



Deadliest Weapon: a Design Document

23.07.2022

CONTENTS

CONTENTS	1
Introduction	4
Overview	4
Goals	4
Overall Objectives	4
GAMEPLAY	4
Objectives	4
GUI	4
Game Progression	5
Controls	5
Story	6
First Minutes	6
Game Flow	6
Victory Conditions	6
Algorithms	6-7
SORTing	7
Data Structures	7
Linear	7-8
Non-Linear	8
Budget	8-9
Fees	9
UX	9
Design Principles	10
UI	10
Design Principles	11
Version Control: GIT	11
GIT LFS	11

Art	11
Splash Screens	11-12
Concept Art	12
Game Mechanics	12
Movement	12
Combat	12-13
Damage Types	13
Environment/Maps	13-14
Enemy AI	14
Weapon Degradation	14
Level Design	15
Finishers	15
Executions	15
Grapping	15
Melee System	16
Characters	16
Game Modes	16
Combos	16
Damage Mechanics	17
Hero Mechanics	17
Hero Mechanics 2	18
Guard System	18
Guard Break System	18
Monsters	18
Arcade	18
Mechanics Breakdown	19
HUD	19
Vertical Slice	19
Vertical Slice: additional features	20
Vertical Slice: extra features	20
Parrys	20
Monetization Model	20

Marketing	20-21
Target Audience	21
Platform	21
Assets	21
Timeline	21
Developmental Milestones	21
Target Release Date	21
Phases of Work	21
Testing	22
Black Box White Box	22
Alpha	22
Beta	22
Database	22
Purpose	23
SQL and HTTP	23
Restful APIs and Node.JS Servers	23-24
Website Design	24
Frameworks	24
Flask	25
CSS	25-26
Javascript	26-27
Network Replication	27
RPCs	27-28
C++	28
Dedicated Server	29-30

Introduction

This document aims to inform the reader about the game being produced by Dragon Games; it is subject to rework and will be continuously updated throughout development to accurately reflect our needs.

Overview

The planned game is a 3D fighting game, featuring 1v1 armed combat, as well as grouped combat with various game modes; the game allows players to control various different characters from different cultures and time periods from around the world, utilizing their unique fighting styles, armor and weapons.

It's intended to first be a digital download being published on the Epic Games store and the Steam marketplace.

Overall Objectives

The objectives of our group will be to have a finished video game by the intended deadline, with the set deliverables being the commits pushed to Github by the developers. To further break this down, we need to discuss the various systems that will make up the game: the menus, the game mechanics, the story and the completion of these systems will ultimately be our set objectives.

We aim to bring on further team members to make up our sound and art development teams; studios are being looked into for purposes of motion capture to aid with animations.

Gameplay

The gameplay will allow players to freely move around a 3D arena, a small level design

Objectives

This will vary across the game modes; the sole core objective will be to deplete the opponent's health and finish them.

GUI

TBA. Will vary on game mode.

Game Progression

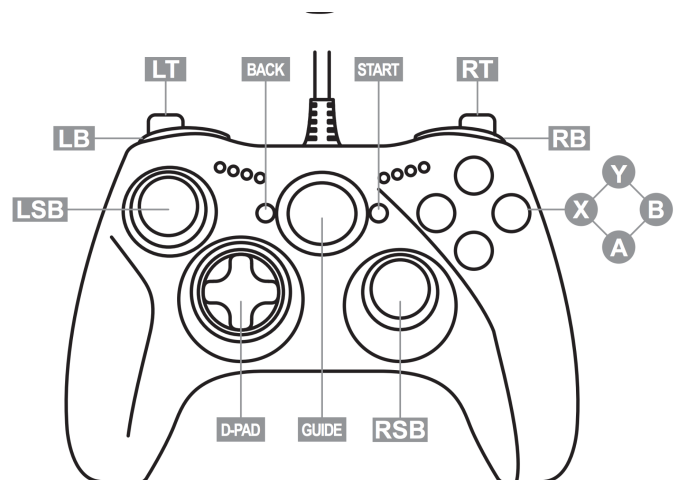
TBA. Will vary on game mode.

Controls

The proposed control system for our game is shown below, where == denotes suggested keybinds for a keyboard and a microsoft standard gamepad (playstation controls are in brackets). The keyboard suggested commands are placeholders.

The * means the input must be inputted at the same time the attack incomes; where top, bottom, right and left correspond to the physical buttons on the right hand side of a playstation game controller.

- Sprint == 'Shift'+'W' == **LT** (L2)
- Dodge == 'F' == **Right Stick Button** (R3)
- Un/Equip == 'Y' == **RB** (R1)
- Grab == 'I' == **X + A** (Left + Down)
- Push == 'O' == **Y + B** (Top + Right)
- Neutral Guard == 'P' == **RT** (R2)
- Special == 'G' == **B Button** (Right)
- Heavy Modifier == 'L' == **LB** (L1)
- High Normal Attack == 'H' == **Y Button** (Top)
- High Heavy Attack == 'P' == **LB + Y** (L1 + Top)
- Mid Normal Attack == 'J' == **X Button** (Left)
- Mid Heavy Attack == 'N' == **LB + X** (L1 + Left)
- Low Normal Attack == 'K' == **A Button** (Bottom)
- Low Heavy Attack == 'B' == **LB + A** (L1 + X)
- High Parry == 'T' == **RT + Y*** (L2 + Top)
- Mid Parry == 'Y' == **RT + X*** (L2 + Left)
- Low Parry == 'U' == **RT + A*** (L2 + Bottom)
- Taunt == 'X' == **Left Stick Button** (L3)



Story

The only details we have at the moment is a medieval theme revolving around themes of fighters inspired from legends from around the world. Individual characters all linked by some general event; can be explored in the 'Arcade' game mode. See characters.

First Minutes

TBA. Will vary on game mode.

Game Flow

TBA. Will vary on game mode.

Victory Conditions

Ultimately it will be the depletion of player health till death, or completion of the individual game modes..

Algorithms

The majority of our programming will be done using blueprints, however we will need to do some 'hard' programming for approximately 25% of the game; we anticipate using the C++ programming language. When creating functions, henceforth known as algorithms, we have to ensure that they're both written efficiently and run efficiently; this can be analyzed using empirical and theoretical analysis.

Empirical Analysis is when the efficiency of an algorithm is analyzed via computational experiments; you write a program that implements the algorithm, run the program, and use a method (another function that you have written) to obtain the actual running time.

Theoretical Analysis involves estimating the time complexity (the running time) of an algorithm; the time complexity expresses the total number of elementary operations (addition, subtraction, comparison) for each possible problem instance.

An Iterative algorithm consists of statements repeated a number of times in a loop; a recursive algorithm calls itself repeatedly, until a base condition (stopping condition) is satisfied; during certain developmental points, we will need to weigh up the advantages and disadvantages of both recursive and iterative algorithms, and which type is more efficient for the specific developmental need at the time.

An iterative algorithm is run n times, the running time of this algorithm would be to the order N ; the time complexity is $O(n)$. Recursive algorithms are typically slower due to the additional required resources and the nature of how recursive algorithms function.

SORTing Algorithms

Practical uses for these algorithms include in the scoring system for the game, and listing players in order of who is winning.

Sorting refers to the task of placing objects in an organized manner; let's say we have a list of numbers. We can run different sorting algorithms to sort the list in different ways depending on which SORTing algorithm is used; each algorithm type has its own advantages and disadvantages and to minimize boredom I will simply list them instead of explaining them.

An algorithm's time depends on

Algorithm Type	Time Complexity		
Selection sort			
Bubble sort			
Mergesort			
Quicksort			

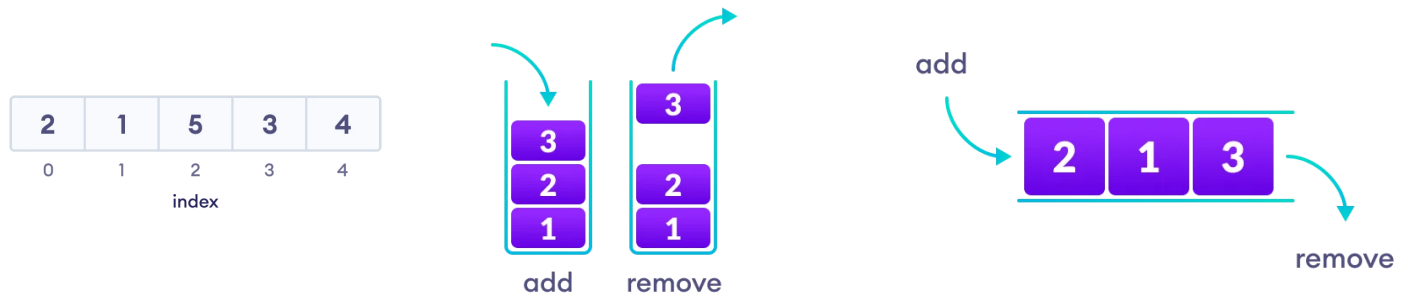
DataStructures

Data structure is a storage that is used to organize and store data; depending on our specific need, we will need to use the appropriate data type. It's also important to ensure that the data structures we're using are the correct type, either: linear or nonlinear.

Linear

Linear data structures are arranged in sequential order; all objects are present on a single layer and it can be traversed in a single run. However, the memory utilization of linear data structures is not efficient and the time complexity increases with the data size; **linear data structures should only be used for relatively small data.**

Examples of Linear data structures are: array (left), stack (middle), queue (right).



NonLinear

Nonlinear data structures are arranged in a hierarchical manner, there are objects present at different layers, therefore to fully traverse the nonlinear data structure requires multiple runs. Different structures utilize memory in different efficient ways. One advantage of nonlinear data structures is that the time complexity remains the same regardless of the size of the data.

Examples of nonlinear data structures are: graph (left), tree (right).



Budget

The initial declared budget for planning and development is £10,000. See basic table for review below.

Item/Service	Cost	Date
Assets	£350	July 2022

Employee Roles:

- Project Supervisor (£100/200 Fixed Rate depending on hours)
- Lead Programmer (£10 p/h 20 hours)
- Programmer (£10 p/h 20 hours)
- Level Designer (£100 per level)

Weekly costs vary from: £500, to £700.

The first development term is between July and September (12 weeks) and will cost **AT MOST** £7500: (12 weeks * £600 = £7200 + £300, assuming 3 levels in 3 months).

This will still put us within budget for the first developmental cycle. As mechanics are being developed, the budget will be increased as 3D modelers and animators are brought onboard.

Fees

GIT Enterprise is a yearly membership subscription for which the sum total is billed in USD \$ and as such a conversion charge in the amount of approximately 3.75% is taken by PayPal (or any other payment provider). Our planned payment method is PayPal and we therefore need to account for the current exchange rate at the time of purchase and the fees taken by the payment handler.

UX

The user always comes first and one of the most common fundamentals of UX design is to put yourself in the user's perspective and see things from their POV; usability testing is essential and optimal design varies from websites to products.

Accessibility testing is essential in ensuring that the users are able to actually access and read the information; elements which affect this includes; the choice of colors, the amount of contrast and the text's readability (the size, font and colors of the text).

Less is More.

Typographical Hierarchy makes designs more user friendly. Information is easily accessible when the user is able to scroll or traverse through the information at their own pace.

Design Principles

- Useful
- Usable
- Findable /Intuitive
- Credible /Reliable
- Desirable
- Accessible
- Valuable

UI

The User Interface should be minimal in nature whilst informing the user with the most need to know information as possible. There must be a match between the language used in the real world, and language used in the system, whether that be in dialogue or internally.

A consistent design should be clearly shown throughout the system and failure to maintain a consistency will result in an increase in the cognitive load of the user, by forcing them to learn something new that may feel strange to them as it's different to what they're used to.

Error prevention should be used throughout the system; an example of such prevention could be the limiting character available to users during times where the system requires user input to continue; this would prevent injection exploitations and other such invasive malicious techniques.

It's beneficial for users to recognise rather than recall aspects of the game as this would minimize the users memory load; an example of this is how it's easier for most people to recall whether something was true or not, rather than what or where something is; for example, it's easier to answer the question "Is Lisbon the capital city of Portugal " rather than, "what is the capital city of Portugal?".

An aesthetic and minimalist design is preferred as every relevant piece of information competes for screen time and focus with every irrelevant piece of information on the screen. Less is best.

The best system is one that doesn't need additional explanation, however to ensure everyone's needs are met, it may be necessary for documentation to be added to aid the users in completing their tasks may make

Design Principles

- Home Style (Familiarity) Consistency
- Shortcuts to save time/ Bookmarks
- Information Feedback (an example is a password strength bar)
- Design Dialogues for Closure
- Prevent error by limiting User Input/ Usability Testing
- Minimize Cognitive Load

Version Control: GIT

A new GitHub account will be created under the account name 'Dragon Games Github' for simplicity sake and it is planned to purchase the Github Enterprise membership to ensure there is sufficient storage for the project's needs.

If the file size turns out to be too large, we can simply break the game down into 3 parts and create 3 separate repositories to manage the games files, these parts could be: the **combat system**, the **game mechanics** and then the **menus** system.

GIT LFS

Due to the large nature of the file sizes of the project, it will be required that we use Git LFS to store multimedia files; the set-up and installation of which can be easily followed through any simple guide.

The HTTPS post buffer in Git will also need to be increased from the default size of 1MB to approx 50MB, to allow for pushing large bits of data to the repository.

We are storing select asset folders in GDRIVE and these can be found in the git ignore file in the root development folder; developers need to remember to pull/push and to consider altered files in the git ignore.


Art

See concept art folders for images- Please check design ideas with the level designer and modelers.

POLYCAM!

Splash Screens

Add from splash screen files. Consider Polycam too, as well as for assets.



Splash screen selection needs to be 'deadliest warrior'/martial arts. This needs to be altered to be changed to suit the actual maps. Pick photos again after levels have been designed/bought.

Concept Art

Add from concept art files

Rustic features; photo realism; leaves/tree examples.

Can use the photo app for assets- check baz messages

Game Mechanics

This section of the document will focus on the actual work the developers will be focusing on in the engine. Not all mechanics will be present in all game modes.

Movement

The base movement will be in the 4 directions.

Unique special movements will be attributed to each character; these unique movements so far have been defined as: a dodge roll (planned for the Ninja/Shinobi) and flip tricks & parkour which are all animations from an asset pack purchasable from the Epic games marketplace. The unique movements could possibly be initiated via an input of button pairs for controllers, or a single key for PC users.


Once a player's health hits zero, they are downed; in this state, they are able to be revived and are able to crawl. This revival mechanic will be known as 'down+out' for simplicity sake. This crawl movement will be 25% of the player's normal walking speed.

Let us define stamina as 100; the planned cost of sprinting is 3 per second and the planned cost of the unique special movement is 10.

A mantling system is purchasable from the Epic games market; this mantling system allows for the player model to climb objects and allows for another set of movement; this could be the 'free roam' set of movements. Destructible barrels are obtainable from an asset pack also; they are interactable and allow for additional actions for the player.

Combat

The targeting system is limb based. High attacks focus on the head; mid attacks focus on the arms and torso; low attacks focus on the legs and feet.



2 types of attacks: lights (no planned property), heavy (knockdown effect). This knockdown effect present on the heavy attacks could knock the player back several feet depending on the size of the maps; a delay to input could be added for a player who has been 'knocked down' to prevent them from being able to dodge/roll recover.

Heavy attacks are planned to not be a part of combos for the time being; they could be combo enders.

Combos for the inputs are entirely up to the developer working on said combos; this is due to the nature of the animations used for the attacks being flimsy and we will be relying on the experience of the developer to determine what combos will be used (animation montages). They are planned to be 3 strings long maximum for the time being; this is so there is some back and forth and one player isn't able to just overly pressure another.

An effect of blocking any attack with a weapon is a lowering of the weapon's 'health'/'sharpness' as well as chip damage from taking the hit, as well as a slight cost in stamina.

Damage Types

Properties for these types are to be determined for some.

Sword animations that end with a 'slash' shall be of the 'slash' type.

Sword animations that end with a heavy 'thrust'-like movement shall be of the 'slash' type.

Weapon animations that utilize non-sharp weapons shall be of the 'blunt' type.

Fire will be present as a hazard on some levels, this could have a damage over time 'burn' effect for a few seconds (3 perhaps); poison will be used on poisoned arrows fired from bows; electric potentially from a lightning strike will be used as a special attack, and could have a stun like effect; bleed is another damage type we can consider working out and adding.

Fire damage will also be present from a fireball projectile attack, this can cause a 'burn' effect for damage over time. The electric damage type can stun the player- their movement will be halted and they will be unable to input anything for a period of time. The Ice damage type can slow the player's movements down by a fixed percentage.

We could also utilize wind as a defensive element to redirect certain attacks/projectiles.

Environment/Map

The mountable assets for some maps are: a horse, potentially a centaur. These are from asset packs.

Interactables in the levels are planned to be destroyable barrels from an asset pack, potentially letting them be throwable; a button/chain like lever could be used to cause an interaction with the map. The chain/button-like lever object can be used in several instances: to open up areas of the map and or to cause an effect to occur in the game (ie dragonfire).

Pickup-able weapons are to mainly be ranged weapons; these are including but not limited to a crossbow, a bow + arrow and potentially a pirate pistol.

Small wolf/4 legged beast objects are purchasable from the marketplace and are planned to be used as small enemies that attack the players on certain map(s). Simple AI behavior to attack until death should suffice.

Map Hazards shall include but not be limited to:

- Hole/Pit
- Campfire/Burning effigy
- Edges/ledges
- Spikes on walls
- River/Moat perhaps with drawbridge (moat for castle level)

The Persian bazaar map should contain an AI crowd. They should exhibit running behavior and fear upon seeing violence; they should panic. They will be killable.

Some levels, perhaps the castle level, will have more advanced enemies (ie knights) that are neutral but upon aggro/when provoked they attack the responsible player.

Enemy AI and the Planned Types

The wolves: Simple behavior of attacking the closest player

Civilian: Often in crowds, simple behavior of panic. No aggressive actions.

Trained Civilian: More advanced behavior. Behavior will be planned out in flowcharts as development progresses. TBA.

Weapon Degradation

Weapon degradation is planned to occur when a player uses their weapon to defend an incoming attack. A weapon's damage output will be linked to a value called 'sharpness' and as players use their weapon to defend, it will dullen. This will prevent the 'broken weapon' problem in gaming where the player is effectively left to die once they lose their main fighting option.

Consider instances of guarding and amount of dullness occuring and various values.

Level Design

Levels will be made from purchasable asset packs from the store; these levels will then have instances of them created and they will be the different types of maps available for the player to play on. Worst case if the level design causes too many issues, we will use premade purchasable levels from the marketplace.

The four legged beast enemies can be locked behind some cage and can be maybe freed via an interaction with a lever of sorts; they could then only attack the player who didn't free them even if provoked by the player who freed them because canines are known to be loyal. Perhaps this could be a feature with only some of the four legged beasts.

Dynamic map features could be:

- A forest could transition into a burning forest
- Apex zone like ring closing in on whatever map
- Night/day unreal engine internal system
- Castle level could have infinite time; AI could be used to force battles
- One level with halo/buff power ups pick ups.
- Dragon skin/dead dragon model could be on the Castle level

Potential maps could be: castle, dojo/shaolin arena, viking themed map, persian bazaar.

Finishers

The finisher system will be implemented from a finisher meter; this can also potentially be used for 'clashes' in a clash system. It's defined as a bar filling, specifically the requirements for filling are yet to be defined.

If the opponent is on 20% or less health, and if you approach them, a prompt will appear to input a button which begins a finisher animation.

Potential for ragdolling limbs that are severed from the body. Finishers are a type of execution.

Executions

This is defined as approaching a downed enemy player, a prompt appears for the player upon doing so, and then the execution animation plays. This will give a score/point bonus as well as potentially increase the respawn time for the player from the default.

Grappling

Implemented as the grab system; only some characters can perform takedowns.

Melee System

This could potentially be added as an alternate combat type; will involve a complete rework in regards to balancing and so should be planned as a separate instance potentially for a unique game mode.

Characters

The character selection screen should include a random select.

Instances where there are multiple genders of a character, they will have slightly different movesets or abilities.

- Knight character: Initially 4 planned, 2 men 2 women.
- Viking
- Ninja
- Vampire Girl (bonus character)
- Lizard Character: Hammer/Axe moveset. 'Wild' attacks.
- Turtle: Master Oogway type character; auto deflect ability for projectiles maybe
- Orc: warhammer, hyper armor, design from hobbit movie; hammer animation pack

Game Modes

Main Game Modes

- Elimination: 1 life per round, first to 5 rounds. 4 to 8 players.
- Duel: 1 vs 1, first to 3 rounds or 5; players can vote/choose after character selection.
- Death Match: Team deathmatch, first to x score, 6 to 8 players.

Additional Game Modes that could be added include but aren't limited to

- Offline VS mode: duels
- Halo style Zombies
- Arcade: MK Tower like/Tekken arcade.

Combos

All of the combos are initially planned to be simple 3 or 2 hits due to the limited amount of animations that seamlessly go together. The inputs will be interchangeable to perform variations of the combos.

The only initial defined rules are for combos that end in heavies to have a knockdown effect; this will be avoided initially until the combat blueprints are working effectively. The animation frames will also need to be balanced to avoid attacks looking too fast or slow.

Damage Mechanics

Knockdown State: a tag, lasts for the duration of the 'downed' state. All incoming damage is reduced by x%. Use for this mechanic is when someone receives a knockdown effect from being struck with a heavy attack.

Hit Recovery: allow knocked down opponents to dodge/roll in a direction dependent on the input (two directions like tekken); do not allow a player character to lay down on the floor in a knocked down state for an extended period of time, like they can in Tekken.

Critical Chance: Percentage chance of bonus damage. Arbitrary value that varies depending on which character the player chooses. An example: 1 in 10 light or heavy attacks does more damage; the critical chance hit will change each time to avoid players pre-hitting and whiffing attacks to 'load' the critical chance hit as their next hit.

Clash System: trigger is yet to be defined but victory can be determined via whomever has the most finisher meter. The loser loses x% of finisher meter (to be determined) and x% of their health. Victor does not gain anything, but simply does not bear any cost for participating in the clash.

NPC Behavior needs to be modeled in a flow chart.

Hero Mechanics

Hero	Attack	Proposed Speed	Block Speed
Ninja	Heavy	FN	N
	Light	F	N'
Knight	Heavy	N	N
	Light	FN	N'
Viking	Heavy	FN	N
	Light	F	N'

Proposed block speed may need modifying: How can a knight in heavy armor block at the same speed as a ninja wearing cloth?

N' is defined as a faster speed.

Hero Mechanics Two

Potential ability mechanics for the viking: 'berserker' - increased speed, increased damage. Reduced stamina cost, reduced incoming damage.

Each character will be planned to have their own unique movesets and abilities.

Special attacks will be triggered by a specific combo; the special attacks will be unique to each character.

Guard System

On blocking, there will be chip damage.

Parry punishments and timings need to be defined.

Guard Break System

As you are blocking, on each successfully blocked hit, you lose stamina. This will be compounded with the chip damage.

Monsters

Monsters are planned to be map specific.

Wolf: small four legged enemy is planned. Simple behaviors. The wolf can be released after x minutes into a match on a specific level.

Wolf like creatures: could be implemented similarly as the Wolves with slight variation on the meshes with what assets there are.

Potential for a bear too.

Arcade

Mortal Kombat tower like experience. Each character could have an image associated with their lore and have white descriptive text narrating their back story and their purpose for participating.

Hidden/secret boss could be hidden on a level. This is the story mode.

The pictures can be individually commissioned after the character models are implemented alongside a menu system and character selection screens.

Mechanics Breakdown

Down and Out: this is an injured state post defeat, the player can be executed by an enemy or revived by an ally; they can also be attacked and harassed by enemies and promptly be killed.

Environment monsters can also finish off downed and out state players.

Ranged weapons and magic can also be used to finish off a down and out player. Executions are not forced.

HUD

- Health
- Stamina bar(s)
- Map Radius (Circle, dots)
- Objective Points (UI elements)
- Abilities/Projectile count
- Finisher bar
- 2 Teams/Fighters icons listed- potentially in a scoreboard form depending on gamemode.

There will be a variation on the GUI and HUD depending on the game mode as there are different camera angles and styles of gameplay planned for the different modes, incorporating different styles.

Vertical Slice: Features

Movements: rolls, flips and unique special dodges.

Targeting system; light attacks and heavy attacks with knockdown effect.

Damage types (all)

Ranged pickable weapons from the ground on some maps.

Wolves and map hazards, alongside a couple of NPCs.

Weapon degradation

Dynamic Maps, Executions, Grabs, 3 Hit Combos, Down+Out, Critical Chance, Hit Recovery

Knockdown state, Unique abilities/special attacks, Dragon (the model being present in a map).

Additional Features for Vertical Slice

Destroyable objects (barrels)

Ranged pirate pistol.

NPCs alongside ragdoll limbs.

Finisher (bar)

Melee System (basic or advanced)

Wolves (additional behavior and more 4 legged animals implemented)

Extra Features for Vertical Slice

Mountable Centaur/Horse. Throwable barrels. Advanced NPC behavior. Additional heroes. Clash system. Arcade.

Parrys


To be determined: a table of balanced punishments needs to be plotted. Placeholder can just be a guaranteed heavy landing after a successful parry, with the player who was parried being locked from an input for a second.

Monetization Model

A Season pass could be used alongside a freemium model with further costumes available for customization. Microtransactions have a very negative reception, especially in fighting games, so ensure they are achievable through in-game play.

Marketing

In its simplest state, marketing is the process of getting to know your customers needs and developing a rapport and establishing and building relationships with them. Our target audience makes up a large viewership on many streaming platforms such as Twitch, Facebook live, Youtube and even TikTok.



A website for promotional purposes will also be made; special editions of the game? Digital bonus' could be offered if people buy from our site; advertised on youtube/tiktok/twitch.

Target Audience

The target audience for this game is adults in the age range of 18-25. The language and bloody nature of the game makes it rated 18; the non serious nature of the game, combined with the fighting genre it is in makes it ideal for young adults.

Platform

The game is being developed for PC using the Unreal Engine Version 5; the intended platform for launch will be the Steam website and/or the Epic Games store.

Assets

Make our own. Include info from the motion capture stuff; POLYCAM can make our own assets. See Epic Vault list.

Timeline

See Sprint Project Management website.

The selected breakdown for our lifecycle is as follows:

1. July - September: Design and Initial Development
2. October - December: Development
3. January - March: Development
4. April - June: Marketing and DLC/Updates

Developmental Milestones

TBA. Will vary on implementations.

Target Release Date

Development is anticipated to take around 6-8 months and our forecasted release date for alpha and beta testing info too mention here

Phases of Work

TBA. Will vary on implementations.

Testing

TBA. Will vary on implementations.

Black Box and White Box Testing

TBA. Will vary on implementations.

Database

Databases will be used for a multitude of purposes in this project, but the main implementation of the database will be for saving states of the game to ensure playability. A Save game object doesn't suffice because the 'save game' object is only applicable for a single server instance(s) and it is much more efficient to manage and control the database using a database server.

A service layer will need to be designed and created that needs to sit on a publicly accessible server to handle requests to and from the project to the database; we can write this however we want but I'd suggest NodeJS. This way the client (players) are unable to send requests directly to the database and potentially compromise the database or information held.

We will need to host a database server to hold variables relating to the player; these cannot be stored locally due to cheating concerns and security bypasses with additional local scripts. We have full control from the server.

The MYSQL relational database will be more desirable than alternative options; we will need a database engineer to maintain the database and write APIs to communicate with the game itself and other services such as external log in-s to other websites. Anticipated traffic will mean the MYSQL will be able to more likely deal with exponential increases to the player base without requiring an entire rework of the database.

Generic queries used in mySQL include: 'select', 'where', 'create', 'insert into', 'update', 'delete from', 'group by' and 'having', 'joins', and aggregate functions 'sum', 'avg', 'count'.

We may be able to purchase some existing SQL systems from the marketplace.

Queries can also be routed via PlayerController/PlayerState blueprints; events need to be queued and the calls to the API need to be structured to avoid jams. If implemented like this, you can create a laggy network, a de-syncing game that won't or can't sync, and even a potential random disconnection from the server if the connection itself gets bad enough and requests cannot be made. A fix for this could be using an Async Function Caller to run the blueprints in ANOTHER thread,

Purpose

Authentication of a user is the simplest reason. We will also need to retrieve and store information from the database; this remote database will need to be accessed via RESTful APIs for the most efficient method of communication. HTTP methods combined with the RESTful APIs to obtain this information will be the most secure and efficient way of doing so.

SQL and HTTP

We will be using MySQL as opposed to SQLite because we will want our database server to communicate with multiple servers and database(s). License for commercial purposes can be purchased or completely avoided by using HTTP methods to access the database itself.

HTTP methods that can be used are:

- Get
- Post
- Put
- Head
- Delete
- Patch
- Options
- Connect
- Trace

Restful APIs and Node.JS Servers

Representational state transfer (REST) is a software architectural style that defines a set of constraints to be used for creating web services; RESTful web services allow the requesting systems to access and manipulate textual representations of web resources by using a uniform and predefined set of stateless operations.

APIs (application program interface) is for communication and restful APIs utilize HTTP methods to access and submit data.

The rest client is the code or the app that can access the REST services; we will be using the 'FetchAPI' for dealing with requests.

Rest servers are made very simply; there's plenty of libraries to choose from that make the creation of the server quite straight forward. ExpressJS can be used for NodeJS and Django can be used for python. Regardless of how the implementation is done, there are options for obtaining the information from them.

The Rest API is the endpoint and is where requests are submitted to (methods) for accessing or submitting data to the server. I've plotted basic code for this; REST APIs have dozens of tutorials online if anyone has trouble following.

Website Design

The purpose of the website is to have a singular point for information relating to our project; whilst social media is good for garnering awareness, the articles or information being reported can be hosted from our website and will even include a section on patch notes- to continuously push users to visit the website.

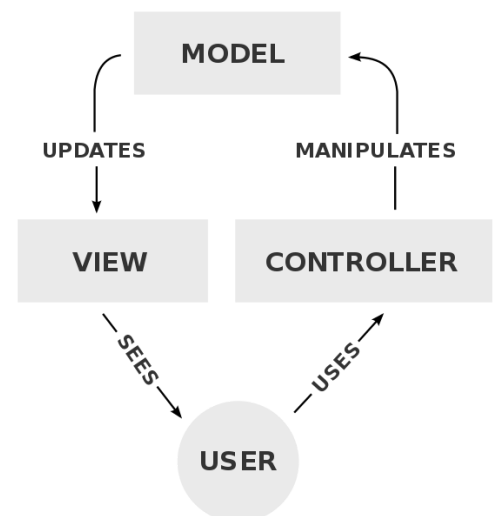
Increased traffic to the website increases likelihood of engagement in the game from the visitors; this increased traffic will also be another source of revenue for us as advertisements and sponsors of our organization or project will be compensating us as a form of passive income from 'website clicks'.

Framework

The framework for creating the website will be Flask; it's more of a microframework but its main features of integrating external libraries as its own is why we'll be choosing it. Flask behaves as a development server and debugger, whilst also giving integrated support for unit testing.

Jinja templating is used for the actual files themselves and RESTful methods are utilized throughout. There is support for secured cookies as well as being unicode based and has extensions available to even further extend functionality. Its complete documentation is also available.

Django is another framework that is available that is also based and written in Python; it is a web framework and object oriented and an MVC architecture. Model is for processing the HTTP requests with a web templating system (view) and a regular url dispatcher (controller).



Flask

The reason I picked Flask for the web application development that'll be connecting and communicating with the clients and server is mainly because of how easy it is for everyone; how scalable and flexible it is; complete control over the codebase and it facilitates experimentation and support for testing.

I'll give an example of some simple 'hello world' code using flask

The lowercase flask and the capital F Flask are different things being referenced. If this code is at all confusing to you, please refreshen on Python and some tutorials on Flask; this is very basic.

hello() can be anything you want. The text being returned can be anything too. As can be the names used for the variables.

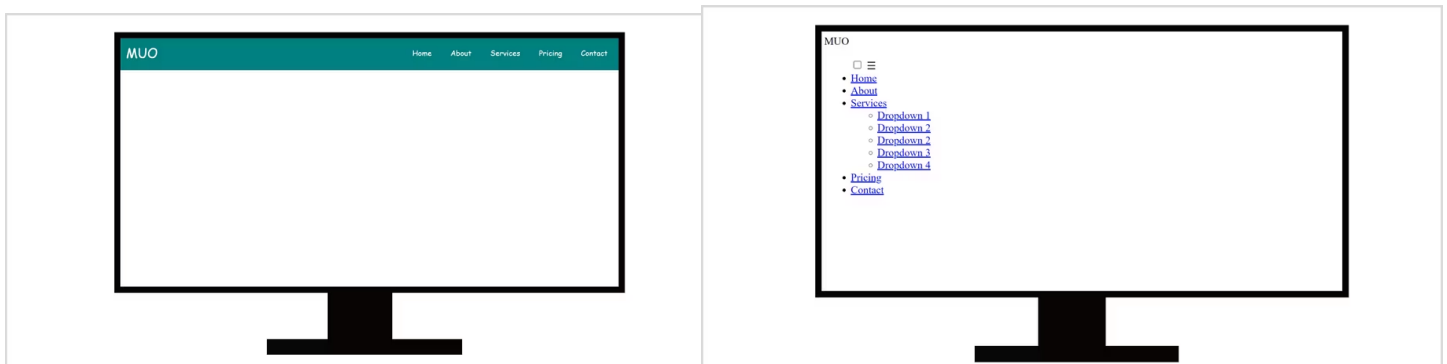
```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello() -> str:
    return "Hello World"

if __name__ == "__main__":
    app.run(debug=False)
```

CSS

CSS is a very straight forward and basic 'language' and the CSS file created for the HTML is a static file. The CSS is the style and formatting that websites require to make them readable; on the right we have a HTML without CSS and on the left we have one with; the stark difference is very noticable.



The Flask framework directory typically is in the following form as a directory tree, where file.py is any arbitrary python file and the absence of a file extension means its a folder:

Root level: static, templates, file.py

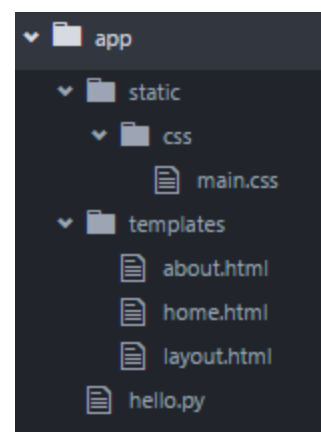
Static: css, javascript, images

Templates: home.html, whatever.html

CSS: main.css

Images: logo.jpg, welcome_message.png

Where main.css here would be the CSS file for the project.



A website has to at least have: html files, css files, javascript for interactivity and then a method for hosting. Here is some example of some CSS code:

`/* text*/` is used to comment on the code, where 'text' is any text.

The values it affects are typically listed above; for example, the final 3 paragraphs in the image are all headers being styled. The header itself, then the logo on the header and then a separate hover effect being detailed for the logo (this is mouse hovering over the logo, and a change appearing)

```
body {
  margin: 0;
  padding: 0;
  font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;
  color: #444;
}
/*
 * Formatting the header area
 */
header {
  background-color: #DFB887;
  height: 35px;
  width: 100%;
  opacity: .9;
  margin-bottom: 10px;
}
header h1.logo {
  margin: 0;
  font-size: 1.7em;
  color: #fff;
  text-transform: uppercase;
  float: left;
}
header h1.logo:hover {
  color: #fff;
  text-decoration: none;
}
```

Javascript

Javascript allows for the creation of dynamic and interactive web pages, as opposed to just static html pages with the formatting from the cascading style sheets. Javascript can change CSS styles and alter HTML content as needed.

Javascript can allow for the use of forms; pop-ups; actively use the current date/time to reflect information (eg time till release for the game/trailers etc) as well as utilizing third party frameworks for the HTML to accelerate the website building and additional third party APIs to incorporate functionality from other websites and content providers such as Twitter, Youtube and Facebook.

Our primary use for the javascript will be to have an interactive and eye-catching website with functionality from embedded youtube content and collecting and utilizing potential streams of gameplay.

We will also be using Javascript to implement several APIs and for tracking our viewer count.

Network Replication

Initially we will implement rounds in the gameplay as seamless travel; this is mainly because non-seamless travel is a blocking call and we do not wish for the client to disconnect from the server and then reconnect. Seamless travel is set up by configuring the Unreal Game Maps Settings and setting up and creating a transition map. The transition map has minimal overheads and is quite tiny and unnoticeable to the player. This way we can easily carry over the actor's from level to level or instance to instance.

One thing to note is that the server does not replicate every single actor update as this would take too much bandwidth and CPU resources; instead the server replicates actors at a specified frequency that's on the "AActor::NetUpdateFrequency" property.

To make up for the lack of information between each update, the client will simulate the actor between the updates; if we're simulating NPCs then we use 'role_simulatedProxy' and if we're simulating an actor that is possessed by a player controller, we use 'role_autonomousProxy'.

The health, projectile travel, movement, incoming damage and other such actions will all be replicated after the dedicated server is up and running as we will need the server there as a foundation to be able to test and run the game. The most important line to run is 'bReplicates = true;' inside of the constructor in the .cpp MP files; where MP is shorthand for multiplayer in the file name for identification purposes.

RPCs

Features that are cosmetic in nature will be replicated using remote procedure calls; this will be for any particle effects, features regarding projectiles and potentially sounds and other temporary effects (damage types, damage over time) that aren't critical to the actors functioning.

We will try to call our RPCs on the server and execute them on the client; this will be to try and lower the risk of false RPCs or malicious RPCs sent from the client to the server to alter the game's state unfairly.

By default, RPCs are unreliable due to the nature of data transfer and to ensure that there is no packet loss and that an RPC is executed on the remote machine as intended, we can specify the 'Reliable' keyword in the function definition; here is an example below:

```

UFUNCTION (Client, Reliable)
Void ClientRPCFunction();

```

C++

We will use C++ to program both the game itself and the network replication; the game itself will initially be made in blueprints and those blueprints will then be converted into C++ code.

Here is an example of some RPC function that could be an implementation function and then act as a validation function.

```

UFUNCTION (Server WithValidation)
void someRPCFunction (int32 AddHealth) ;

bool someRPCFunction_Validate( int 32 AddHealth)
if (AddHealth > Max_Add_Health) {
    return false; }    //disconnects who made the RPC
return true;          //allows it to be called

void someRPCFunctionImplement (int32 AddHealth) {
    Health = Health + Add Health; }

```

The first two lines are the RPC being made with validation. The next four lines are a validation function, which checks to make sure the values are okay to pass; then the last two lines are the RPC being called to do its purpose. In this example, it is adding 32 health to the player. In this context, the purpose of the RPC is to not allow the player to heal their health greater than their health upper limit.

Dedicated Server

The only requirements for a dedicated server are the project being entirely in C++ that has been designed to support server-client multiplayer gameplay. The dedicated server will be simple enough to create, it will be the designing of the frontend and providing a means for players to connect to the server over the internet; once this has been implemented and done, we can begin replicating the individual features of the project.

The following is some basic code to design a basic server for a test project and you can replace any existing files 'NAME.Server.Target.cs' with this.

```
using UnrealBuildTool;
```

```
using System.Collections.Generic;
```

```
public class TestProjectServerTarget : TargetRules //Change 'TestProjectServerTarget' to your  
projects name
```

```
{
```

```
    public TestProjectServerTarget(TargetInfo Target) : base(Target) //Change  
'TestProjectServerTarget' to your projects name
```

```
{
```

```
    Type = TargetType.Server;
```


```
    DefaultBuildSettings = BuildSettingsVersion.V2;
```

```
    ExtraModuleNames.Add("TestProject"); //Change 'TestProjectServerTarget' to your projects  
name
```

```
}
```

```
}
```

After this we just need to set up the default maps for the project to run so when players connect to the server they're instantiated into a level in a valid game state. Then it's time to package the project and build the target as a server.



The unreal documentation is incredibly in depth and this is a very rough overview of the general implementation we would have to do to set up the dedicated server for our project.