Practical Gaming 2024

# Name of Student….. Dylan Barry

# T Number …. T00252785

# Name of Project – Age of Battle

# Gameplay

The game is played by building units through a button interface in the top right-hand corner and deploying enough units from your base on the left to destroy the enemy base on the right (setting the counter from 50 to 0.) Camera is controlled with WD and moves from left to right, you can also pan the camera by placing the mouse at the left or right edge of the screen.

The button menu features from left to right – a unit spawner and a larger special ability button which can be used for free are the only two functional buttons [First button on the left and the big rectangular button below] Rest are unimplemented and don’t do anything. To exit the unit spawner menu click the last unit icon on the right (It looks like a black chess pawn)

# Coding

Under each of the following headings, please describe the concept, why is it or isn’t it useful/needed, where do you implement in your project, you may provide screenshots or cut and past code segments etc..

* Frame Rate Independence
  + Frame rate independence means to keep movement independent of frames and to not have its speed influenced by the frames per second.
  + It is needed to ensure that low-performance machines do not ‘slow’ the game down and high-performance machines do not ‘speed’ the game up which would lead to a potentially jarring experience.
  + Frame rate independence is achieved in this project through using Time.DeltaTime
  + This code is located in the Move() function for the AbstractUnit script

Vector3 movement = new Vector3(speed \* direction \* Time.deltaTime, 0, 0);

transform.position += movement;

* Interfaces
  + Interfaces are classes with no implementation.
  + They are useful for situations where a diverse range of implementations for a method might be used by different classes.
  + The IHealth interface is used by the AbstractUnit to implement the TakeDamage function.

interface IHealth

{

void TakeDamage(int dmg);

}

* Inheritance
  + Inheritance is when a class inherits code from another class and essentially becomes a clone of that class, along with containing any specific code written for that class
  + It is useful for reducing the amount of code written in a project and recycling often repeated functions
  + E.G. my AbstractUnit class is implemented by every unit in the game and saves having to write and manage Move() for every single class.
  + It also makes updating code much easier and straight forward as only one class needs to be updated to update all inherited classes.

public abstract class AbstractUnit : MonoBehaviour, IHealth

public class GiantUnit : AbstractUnit

* Case pattern
  + A case pattern is a set of behaviours that AI might use in a game, patrol, idle, roam etc.
  + This is necessary for determining appropriate behaviour for AI.
  + Every unit in the game features a walk, idle, attack and death cycle

protected bool isAttacking = false;

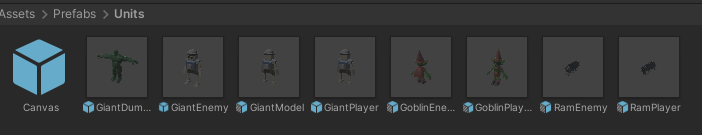
protected bool isMoving = true;

protected bool isDead = false;

* Observer Pattern
  + An observer pattern is essentially when a manager script makes a list and keeps track of that list to update and change it
  + This is useful for managing AI characters or UI elements
  + My ButtonManager script makes a list of buttons, instantiates a button for each index and keeps track of them, causing a unique outcome for when each button is clicked.
* Polymorphism
  + Polymorphism is a class taking on many different subsequent forms E.G. car class becomes 3 different classes, “sport”, “off-road”, “SUV”
  + Polymorphism is not used in my code so far
* Communication between scripts/game objects
  + This is very important for gameplay – A player needs to know the HP of the enemy base or their base for example so a script needs to find out this information from the script managing base HP.
  + The base health counter checks an instance of the BaseHealthManager class assigned in the inspector to display its HP as text.
  + My ButtonManager script is also attached to a canvas that exists in the game world and has other manager scripts attached to it that will get called as their respective buttons get clicked

public BaseHealthManager baseHealthManager; // Reference to the base health manager

* Instantiation and Prefabs
  + Instantiation is necessary to actually put a new instance of a GameObject or prefab into play during run-time.
  + Prefabs are an indispensable tool for keeping a copy of, for example, a unit and its required add-ons like a canvas and a health bar script for displaying its HP, aswell as its scripts and animation controller.
  + There is a plethora of instantiation and prefabs used in my game, the player instantiates player controlled prefabs using the button menu and the enemy AI instantiates enemy controlled prefabs, setting isPlayerControlled = false; after spawning them.
  + There are separate enemy and player prefabs, some with separate animations. This is to resolve issues with animations facing the wrong way. (No separate player and enemy scripts exist, they are all GoblinUnit, GiantUnit, etc.)



* Magic Numbers
  + Magic numbers are when variables are set to a value, but it is unclear why that value in particular thus the term Magic Numbers
  + I have introduced enough variables to provide clarity in my code as to why something is being set to a certain value
  + E.G. my attack coroutine calls WaitForSeconds(time goes here) and I put a variable called attackTime to make it clear that this is the time the unit waits between attacks

yield return new WaitForSeconds(attackTime);

* Model Animation
  + Model animation is necessary to bring life to a character and convey what they are doing – it’s practically essential for almost every single game.
  + All the units in my game have their own unique animations which convey whether they are idle, attacking, moving or dying.
  + These are controlled by animation controllers attached to the game rig which are fed Boolean values by the script – E.g. isRunning = false, isAttacking = true
  + This determines what animation the model chooses to do.

private IEnumerator AttackRoutine(AbstractUnit target)

{

isAttacking = true;

isMoving = false;

while (target != null && target.health > 0)

{

target.TakeDamage(damage);

PlayAttackAnimationAndSound();

yield return new WaitForSeconds(attackTime);

}

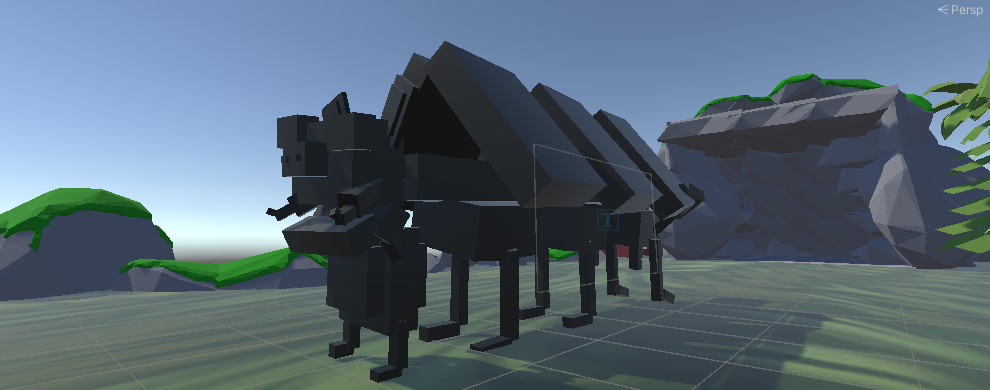
isAttacking = false;

isMoving = true;

}

* Self made models and or animations
  + Self-Made models and animations are often necessary to bring the unique creative vision of the game designers to fruition in a way that can’t be achieved by online assets alone
  + The enemy giant and battering ram animations and models were made by me in Maya Autodesk and exported as FBX
  + Each animation was baked into the model and exported as its own FBX

**Battering Ram model:**

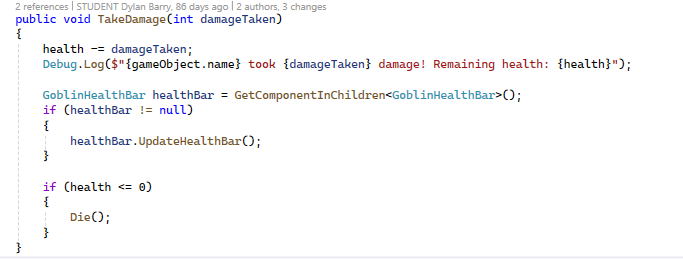


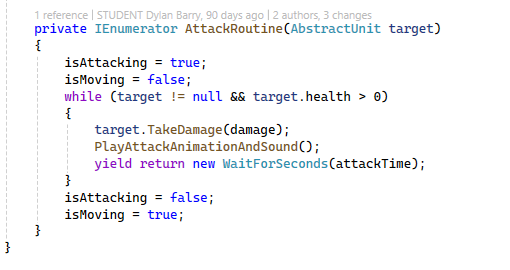
**Enemy (Skeleton) Giant model:**

****

* Interactions between objects/scripts
  + This is necessary for any kind of game at all to create basic gameplay, such as enemies getting hit by player bullets or detecting collisions between vehicles in a racing game
  + In my game every unit will react to units in front of them by using trigger box detection, if the unit is friendly and in front they will call functions to idle in the code or an attack function if the unit is hostile.
  + The enemy base will also react to units if they enter its trigger box, depleting HP from the base if they are hostile.

checkForFriendlyUnitCollisionAhead();

* Proper code placement
  + Classes should follow the first principle of SOLID – Self.
  + They should not interfere with another code's variables directly for example, this is why variables are typically set to private and getters and setters are used.
  + In my code for example when units deal damage to another unit, they don’t just ‘reach in’ and directly change their code. They pass in a damage value to the attacked unit’s TakeDamage() function. This gives any AbstractUnit derived class full control of their own variables.
  + 



* Code repetition
  + Code repetition needs to be minimised as much as possible to reduce bloat and errors in the code as well as making it easier to update.
  + Code should be recycled through abstract classes and inheritance where possible to minimise waste.
  + I have done this through an AbstractButton and an AbstractUnit script
* Feature 1 – Visual Design
  + The game was visually designed to sync together using a low-poly cartoonish style
  + The background environment the enemies fight in was made using low-poly asset trees and rocks, adding a nice remote feel to the game
  + The unit designs convey a fantasy medieval setting in stage 1 – Goblins, ogres, skeleton giants and sentient battering rams fight each other.
  + All units have health bars if you hover over them, this increases player excitement as they can see how battle is progressing
* Feature 2 – Sound Design
  + An ambient track has birds chittering and the wind blowing throughout, consistent with the rural landscape
  + Units play various sounds, increasing immersion and interest
  + An overpowering and epic music track adds tension to the game as the player tries to defeat the enemy, this turns the game into a glorious struggle.
* Feature 3 – Cooldowns
  + Unit buttons have a cooldown
  + This allows for stimulating gameplay as the player can intrinsically feel as if the units with longer cooldowns are stronger and the shorter ones are weaker.

