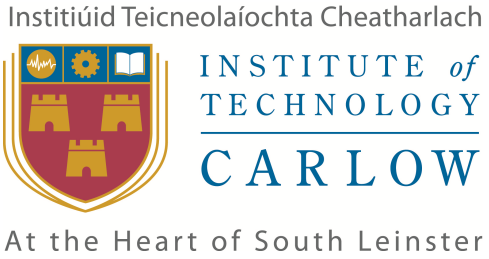


Computer Games Development CW208
Technical Design Document
Year IV

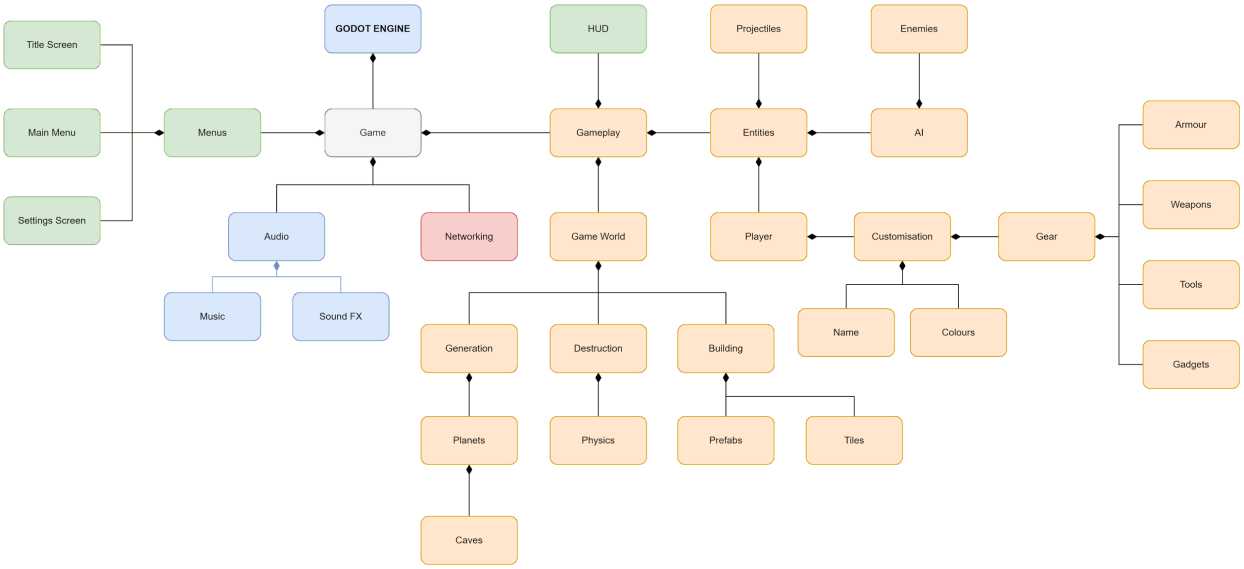
Michael Rainsford Ryan
C00239510
28/04/2022

Contents

Technical Design

Architecture

Overview Diagram



Blockchain

Enjin

Enjin is a blockchain service designed for video games, providing the usual features such as creating, minting, and sending Fungible and Non-Fungible tokens, but also user management and authentication.

Enjin provides platform-specific SDKs, as well as providing a general GraphQL API. While there used to be an official Enjin SDK for Godot [1], it has long since been deprecated and is completely out of date.

The official documentation is both vague and out of date, and as such, nearly everything I learned about the platform was from trial and error. The following sections are everything I gathered while working with the platform.

Enjin is broken into several pieces to provide different levels of authorization and management.

App

Rather intuitively, an app is a container for token and user data for a specific game or application. After creating an Enjin account, the next step is to create an app for your project.

API requests can be authenticated with either a user or app access token. While most requests can be made with either, some requests (such as user creation) must be authenticated with an app access token.

Access Tokens

Access tokens are used to authenticate requests on the platform, provide different levels of authorization depending on their source, and can be received in a couple of different ways. The different types of tokens and their source are as follows:

- App access tokens can be received with the AuthApp query by providing an app's id and secret (similar to a password). App access tokens can authorise any requests specific to an app or its users.

- Enjin User access tokens can be received with the EnjinOAuth query by providing the user's email and password. These access tokens can be used to authenticate requests specific to any apps they own (with some limitations, such as user creation), and some requests (such as sending tokens) within other apps they have identities with.

- Player User access tokens can be received with any query that returns an EnjinUser object (such as the EnjinUser query) using a relevant Enjin User access token or the

app access token. These tokens can be used to authenticate some requests specific to this user within the app (such as sending tokens).

-

User

‘User’ can refer to either an Enjin account that can be used in any number of applications or an app-specific user that must be created through the API. The data is the same for both, with the distinction being a flag ‘isPlayer’, where Enjin accounts are not players, and app users are players.

Non-player users can request access tokens through a login query, whereas player users must receive the access token from the owner of the app as mentioned in the previous section.

Users contain a number of fields, notably a name, a number of identities, and a collection of tokens. Non-player users can have a wallet address linked to any associated identities, but cannot authenticate spending from that wallet on any apps, whereas player users can.

A workaround for the above issue, allowing users to authenticate themselves with an email and password and also make transactions within an app, is to link a wallet to the Enjin user’s app identity, then create a player user and link the same wallet to the new user’s app identity, allowing the player to link their wallet to the app. Once the wallet is linked to the app, it can be used in any linked identity regardless of user type.

Identity

Identities share a many to one relationship with both apps and users (e.g. each user can have multiple identities but each identity only relates to one user) and are used for linking wallets and making transactions. Identities are, in a way, a wrapper for a wallet, and in many queries, identity ids and wallet addresses are interchangeable.

Wallet

Enjin uses a platform specific wallet [2] for managing assets and validating requests. A wallet can contain many Ethereum wallet addresses which can be linked to identities for making transactions. An Enjin wallet address is no different from any other Etherium wallet address and can be used interchangeably, though the wallet app is necessary to use the platform (Needs confirmation).

Asset

An app may contain any number of assets, which can be either Fungible or Non-Fungible Tokens. Each asset is made from Enjin Coin and can be melted back into them if needed.

When the items of a user are queried, all their assets from any linked wallets are returned/

Transaction

Enjin transactions are used to control anything blockchain related on the platform, such as

- Creating tokens,
- Minting tokens,
- Sending tokens,
- Requesting permission to spend crypto from a linked wallet,
- And many more.

All transactions must be validated from the wallet in question.

Enjin API Class

The project implements an Enjin API wrapper to allow other classes within the game to easily access the blockchain aspects without having to directly make GraphQL requests. Classes receive the requests' responses through signals that they can listen to from the Enjin API class.

The Enjin class provides the following functionality:

- Connect to an Enjin Network (regular or test network)
- Login
- Logout
- Create Identity (user's app-specific data).
- Get User Data
- Mint Tokens
- Send Tokens
- Get Token Balance

If HTTP requests are made before the scene tree is fully initialised (i.e. as a scene is loading in), it will throw an error. To amend this, all calls are added to a queue and executed once the scene tree is fully initialised.

Development of this wrapper class began as a separate Godot addon, but development was later moved directly to the game due to time constraints. The add-on repo can be found here [3].

Usage

The API class is used on the login screen to authenticate users and link their wallets. Users must first create an Enjin account and login on the app, then, if they do not have an identity or linked wallet for the game, they will be asked to provide their wallet address, which will be linked (using the workaround to the Enjin user wallet linking problem mentioned above).

Once a user has logged in, their balance of the game's crypto currency token ‘Elixirite’ is queried, and displayed on the ship menu screen.

When players enter an online game, their balance is queried once again, and they’re asked to pay a fee of 1/3rd of their balance. Once a player returns to their ship after collecting minerals in a game, the appropriate number of tokens are minted and sent to their linked wallet.

GraphQL

For making GraphQL requests and receiving the result, a simple Godot GraphQL addon was used. The addon uses HTTP nodes to send the requests and returns the results through signals.

The addon was partially modified for this project, changing the way query objects were serialised into a GraphQL request to make it easier to write the queries in Godot, and additional modifications were made to allow for the attachment of access tokens.

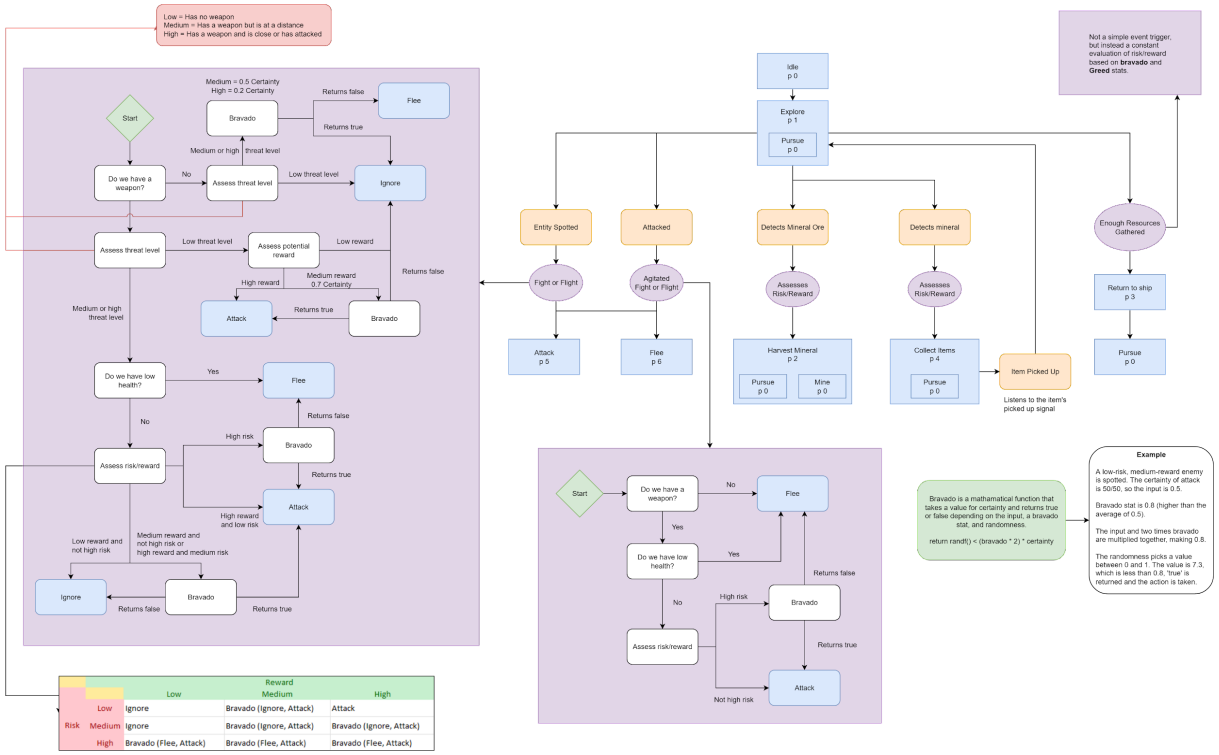
AI

Overview

The AI in the game uses a prioritised Finite State Machine for the AI's Behaviours, uses Sensors to detect the state of the world around them in a limiting fashion and utilises decision trees in interactions with other entities to decide whether to attack, flee, or ignore them.

Additionally, the AI has a number of stats, such as Bravado and Greed which are randomly generated upon instantiation and are used in determining the levels of risks the AI will take.

Both the player and the two types of AI inherit from a base entity class, which implements all the physics, inventory, and weapon & equipment logic and provides an interface for the player and AI scripts to control it through.



[Diagram File]

Behaviours

Each of the AI's behaviours are the FSMs states, which it switches between using a stack, meaning that once a behaviour finishes, it simply pops itself off the stack and the previous behaviour takes over.

Each behaviour is additionally prioritised, allowing behaviours and sensors to either outright add another behaviour to the stack, or request to add one. If a behaviour is requested, it will only be added if it has a higher priority than any other behaviour already on the stack. This means that while an AI is fleeing, it will not stop to collect items, nor will it stop to mine more minerals on its way back to its ship, though it will collect items. The priority value of each behaviour can be seen in the graph above.

Sub-Behaviours

Each behaviour can be added as either a regular behaviour, or as a sub-behaviour. While only the top behaviour on the stack will run at once, any sub-behaviour on the currently running behaviour will run simultaneously.

Sub-behaviours are used, for example, to allow the AI to move towards minerals and mine them at the same time.

Idle

The idle behaviour is automatically added to the AI's state machine upon initialisation. The idle state doesn't do anything itself, but is there to avoid having a null state at any time. When a behaviour is removed, if there's no more behaviours on the stack, an idle behaviour is added.

Explore

The explore behaviour utilises the *Spatial Sensor* to directionally explore the cave. The behaviour causes the AI to move towards the lowest point it can see or has seen at any time using the 'pursue' sub-behaviour, but can easily be modified to explore in any direction simply by changing its preferred direction.

The explore behaviour is the main branching point into other behaviours. The behaviour listens to signals from the *Mineral Sensor* and *Item Sensor* and switches to the appropriate behaviour when something is spotted.

Harvest Minerals

The harvest minerals behaviour loops through all the spotted minerals from the *Mineral Sensor* and moves towards the closest one using the 'pursue' sub-behaviour. Once within range of the mineral, a 'destroy cell' sub-behaviour is added. Once the target cell is destroyed, the behaviour will find the next best cell.

If the AI detects another entity too close to the minerals before they have already set their target on it, they will ignore it and move on to avoid conflict. Additionally, the behaviour listens to a signal on the *Item Sensor* and moves to pick up any items spotted while mining.

Collect Items

The collect items behaviour loops through all the spotted items from the *Item Sensor* and moves towards the closest one with the 'pursue' sub-behaviour. If an entity is spotted too close to the item, the AI will ignore it to avoid conflict.

Return to Ship

To be written.

Attack

The attack behaviour simply moves towards (using the 'pursue' sub-behaviour) the highest priority detected threat from the *Threat Detector* and shoots continuously until the target is dead. The AI listens to the target's death signal to know when to give up. If there are other threats still when the main target is killed, the behaviour will continue and the AI will move on to the next target.

Flee

The flee behaviour uses Dykstra's algorithm to expand outward from itself within a range and identifies the furthest non-dead end it can take that will immediately bring it further from any detected threats in the *Threat Detector*. If the AI has multiple threats, it will move in the direction with the largest average distance from them all.

The behaviour then moves with the 'pursue' sub-behaviour towards its chosen target until out of range of threats.

Pursue

One of the most common behaviours and also one of the most complex. This behaviour, at a high level, moves the AI towards a set target following the shortest path.

The behaviour generates a path to the target using Godot's built-in A* implementation and follows each node of the path, regenerating it if strays too far from the first path node or if the AI is blocked.

When generating the path, any entities detected by the *Entity Avoider* are marked as obstacles and will be avoided if possible.

Destroy Cell

The destroy cell behaviour simply activates the entities drills and guns until the target cell is destroyed. This behaviour is used exclusively as a sub behaviour of more complex behaviours.

Sensors

Instead of giving the AI access to all the data in the game and having it act with godlike intelligence, they instead see the world through sensors. Each sensor specialises in specific areas and provides signals that behaviours can listen to to be notified of events.

Spatial Sensor

The spatial sensor is used by the explore behaviour to chart its path forward. The sensor uses something similar to Dykstra's algorithm to detect the edges of the caves within a range. The sensor keeps track of all empty terrain tiles at the edge of their explored area in a priority queue, sorted by whichever is furthest in their target direction, which allows them to explore the caves directionally.

The explored area is widened any time the AI moves by a tile or so from the last space the algorithm was called. Any time a new best tile is found, a signal is sent out to notify any listening behaviours (which in this case, is the explore behaviour).

Mineral Sensor

The mineral sensor uses a raycast that continually rotates around the AI and detects mineral cells within its range. Any minerals spotted are added to a container and a signal is sent out to any listening behaviours.

Item Sensor

The item sensor simply uses Godot's Area2D to detect physics bodies on the Item physics layer with a range. Any items spotted are added to a container and a signal is emitted. The sensor also listens for the items 'picked up' signal and removes the item from the container when it's picked up or when it leaves the sensor's range.

Entity Sensor

Similar to the item sensor, the entity sensor listens uses an Area2D to detect physics bodies on the Entity physics layer, adding them to a container and emitting a signal upon spotting a new entity.

The sensor listens to the entity's death signal and removes them from the container when they die or leave the sensor's range.

Threat Detector

The threat detector listens to the *Entity Sensor*'s signal to evaluate newly spotted entities within range, while also using its own, smaller collision area to detect closer (and thus more threatening) entities.

The threat detector uses the Fight or Flight and Agitated Fight or Flight decision trees to determine the AI's course of action in regards to entities. When an entity is spotted by the *Entity Sensor* or comes within range of the threat detector itself, the Fight or Flight tree is called with the entity in question being passed into it.

The detector also listens to the AI's damaged signal and runs the Agitated Fight or Flight decision tree upon being attacked, passing the attacking entity in as an argument.

Entity Avoider

The entity avoider detects entities in a small range around them and applies a small force away from them each physics update. It also keeps track of all entities within this radius, that's then used the 'pursue' behaviour's pathfinding to avoid entities that are in the way.

Decisions

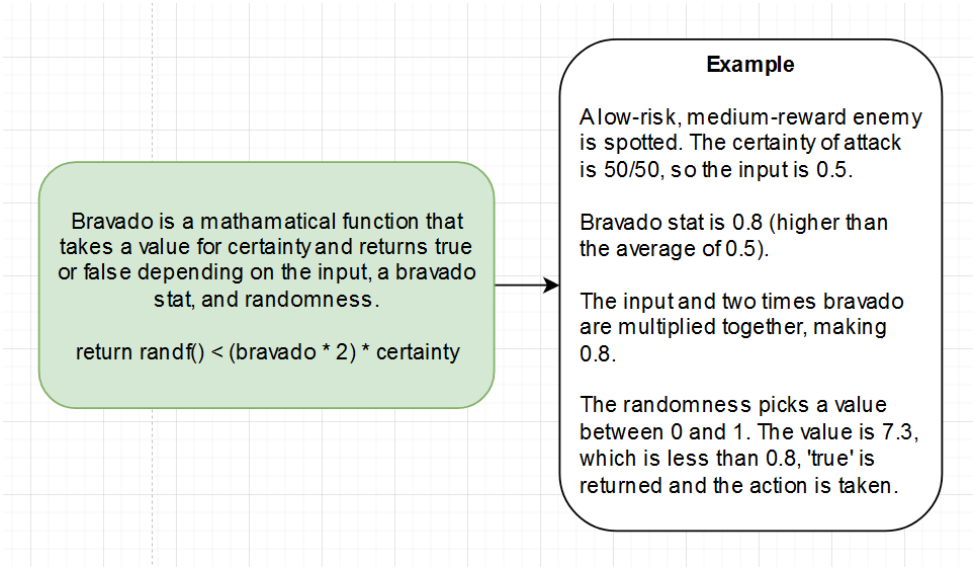
The decision of whether to attack, flee, or ignore another entity is decided by a decision tree that takes many aspects of the AI's state and surroundings into account. When an entity is first detected or comes within a certain threshold, the Fight or Flight decision tree is run on the entity to evaluate the course of action.

When an AI is hit by a bullet, the Agitated Fight or Flight decision tree is run on the entity. This tree provides less options and takes less into account.

Bravado Function

Both trees use the Bravado and Greed stats to determine the risks they're willing to take. The greed stat paired with the number of collected resources determines the risk of the situation,

which is fed into a Bravado function that uses the risk as a certainty value and calculates it with randomness to compute a true or false value.



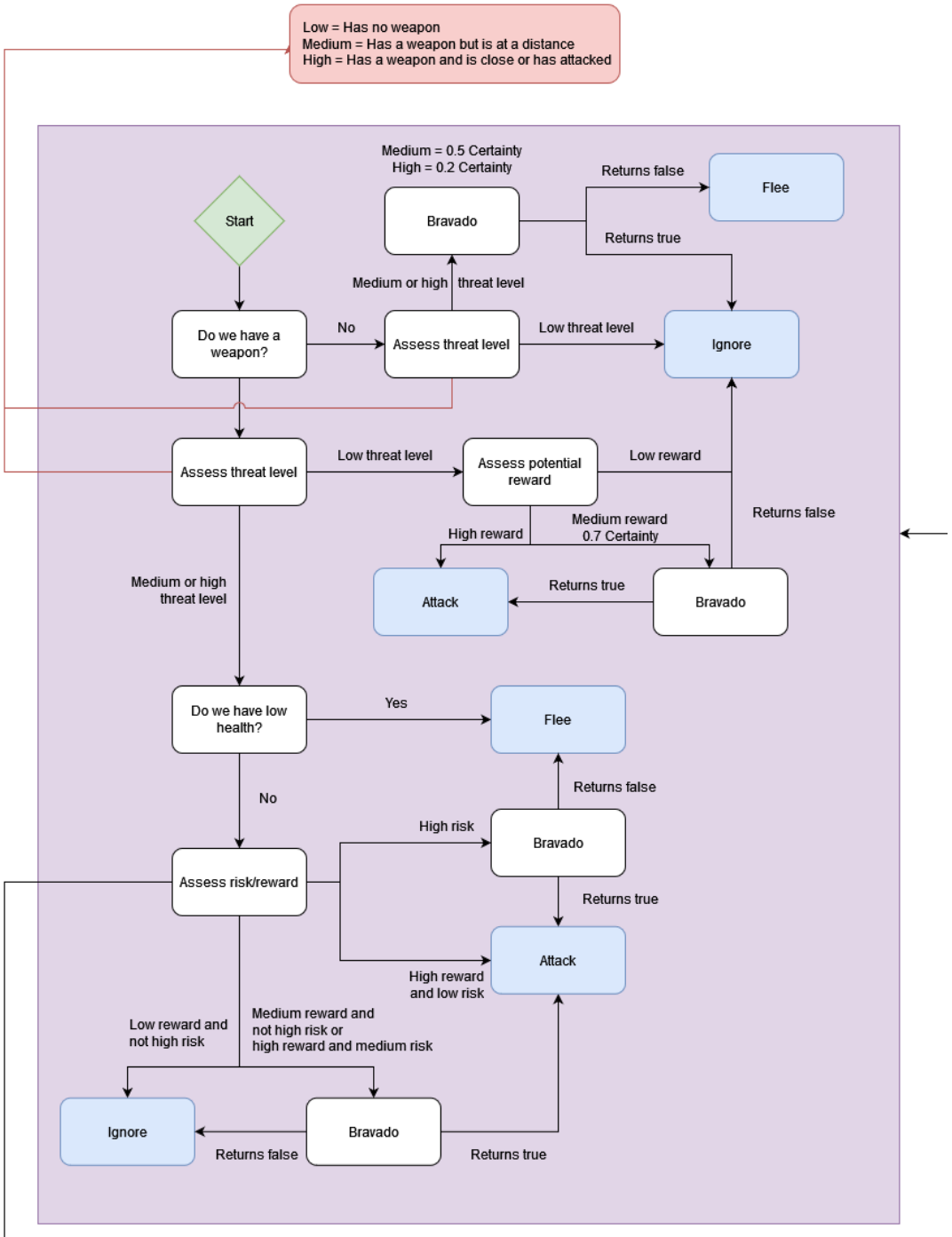
Decision Node

The decision trees are made up of decision nodes that all implement a simple interface. Each node has a container of ‘branches’ which contain more decision nodes mapped to an output value of the parent node.

```
11  # ------------------------------------------------------------------------------
12  class DecisionNode:
13
14      var _branches = {}
15
16      # ------------------------------------------------------------------------------
17      func make_decision(_brain : AIBrain, _entity : Node2D) -> void:
18          assert(false)
19
20      # ------------------------------------------------------------------------------
21      func map(key, value : DecisionNode) -> DecisionNode:
22          _branches[key] = value
23          return self
24
25      # ------------------------------------------------------------------------------
26      func _branch(result, brain : AIBrain, entity : Node2D) -> void:
27          if _branches.has(result):
28              _branches[result].make_decision(brain, entity)
29          else:
30              print_debug("No branch mapped to result of " + str(result) + ".")
31
```

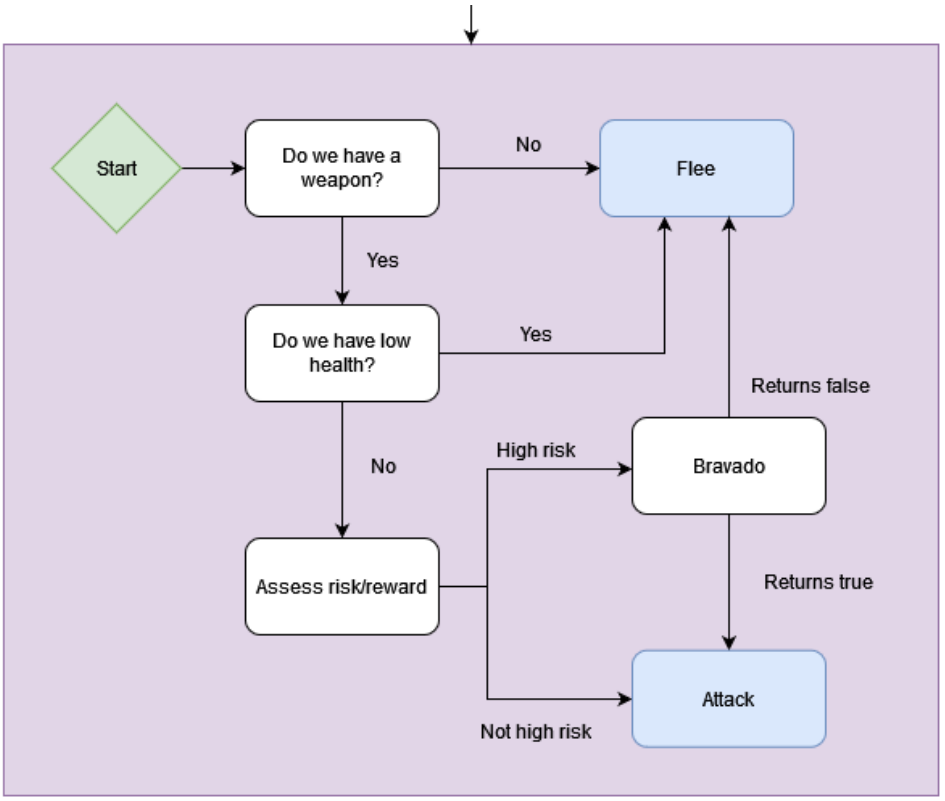
Fight or Flight Decision

Code implementation left out due to length but can be found here.



		Reward		
		Low	Medium	High
Risk	Low	Ignore	Bravado (Ignore, Attack)	Attack
	Medium	Ignore	Bravado (Ignore, Attack)	Bravado (Ignore, Attack)
	High	Bravado (Flee, Attack)	Bravado (Flee, Attack)	Bravado (Flee, Attack)

Agitated Fight or Flight Decision




```
# Creates the agitated fight or flight tree.
_agitated_fight_or_flight = fof.HasWeapon.new() \
    .map(false, fof.Flee.new()) \
    .map(true, fof.HasLowHealth.new() \
        .map(true, fof.Flee.new()) \
        .map(false, fof.AssessRisk.new() \
            .map(fof.Values.LOW, fof.Attack.new()) \
            .map(fof.Values.MEDIUM, fof.Attack.new()) \
            .map(fof.Values.HIGH, fof.AssessPotentialReward.new() \
                .map(fof.Values.LOW, fof.CalculateBravado.new(0.2) \
                    .map(false, fof.Flee.new()) \
                    .map(true, fof.Attack.new())) \
                .map(fof.Values.MEDIUM, fof.CalculateBravado.new(0.5) \
                    .map(false, fof.Flee.new()) \
                    .map(true, fof.Attack.new())) \
                .map(fof.Values.HIGH, fof.CalculateBravado.new(0.8) \
                    .map(false, fof.Flee.new()) \
                    .map(true, fof.Attack.new())))))
```

Procedural Generation

To keep each individual game interesting, the world is procedurally generated for each one. The generation is implemented as a Godot ‘tool’ meaning it can not only run in game, but also in the editor, allowing for instant feedback when changing generation parameters.

The generation uses four perlin noise objects to generate different aspects of the world as follows:

- Surface height,
- Horizontal cave movement,
- Cave radius,
- Mineral distribution.

As well as providing more concrete generation parameters such as min and max cave number, the generator also exposes the parameters of each noise object to allow for fine grained modification.

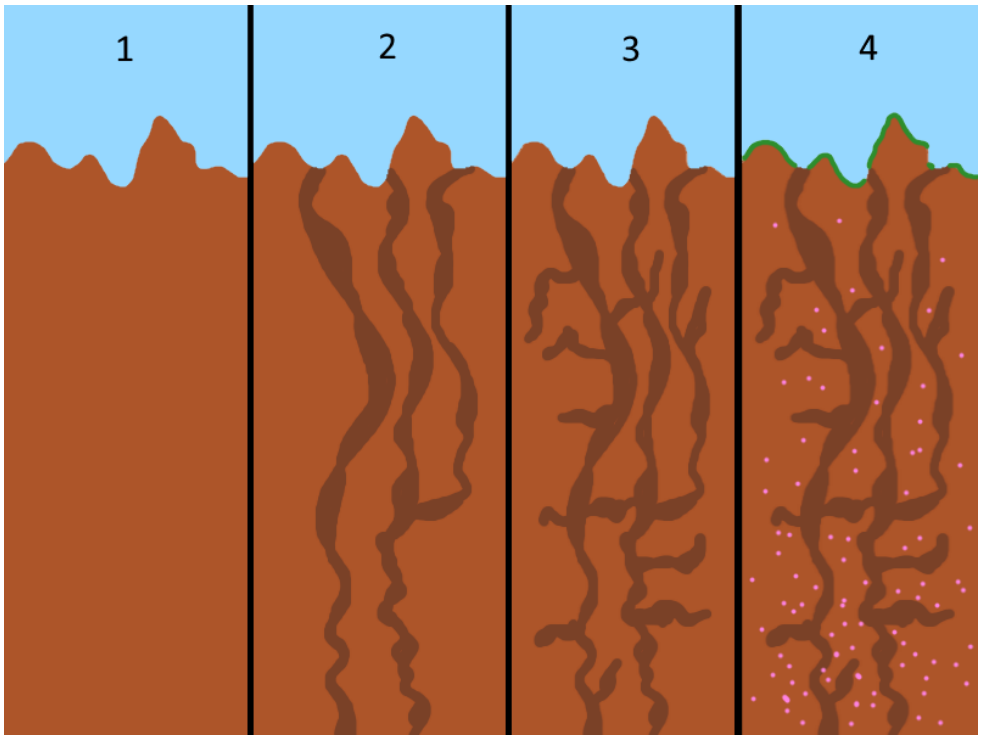
Simplex Noise

The generation uses Open Simplex Noise to generate “smoothed randomness”. Open Simplex Noise is essentially an open source version of the popular Perlin noise and is included out of the box in Godot.

More about Open Simplex Noise can be found here [4].

Generation Process

The generation happens in four main steps as follows:



1. 1D Noise is used to generate the height of the surface given an x position as the argument.

2. The number of main caves to generate is decided, and the world is split into that many segments. All the surface positions for each segment are prioritised by lowest first. The start position is then randomly picked from the best ¼ of options and generated downwards to the bottom of the level, using 2D noise to decide its divergence from the start point, passing the current x and y as parameters. The radius of the cave at any point is sampled from a radius noise object, again passing x and y and parameters.

3. Additional caves are branched off of the main caves when the verticality of the cave is at a certain (customisable) threshold, at which point a random check is used to decide whether or not to generate a cave off of it. The cave length is decided randomly upon beginning the generation of it and its radius is once again sampled from the radius noise object.

4. Finally, grass is laid upon each surface tile, and crystals are spread throughout the caves, using noise and depth to decide their frequency. Once all tiles are placed, the textures of each are updated to match their surrounding tiles.

Physics

The physics of the game is mostly handled automatically by Godot, with the velocity being calculated manually by each physics body. The game features mostly static objects, with the exception of a couple physics objects.

Character Movement

Each character in the game has a number of public properties that are used to customise the way they move. The following parameters are used to generate the character's gravity, jump speed, acceleration, and deceleration:

Max Speed	50
Time To Max Speed	0.1
Time To Full Stop	0.1
Jump Height	32
Time To Jump Peak	0.6

The vertical movement is fairly standard addition of gravity multiplied by time, and the horizontal movement calculation is as follows:

```
203  # ------------------------------------------------------------------------------
204  func _handle_horizontal_movement(delta):
205
206      # Applies deceleration if no movement input.
207      if direction == 0.0:
208          velocity.x -= min(_deceleration * delta, abs(velocity.x)) * sign(velocity.x)
209
210      # Else applies acceleration in the movement direction.
211      else:
212          velocity.x += _acceleration * direction * delta
213
214      # Clamps the horizontal movement to the max speed.
215      if abs(velocity.x) > _max_speed:
216          velocity.x = _max_speed * sign(velocity.x)
217
```

Physics Objects

Drop Pod

The drop pod is the way each player and AI enters and exits the game. The pod begins at the top of the world and drops to the surface. The physics of the ship are more so interpolation than a physics simulation to give a better cinematic experience.

The ship falls at a constant speed from the top of the world until it comes within 100 tiles of the ground, at which point it starts decelerating at a rate dependent on its distance to the ground. This gives the impression of a landing sequence.

When landing, if the ship lands on an uneven surface, the highest cells will be removed until all cells underneath the pod are the same height.

Upon taking off, the ship takes one second to reach maximum speed, allowing it to appear heavy at take off, but not waste time in bringing the player out of the game.

Once landed, the ship is mostly stationary unless the ground underneath it is removed, at which point it will fall with gravity.

Bullet

Bullets within the game simply move in their facing direction with a constant velocity until hitting a solid or travelling a certain, predetermined distance.

Item

Items react with gravity and acceleration much the way the characters do, though they have no driving force, meaning they will not move horizontally unless propelled (by sliding off a surface or similar).

When a character comes within a small range of an item, they will start applying an acceleration on the item in the direction of the character. The magnitude of the acceleration is exponentially dependent on the distance, with closer items moving far faster than those further away.

Multiplayer

The multiplayer in the game uses Godot's built in multiplayer to sync up the state of the game across all connected clients. The server is mostly authoritative, with some aspects taking a more peer-to-peer approach where performance was an issue (such as player movement).

The server application uses the same project and code as the client, with different scripts running upon initialisation as a server. An instance of the project can be set as a server by changing the IS_SERVER constant on line 11 of Network.gd to true.

Server Hosting

The server was originally hosted on a DigitalOcean Ubuntu server in the early days of development, but had to be taken down due to the high costs of maintaining it.

References

Website

[1] Lindsay, E. (2020, November 13). Enjin Godot SDK [Online]. (URL https://github.com/enjin/enjin-godot-sdk). (Accessed 2021, November 10).

[2] ???. (???). Enjin Wallet [Online]. (URL https://enjin.io/products/wallet). (Accessed 2022, April 8).

[3] Rainsford Ryan, M. (2022, March 16). Godot Enjin API [Online]. (URL https://github.com/MichaelRRyan/godot-enjin-api/tree/dev-v0-1-0). (Accessed 2022, March 16).

[4] Linietsky, J. (???). Godot Open Simplex Noise [Online]. (URL https://docs.godotengine.org/en/stable/classes/class_opensimplexnoise.html). (Accessed 2022, April 27)