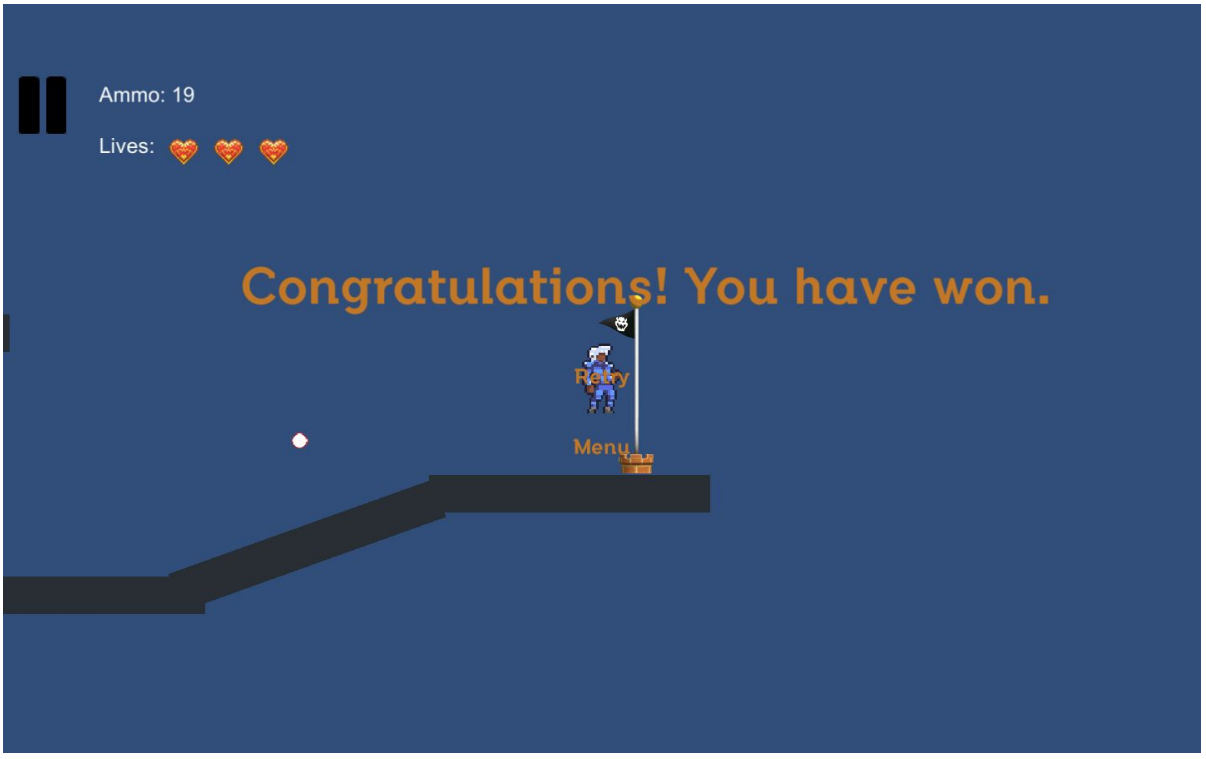
The basic UI shows the player their remaining ammo, their remaining lives and also a pause button (The ‘Escape’ key can be used for pausing but is not required as the user can simply click the button).

The Pause button gives the user three options, resume the game, go to the main menu or quit the application. Each of these buttons can be clicked to perform their respective tasks.

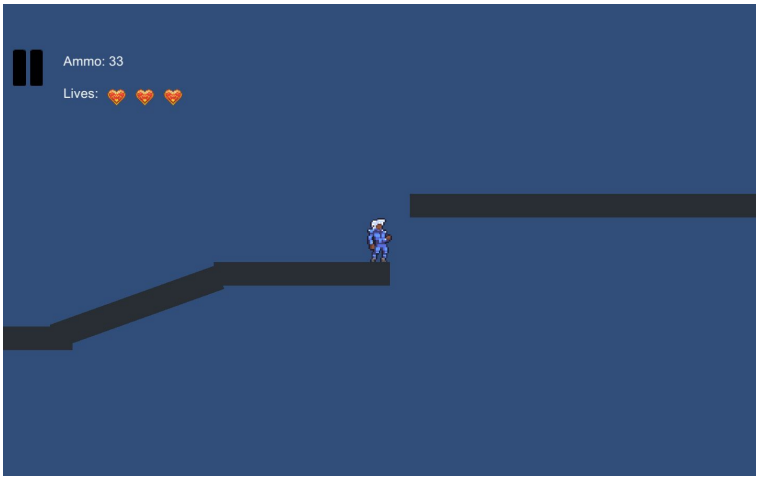


Upon death, the user is given only two choices, retry the level or return to the main menu. Each of these buttons may be clicked to perform their respective tasks.

When the user reaches the flag at the end of the level, the game is won. Again the user is given only two choices, retry the level or return to the main menu. Each of these buttons may be clicked to perform their respective tasks.



Troubleshooting:



A tricky jump for all but the most skilled of Bruvforce players. Careful timing and/or spam clicking of the jump button is required to achieve this feat. Do not despair if at first it seems impossible as it can indeed be conquered.

Test Data

Ammo Test

```
public int ammo = 20;
```

```
if (Input.GetMouseButtonDown(0) && ammo >= 1 && canmove == true){
  if ( ! EventSystem.current.IsPointerOverGameObject())
  {
    tempbullet = Instantiate(bullet, bulletspawn.position, Quaternion.identity);
    ammo--;
    setAmmoText();
    Destroy(tempbullet, 10f);
  }
}
```

Changing the ammo variable to values such as 1, 0 and 2 helped me test the boundaries of this algorithm and test what would happen when attempting to shoot with different amounts of ammo. This showed that my algorithm worked as intended as the shooting didn't work when the ammo count was 0 but it did shoot when the ammo count was 1 and 2.

Enemy Health Test

```
public int enemyHealth = 10;
```

```
void OnTriggerEnter2D(Collider2D collision)
{
  if (collision.gameObject.tag == "bullet"){
    enemyHealth -= 5;
  }
}
```

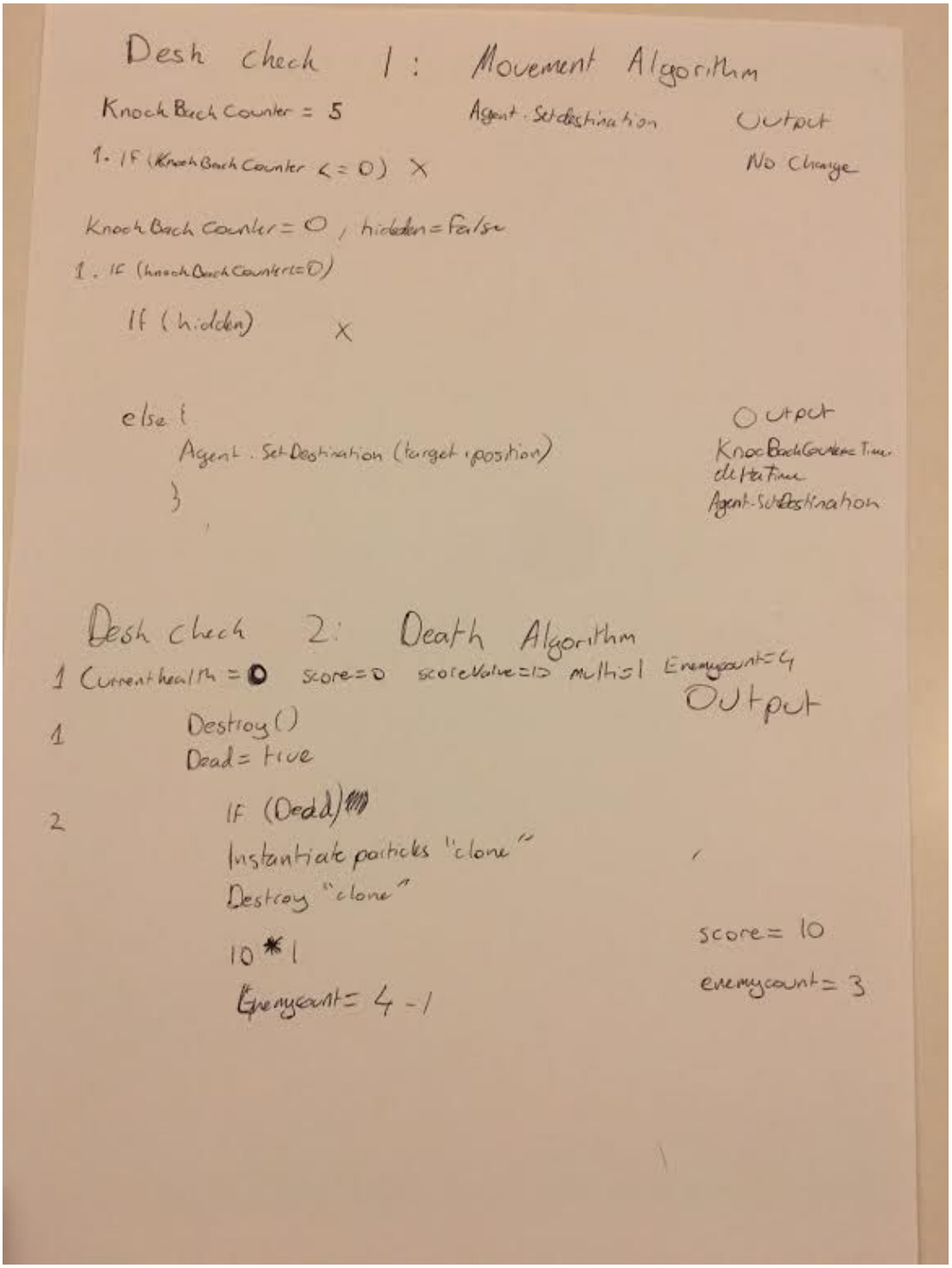
Lots of testing was needed to find the most balanced amount of health for an enemy. Eventually I settled on 10hp for each enemy, with shots doing 5 damage each it takes two shots in order to kill an enemy.

Peer Report

Billy's project, similar to my own makes use of the Unity framework and MonoDevelop. This means that many of the functions are very familiar to me. His project however makes use of the third dimension and therefore uses a third axis for movement and rotation. Despite the 3D nature of his project I found the code very familiar and easy to read and understand. His use of internal documentation also helped.

Through play testing there were some instances in which the player would become stuck against a wall and unable to shoot or move as they are stuck in an endless loop with the enemies. In these situations I would recommend spending slightly more development time on fixing bugs rather than focusing on things like sound effects, which can become slightly irritating to listen to after a while.

Billy makes use of the Object Oriented Programming paradigm by instantiating objects through the enemy classes and this really helps with code efficiency and simple design. The weapon system is also Object Oriented and seems to be very intuitive in its application. Overall Billy's code is modular and follows the outline of the task well.



Justification of Coding

The code written for Bruvforce was modular, making use of various classes in order to complete the game. The code was well documented both internally through comments and intrinsically through a proper naming scheme.

Player Controller

The player controller class handled all data about the player. The state of the player's movement, its shoot state and the lives and ammo it has.

Movement

Movement is extremely basic in Bruvforce, simply using 'A' and 'D' for movement on the x axis and the 'Space' key for movement on the y axis.

```
// Basic movement using the A and D keys to go left and right
if (Input.GetKey (KeyCode.D) && canmove == true) {
  transform.position += Vector3.right * speed * Time.deltaTime;
} else if (Input.GetKey (KeyCode.A) && canmove == true) {
  transform.position += Vector3.left * speed * Time.deltaTime;
}
// Basic jump function using the space bar
if (Input.GetKey (KeyCode.Space) && canjump == true && canmove == true) {
  rigid.velocity = new Vector2 (0, jumpheight);
  canjump = false;
}
```

Shooting

Shooting is done in two parts, part one is done through the player controller in which the bullet is instantiated and part two in a script tied to the instantiated bullet itself which determines its movement. The bullet automatically destroys itself after either hitting something, reaching its target or after 10 seconds, this is to prevent unnecessary performance issues if large amounts of bullets are spawned.

```
// Shoot a bullet on click
if (Input.GetMouseButtonDown(0) && ammo >= 1 && canmove == true){
  if ( ! EventSystem.current.IsPointerOverGameObject())
  {
    tempbullet = Instantiate(bullet, bulletspawn.position, Quaternion.identity);
    ammo--;
    setAmmoText();
    Destroy(tempbullet, 10f);
  }
}
```

```
void Update () {
  //make the bullet move towards the 'target' aka mouse click position at an angle
  transform.position = Vector2.MoveTowards(transform.position, target, speed * Time.deltaTime);
  //when the bullet reaches the position of the click, delete it
  if(transform.position.x == target.x && transform.position.y == target.y){
    Destroy(this.gameObject);
  }
```

Death

The use of an IEnumerator for the death mechanic was an interesting undertaking as it was the most simple way that I found to create a death mechanic similar to that of the mario games in which the player launches up into the air before falling through the ground until finally the game declares him to be dead. The IEnumerator allowed me to start a function and then pause it halfway through whilst it waited for the character to fall through the floor, once this happened then the game would finally show the death screen.

```
IEnumerator death(){
  // Adds a small force to the character to make him jump similar to how Mario dies in Mario games and removes all collisions
      // so that the character falls through the floor, again like Mario.
  rigid.velocity = new Vector2(0, jumpheight);
  circleCollider2D.enabled = false;
      capsuleCollider2D.enabled = false;

      // Waits a second before pausing the game so that the character begins to fall through the floor in a Mario style death
  yield return new WaitForSeconds(1);
  Time.timeScale = 0;
      // Activates the Death Menu that allows the player to retry or go to menu.
  DeathUI.SetActive(true);
}
// remove health when colliding with enemy
void OnCollisionEnter2D(Collision2D collision){
  if (collision.gameObject.tag == "Enemy")
      {
          health -= 1;
          Debug.Log("Lost some health");
          checkHealth();
      }
}
```

Enemy Controller

The enemy controller handles the basic AI of the enemies encountered in bruvforce and includes movement, shooting and death.

Movement

The movement for the enemies took a lot of planning and research, many things I tried simply failed to work as I couldn't get the enemies to turn around once they reached the end of a platform. Eventually the solution was found to use a raycast in front of the enemy to detect whether or not there was ground in front of the enemy. If the raycast did not detect anything then the enemy would rotate itself 180 degrees and start walking the opposite direction.

```
// makes the enemy continually move to the right
transform.Translate(Vector2.right * speed * Time.deltaTime);

//detects infront of the enenmy for a ground, if no ground is found the enemy turns around.
RaycastHit2D groundinfo = Physics2D.Raycast(groundDetection.position, Vector2.down, 2f);
if(groundinfo.collider == false){
  if(movingRight == true){
      transform.eulerAngles = new Vector3(0, -180, 0);
      movingRight = false;
  } else {
      transform.eulerAngles = new Vector3(0 , 0, 0);
      movingRight = true;
  }
}
```

```
// if an enemy collides with another enemy, turn around
  void OnCollisionEnter2D(Collision2D collision){
    if (collision.gameObject.tag == "Enemy")
         {
           if(movingRight == true){
              transform.eulerAngles = new Vector3(0, -180, 0);
              movingRight = false;
          } else {
              transform.eulerAngles = new Vector3(0 , 0, 0);
              movingRight = true;
          }
         }
  }
```

Shooting

Enemy shooting was equally challenging, finding a solution that would allow them to accurately target the player proved to be difficult. At first they could only shoot in front of them straight along the x axis but this proved to be much too easy to avoid and many of the platforms were not level with each other so the bullets would end up just flying over the player anyway. The new system takes a snapshot of the players location before firing upon it at an angle directly towards them, it will not follow the player if they move and will delete itself upon reaching the spot that the player was standing when the projectile was fired. The current system is still possible to dodge but is more challenging than it was before. I also had to implement a range system so that enemies that were way off screen could not fire upon the player and become impossible to avoid.

```
    //must be within 20 'x' units to start shooting at player
    if (this.transform.position.x - Player.transform.position.x < 20){
        if(shootDelay <= 0){
        Instantiate(Bullet, transform.position, Quaternion.identity);
        shootDelay = startDelay;
      } else{
        shootDelay -= Time.deltaTime;
      }
    }
  }
```

```
 void Start () {
   player = GameObject.FindGameObjectWithTag("Player").transform;
//target is the position of the player
   target = new Vector2(player.position.x, player.position.y);
 }

 void Update () {
     //make the bullet move towards the 'target' aka mouse click position at an angle
   transform.position = Vector2.MoveTowards(transform.position, target, speed * Time.deltaTime);
//when the bullet reaches the position of the click, delete it
   if(transform.position.x == target.x && transform.position.y == target.y){
     Destroy(this.gameObject);
   }
 }
```

Personal Reflection

Overall I am pretty disappointed with how my solution turned out. I managed to get done what one might call a complete prototype but overall I believe there were alot of things outlined in my Part A which I was unable to complete. In the early stages of this project I think that I was highly motivated and managed to get a good amount of work done each lesson. In the lead up to the trials period and throughout my motivation dropped severely as I struggled to find both the time and energy to continue working on the project. Once trials finished I really felt quite burnt out and my efforts to continue work on this project were usually only felt in short bursts. Once again I have found myself cramming the night before the task is due, as I did in Part A and I feel as though my lackluster hand-in may show that. Overall I am proud of the work that I did manage to complete, such as the player movement and enemy AI but I feel as though there were more things I should have implemented, specifically more levels, a checkpoint system and a scoring system.