Game Design Document

Snatcher

Spring Quarter 21-22

Table of Contents

Overview

High Concept

“Snatcher” is a 2.5D isometric action explorer where you play as the Snatcher.

The main objective of the game is the completion of room-to-room dungeons against environmental and movement-based challenges to advance to the next level until the final challenge dungeon.

Category

“Snatcher” is set to be a single-player real-time action, exploration, and puzzle-based game.

Audience

This game is targeted at those who enjoy a fair challenge. This group typically consists of core gamers that are previously experienced in games with more sophisticated mechanics.

Accessibility

This game is set to be developed for PC along with controller support.

Budget

This game was given an initial budget of 343,350 DB (Diefenbucks). For a detailed breakdown of spending, please visit the Gantt chart. A projected increase in budget is yet to be discussed at this time, but the decision is to be made on executive review.

Timeline

This game was given an initial development cycle of 10 weeks. For a detailed task-oriented breakdown of events during this development period, please visit the Trello and the Gantt chart. For a by-teammate recollection of their work weeks during this development period, please visit our team’s website at https://sites.google.com/view/tbagames. A projected extension in the development cycle is yet to be discussed at this time, but the decision is to be made on executive review.

Getting Started

Welcome to the team! Below you will find the current team members and the different platforms we use to communicate, work on the game, and keep everything organized.

TBA Games Team

Mahin	Production Lead	**Erica**	Technical Director/Scrum Master
Ben	Lead Programmer	**Adam**	Lead Programmer
Aaron	Writer/Programmer	**John**	Level Designer
Sandalu	Programmer/QA	**Sarah**	Lead Artist/Modeler
Wuheng	Programmer/Modeler		

Communication and Meetings

The team will be mainly using Discord to hold meetings and communicate regularly. Weekly recurring meetings are held on Mondays, Wednesdays, and Fridays.

Team Site

This is where we showcase our work to the public including things like weekly versions of our build, the team blog, this document itself, and more. Be sure to add yourself to the 'Meet the Team' and 'Team Blog' section. Message the team on Discord for access to edit. Visit our team site at https://sites.google.com/view/tbagames. Once you've gotten settled, make sure to update your personal blog every week.

Tasks and Planning

The team will be using Trello to detail and assign tasks. The Gantt chart will be used to plan and keep track of where we are in development. Message the team on Discord for access.

Collaboration and Organization

The team will be using Google Drive to share and work on documentation and presentations, and will be using GitHub for version control, to work on the game's codebase and to manage our assets. Message the team on Discord for access to these resources.

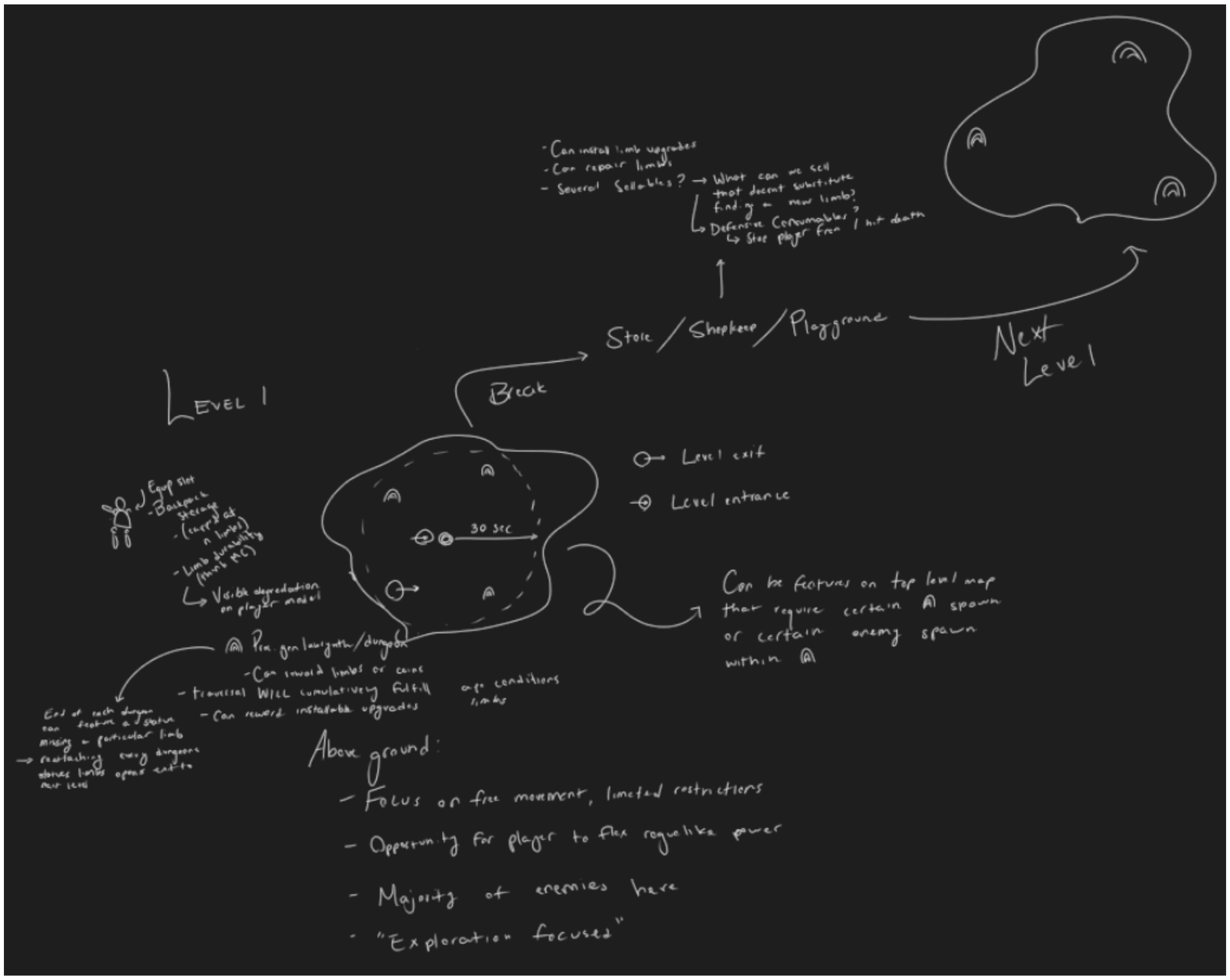
Development

We will be using Unity 2021 LTS as our choice of game engine to develop our product.

Core Gameplay

Game Flow

The player spawns as Snatcher in an outdoor overworld area where entrances to multiple wells are scattered about. Each well leads to a dungeon variant that holds a key. Retrieving these keys in the various dungeons allows the player to advance to the corresponding well/dungeon until they retrieve the key in the final well to pass the current overworld into the next overworld with new dungeons. This is repeated until they reach the final challenge dungeon.



- Can install limb upgrades
- Can repair limbs
- Several Sellables? → What can we sell that doesnt substitute finding a new limb?
↳ Defensive Consumables? ↳ Stop player from 1 hit death
Store / Shopkeep / Playground
Next Level
Break
Level 1
Equip slot
- Backpack storage (carry at n limbs)
- Limb durability (Minish MC)
↳ Visible degradation on player model
30 sec
Level exit
Level entrance
Can be features on top level map that require certain spawn or certain enemy spawn within
Proc. gen labyrinth/dungeon
- Can reward limbs or coins
- traversal WILL cumulatively fulfill age conditions limbs
- Can reward installable upgrades
End of each dungeon can feature a statue missing a particular limb
→ reattaching every dungeons statues limbs opens exit to next level
Above ground:
- Focus on free movement, limited restrictions
- Opportunity for player to flex roguelike power
- Majority of enemies here
- "Exploration focused"

Snatcher

Snatcher is a frog-like creature. Snatcher can launch a hook to snatch limbs from various enemies to use as his own. The abilities granted from the various limbs will be used to help with enemy encounters and terrain obstacles to progress through the dungeons.

Character Controls

Keyboard	**Controller**	**Action**
WASD	*Left Analog Stick*	*Directional movement*
Space	*East Button*	*Dash*
Left Click / Left Mouse Button	*Left Trigger*	*Use current limb*
Q / E or Scroll Wheel	*Shoulder Buttons*	*Swap current limb*
X	*West Button*	*Drop current limb*
F	*South Button*	*Interact with game object*
ESC (Escape)		*Pause/Unpause*

Enemy

Enemies cannot be killed, however some can have their limbs stolen. Current enemies with snatchable limbs are the chameleon, mushroom, and maple seed. There is a tiger enemy that has a different behavior associated with a limb, but no snatchable limbs at the moment. Enemies planned for future development include a gorilla/muscular animal, a snail, and a lamb. More details about enemies can be found in the Game Asset section.

Enemy Behavior

Current enemies all travel on the same level as the ground. Future plan is to have an additional unique behavior for each enemy based on the ability Snatcher gains from snatching its limb.

Standard Behavior

Enemy will be idle until it detects Snatcher in their sight. The enemy will chase Snatcher until Snatcher is out of its sight range.

Scent Tracking Behavior

Enemy will be idle and will not detect and chase Snatcher unless Snatcher is actively using a limb that stinks. Once the stinky limb is inactive, the enemy immediately stops chasing Snatcher and returns to its initial idle location.

Weight System

Each limb has its own weight. The more limbs Snatcher carries, the more Snatcher weighs. This lowers the overall speed of Snatcher. Individual limbs are also affected by this shown in the Obtainable Limbs section below.

Stamina System

Snatcher has a total of 100 stamina. Each limb has its own stamina cost on usage, it may be stamina per use or stamina drain for the duration of usage. Snatcher regenerates stamina over time. The more Snatcher weighs, the stamina cost on each limb will increase.

Obtainable Limbs

Limbs can be dropped and picked up at a later time.
The chart below contains ideas for limbs.
Bolded contents are currently implemented or in development

‘Limb’	Ability	Stamina Cost	Weight	Effect by weight
Chameleon’s tail	**Invis**	**5**	**5**	TBD
Mushroom’s legs	**Jump**	**15**	**3**	**Jump height decreased**
Maple Seed’s wings	**Flight**	**10**	**10**	**Distance gained decreased**
Gorilla’s arm	Destroy terrain/ shoves enemy	50	20	TBD
Snail’s shell	Roll	20	15	TBD
Lamb’s wool	Decoy	40	10	TBD

Chameleon’s Tail

This limb is stinky. When active, enemies with standard behaviors will not detect and chase Snatcher. Only enemies with scent tracking behavior will detect and chase Snatcher when the tail is used. Simply equipping the limb does not trigger this behavior.

Mushroom’s Legs

When used, Snatcher can jump. However, Snatcher’s walk speed becomes slower than usual.

Maple Seeds’ Wings

Snatcher can gain more and more verticality with each use of the wing. When equipped, Snatcher has a bigger hitbox and gets pushed back in windy areas of the map.

Challenges

Current

Enemy avoidance	Snatcher has to avoid or run from enemies.
Terrain navigation	Snatcher has to use one or more limbs to navigate the terrain.

Future

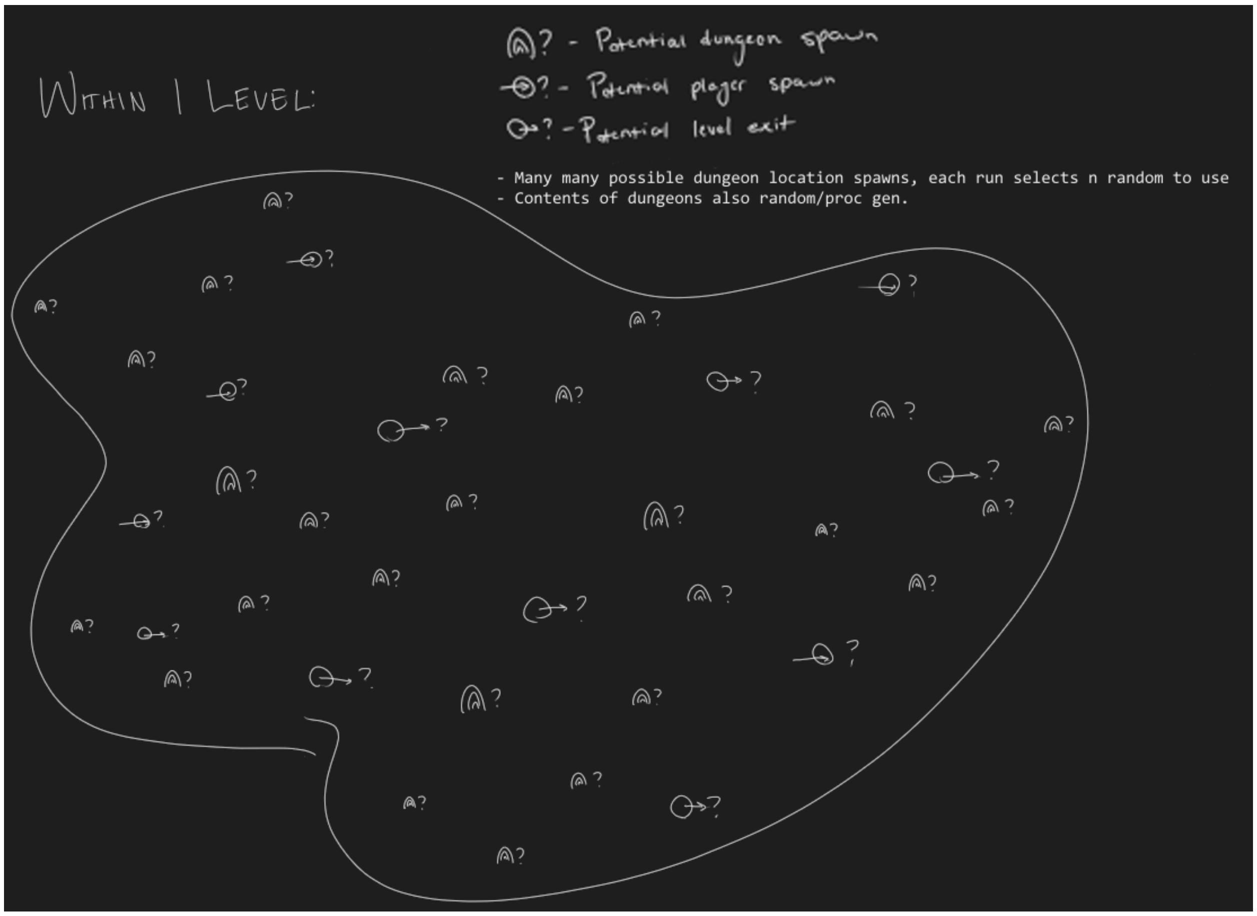
Race against time	Snatcher has to reach the goal before the toxic water rises to its level.
Pathmaking	Snatcher has to modify terrain using its limbs to reach the goal

Death

Snatcher dies from coming into contact with enemies or when it falls off the platforms into the depths below. When death occurs in the dungeon, progress is reset back to the state when Snatcher first entered the dungeon. When death occurs in the overworld, progress is reset for the current overworld. Keys and limbs obtained in the current overworld will be reset.

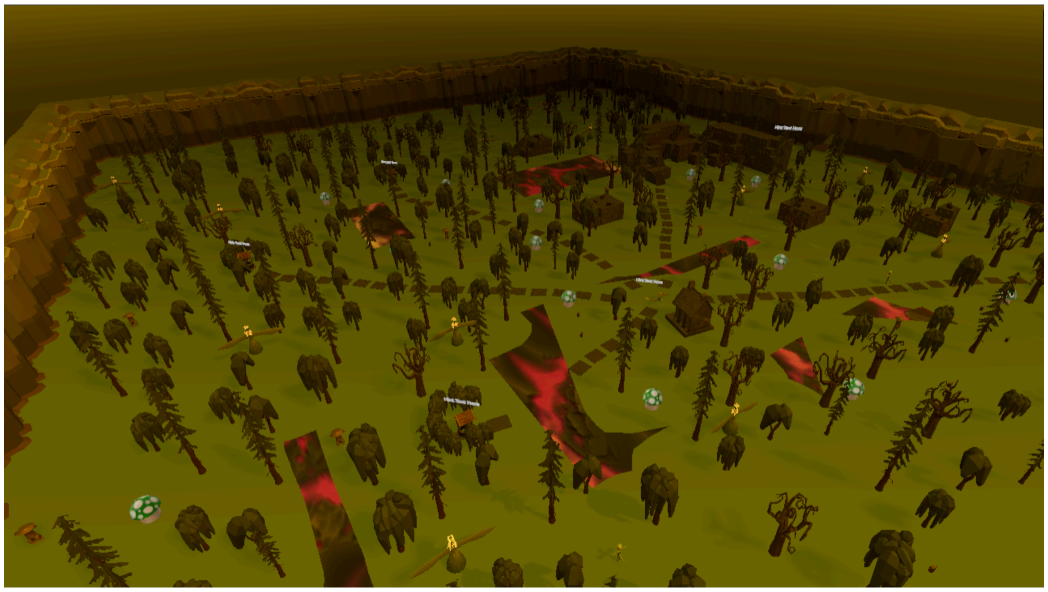
Level Design Overview

A level will consist of the main hub, currently being called the “Upper World", which will contain access to a varying number of sub-levels classified as “dungeons”, as well as an exit that will lead to future levels in planning. The current version of the game displays one of these “Upper Worlds” with four iterative “dungeons” attached. Each “dungeon” will contain a series of movement and enemy-based interactions players must navigate to locate the key for the next “dungeon”. Upon collecting all of the keys players will be provided access to the exit and in turn the next level or “Upper World”.


WITHIN 1 LEVEL:
? - Potential dungeon spawn
? - Potential player spawn
? - Potential level exit
- Many many possible dungeon location spawns, each run selects n random to use
- Contents of dungeons also random/proc gen.

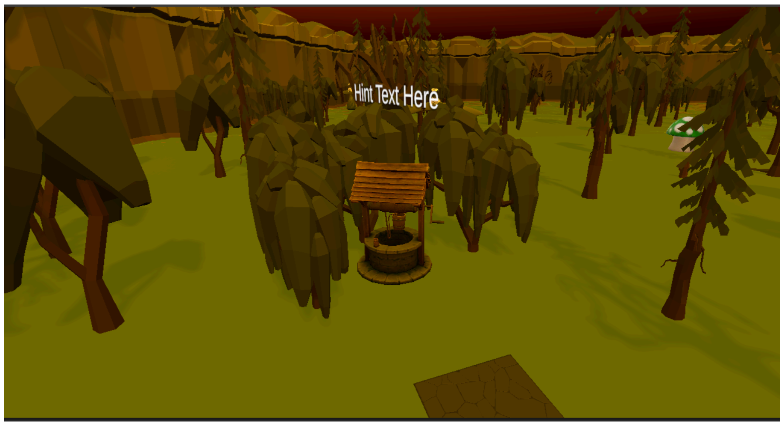
Upper World

The initial spawn for players will be a centralized point in the Upper World. Here players will be introduced to a brief background on the Snatcher and be allowed to explore the terrain and interact with various enemies they may find useful. Depending on the time taken to explore players will eventually discover the dungeon entrances and begin progression through the first sub-level.



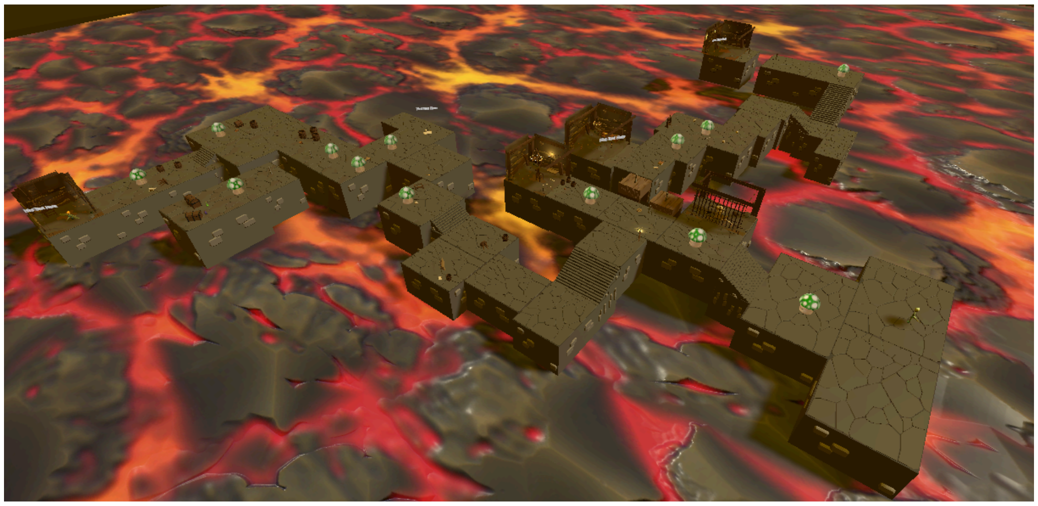
Wells

The wells are scattered around the Upper World and represent the transition points from the Upper World into the dungeons. A shift in audio will also signify the engagement of a new environment.



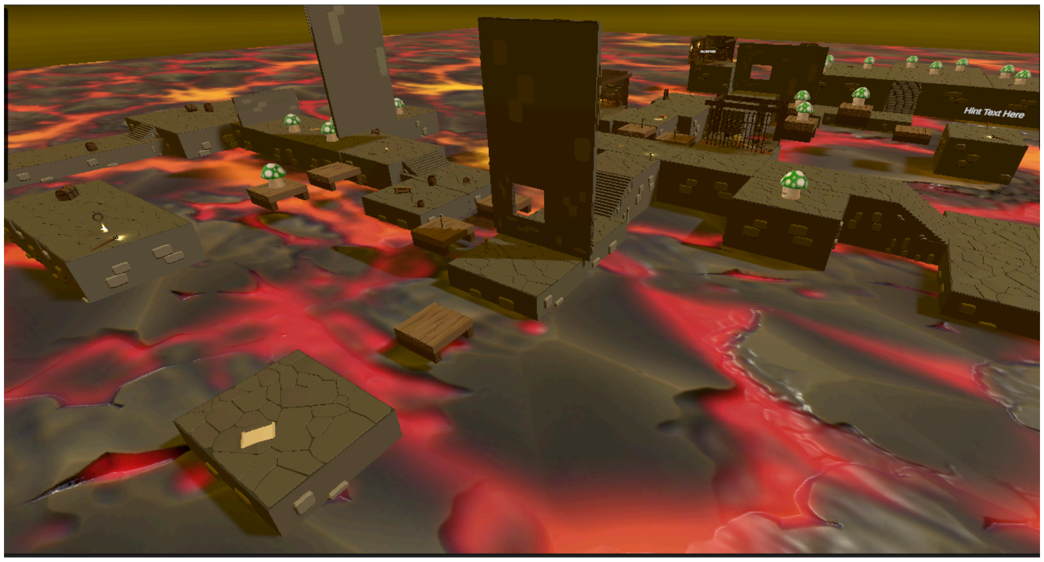
Dungeon 1.1

Upon entering the first dungeon players will find themselves in a much more confined space facing off against enemies. This section focuses on the player's understanding of stealth and familiarization with standard enemy movement, allowing for virtually easy maneuverability with the Chameleon's Tail. After finding and collecting the key here players will gain access to the second dungeon.



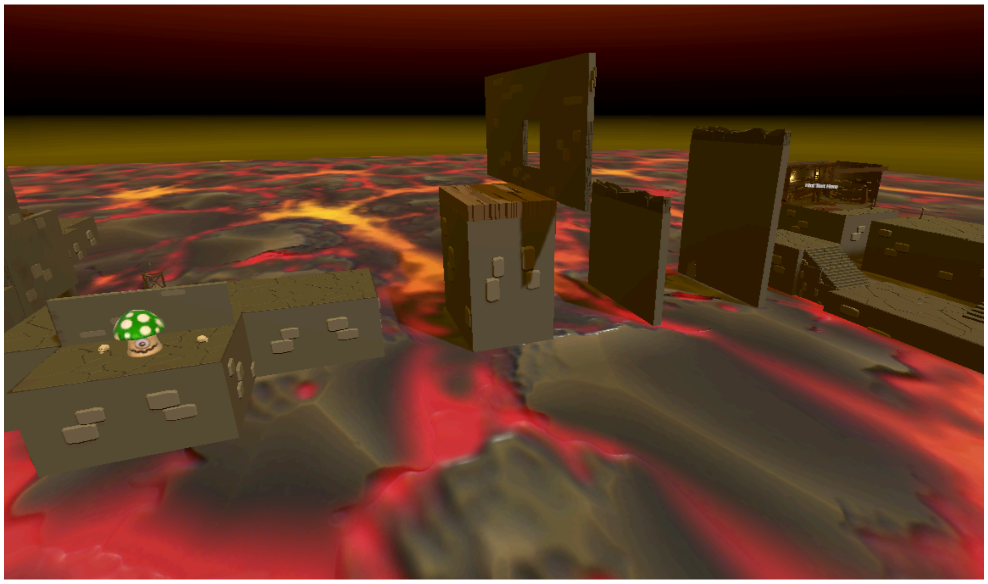
Dungeon 1.2

This section will introduce the player to more movement-based challenges on top of the enemy interactions. It will now be impertinent that players begin to investigate the benefits and drawbacks of certain limbs to survive. In particular, the Maple Seed's Wings may seem an obvious choice to clear gaps and walls. However, players will run into strong winds rendering flight useless at times. In turn, the Mushroom's Legs may provide better progression, but they come with their own set of drawbacks leaving the player to decide when to engage and disengage limbs. Again, after collecting the key here access to the next dungeon will be granted.



Dungeon 1.3

Upon completion of the previous dungeon the player will now be given more options in the limb usage for progression. The iterative style will carry over similar mechanics, however, they will be meshed with new interactions to allow for thoughtful combinations of limb usage. The strong winds previously mentioned will now be sporadic, providing the option of navigating between currents or perhaps shedding some weight for an alternate route. In correspondence with the previous dungeons, a key must be found here for access to the final dungeon.



Dungeon 1.4

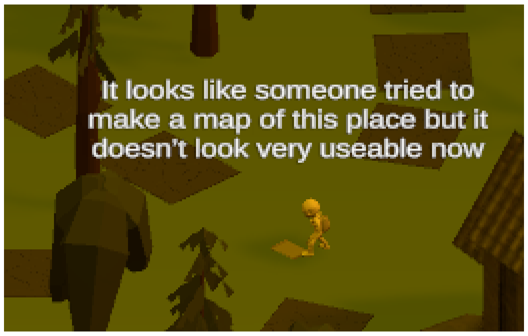
The final dungeon of this level will incorporate completely new enemy interaction and behavior. Similar movement mechanics will be present as in the three dungeons prior, however, the main idea is to demonstrate the possibility of additional enemies that players will have to investigate in an instant with no prior knowledge. Upon key collection and completion of this dungeon, access to the Upper World exit will be provided and progression to the Upper World, which is still in development and will be available in full production.

Fun

The enjoyment of the game stems from the ability to build your own creature; the joy of discovering how playstyles can differ from run to run based on the approach you want to take. Players will find themselves being more meticulous by exploring the level for potential new solutions and then acting on them, by putting more emphasis on problem-solving rather than combat.

Story

The story of Snatcher is one that is not told linearly; instead, it is told through descriptions of items found in the dungeons and overworlds of the game.


It looks like someone tried to make a map of this place but it doesn't look very useable now

Snatcher is the latest in a long line of creatures that shared that name and had the goal of conquering the dungeons in the area. The player is unaware of this fact when first loading the game, only knowing that Snatcher was drawn to this area at the very start. While the story is not an integral component of this game, players have the option of exploring the dungeons in order to piece the story together in order to gain a better understanding of the world and the characters in it.

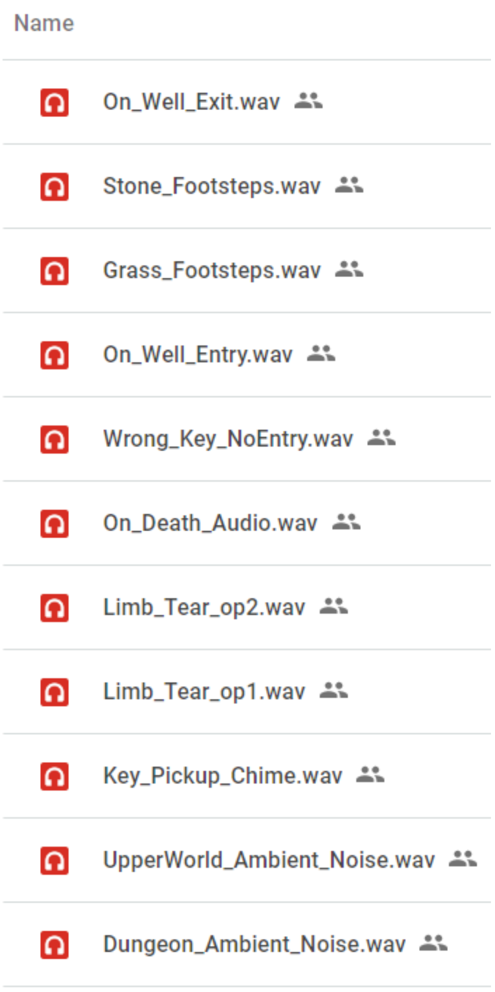
Audio

Volume

Currently the game only has a master volume slider. Future plan is to split existing and future audio into two categories of music and sound effects. The two categories will be controlled by separate sliders.

Sound effects

Footsteps and ambient noise are different depending on whether the Snatcher is in the overworld or in the dungeon. There are also on-event sound effects included, as shown in the list on the right.



Name
On_Well_Exit.wav
Stone_Footsteps.wav
Grass_Footsteps.wav
On_Well_Entry.wav
Wrong_Key_NoEntry.wav
On_Death_Audio.wav
Limb_Tear_op2.wav
Limb_Tear_op1.wav
Key_Pickup_Chime.wav
UpperWorld_Ambient_Noise.wav
Dungeon_Ambient_Noise.wav

Graphics

Perspective

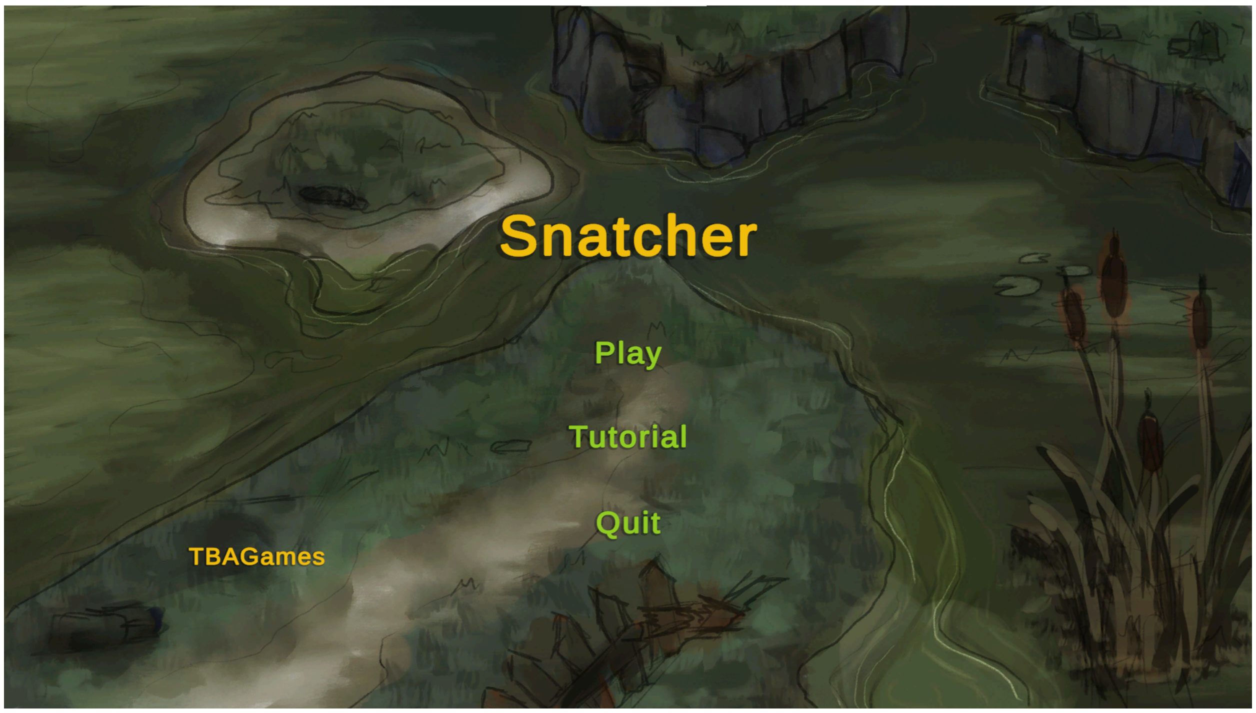
Snatcher is currently set to be a 2.5D, isometric game with a third-person POV. The camera will remain at a fixed angle, and stay still or follow the player, depending on the size of the room.

HUD

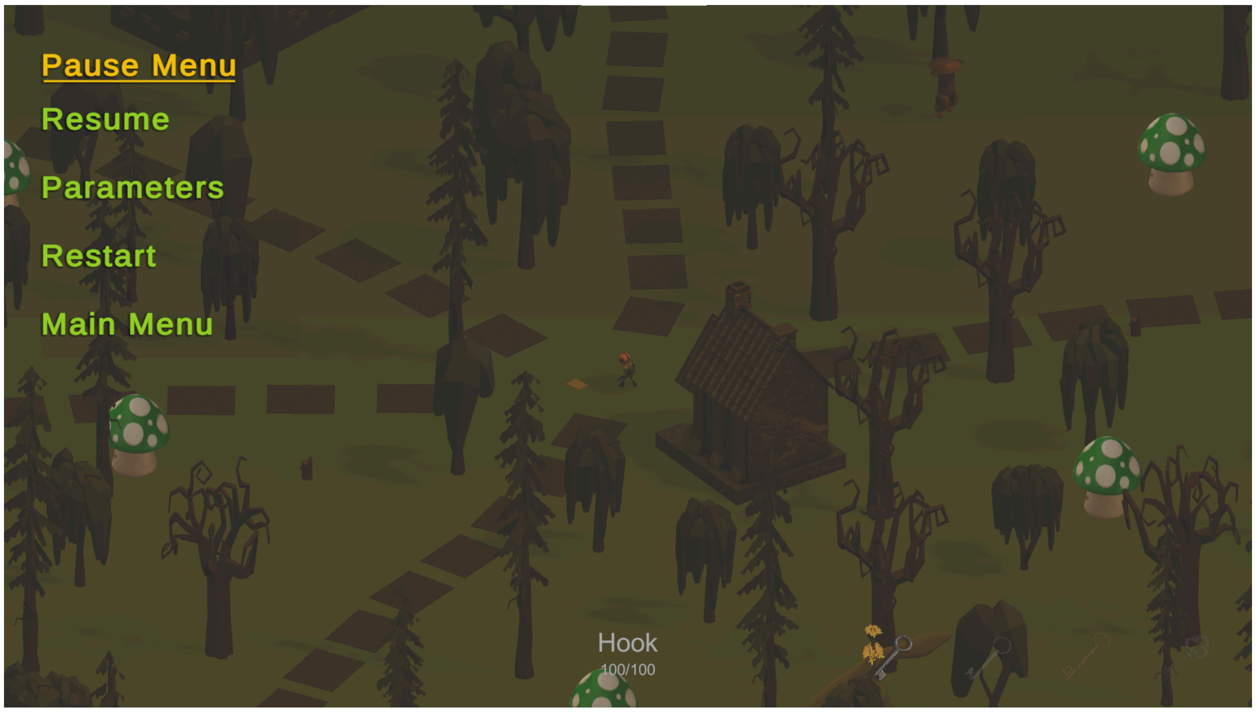
We will have a simple HUD to display the player's current key collection progress, limb inventory, and current stamina.
When the game expands to potentially include limb modification, currency, collectibles, and bestiary, this will also be accessible through the HUD.

Screen System

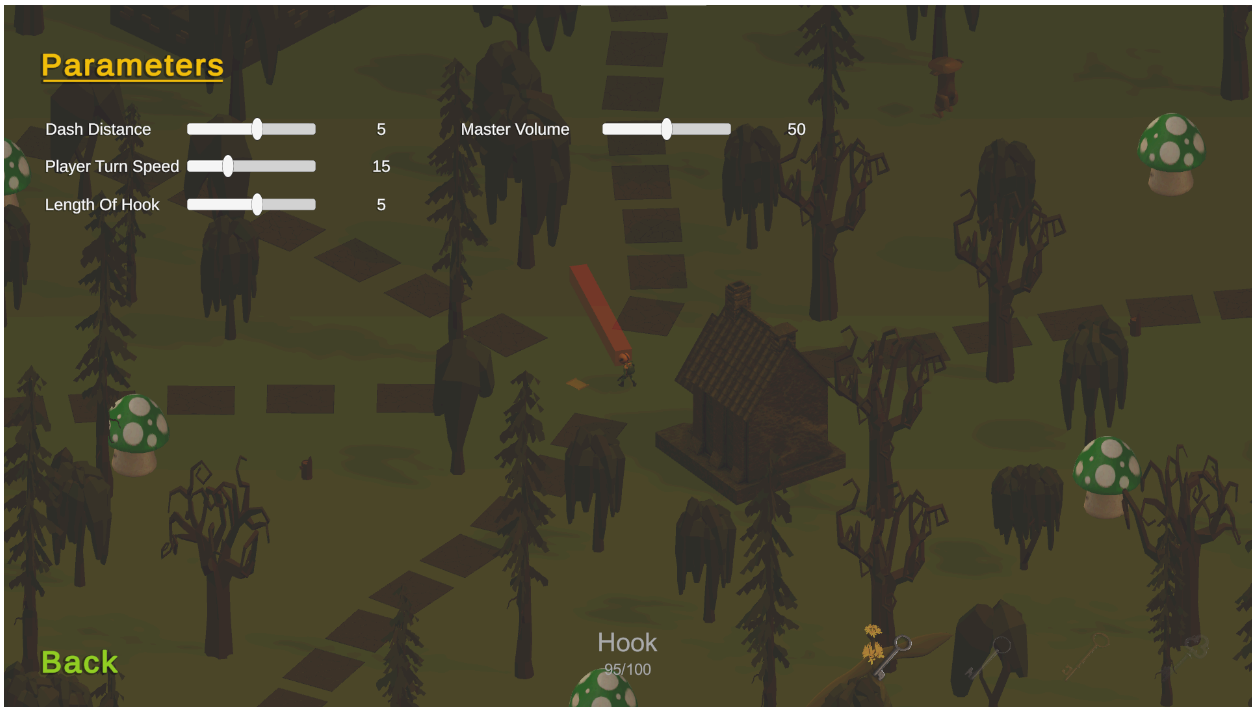
Start menu screen



Pause menu screen

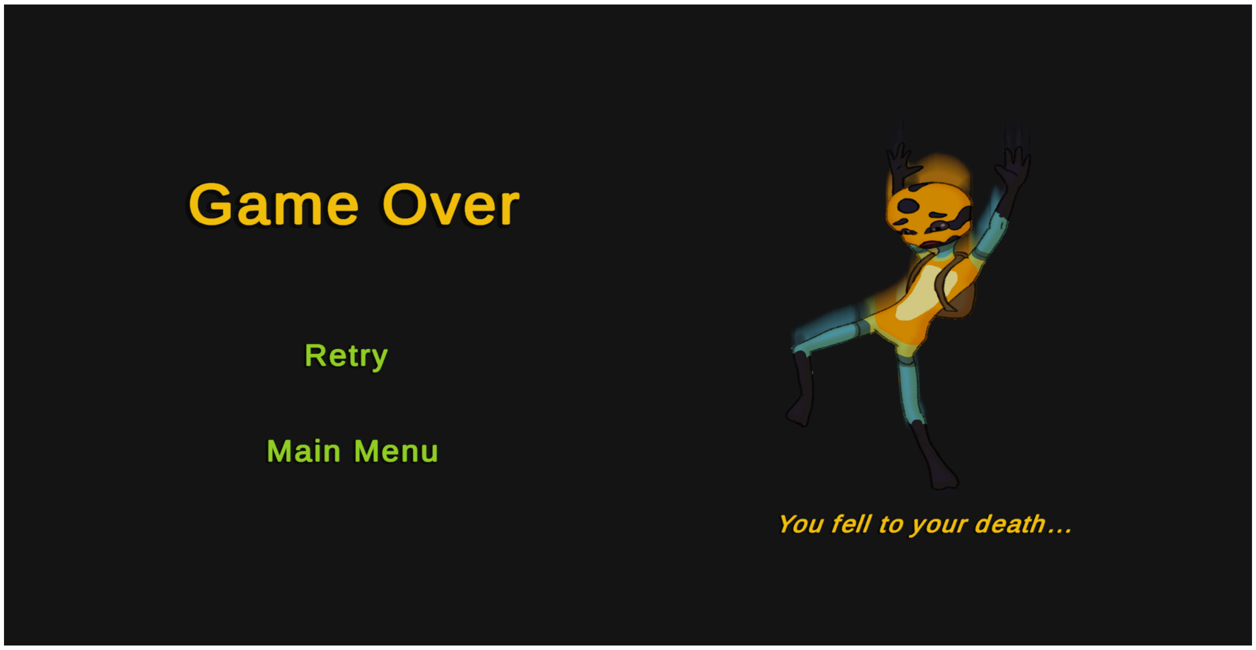


Parameter menu screen

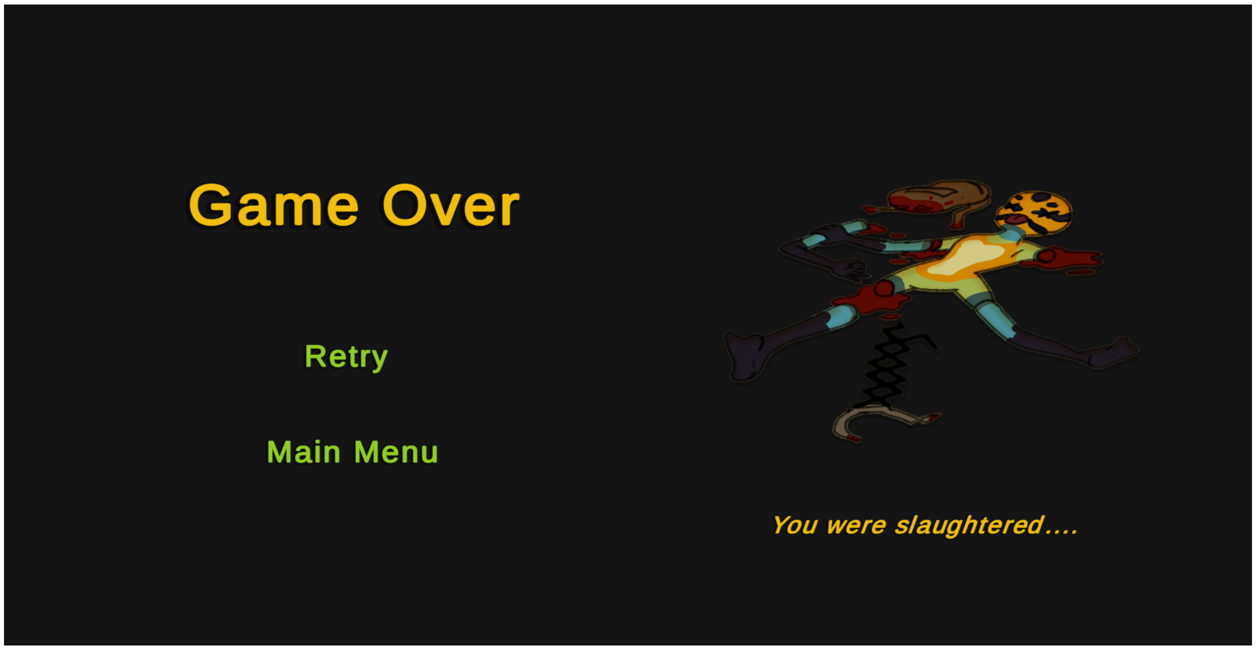


Dash Distance 5
Player Turn Speed 15
Length Of Hook 5
Master Volume 50

Death by falling screen

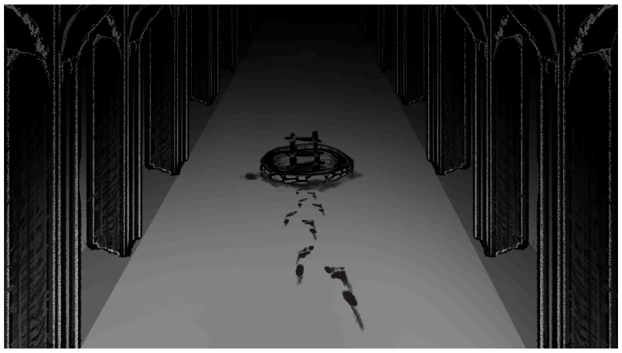


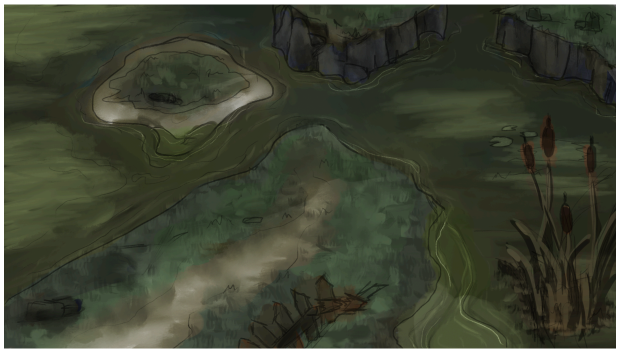
Death by enemy screen

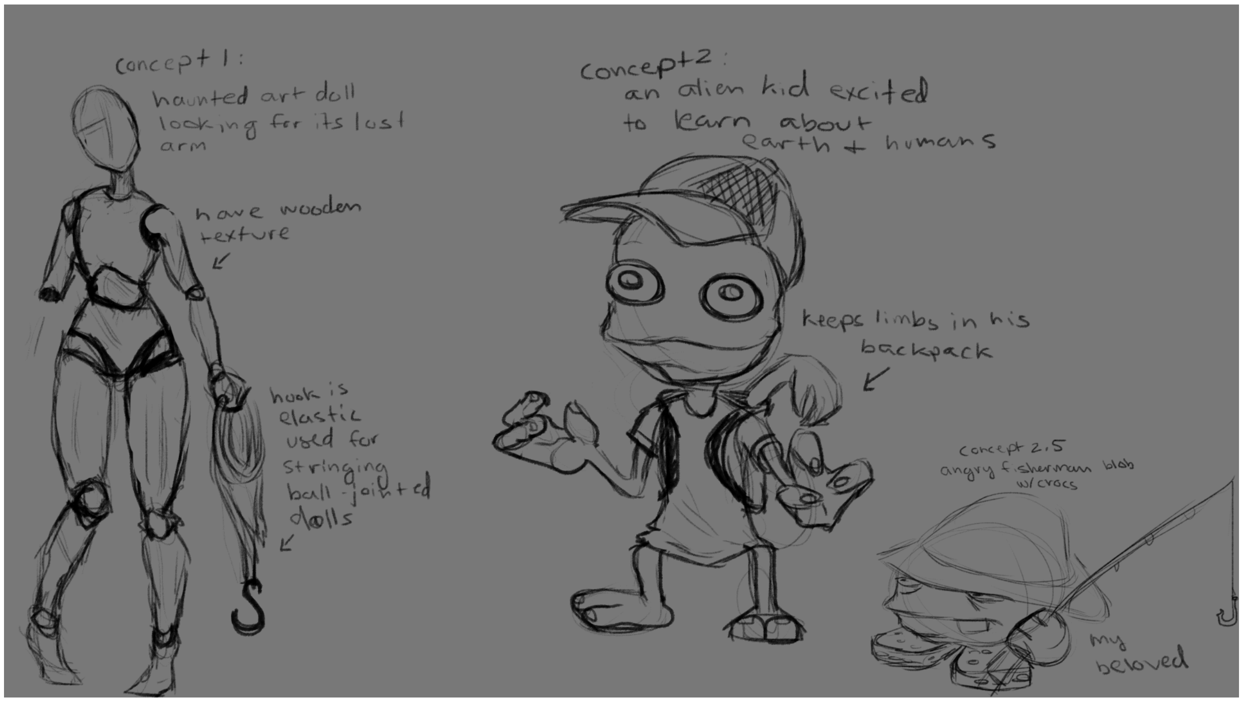


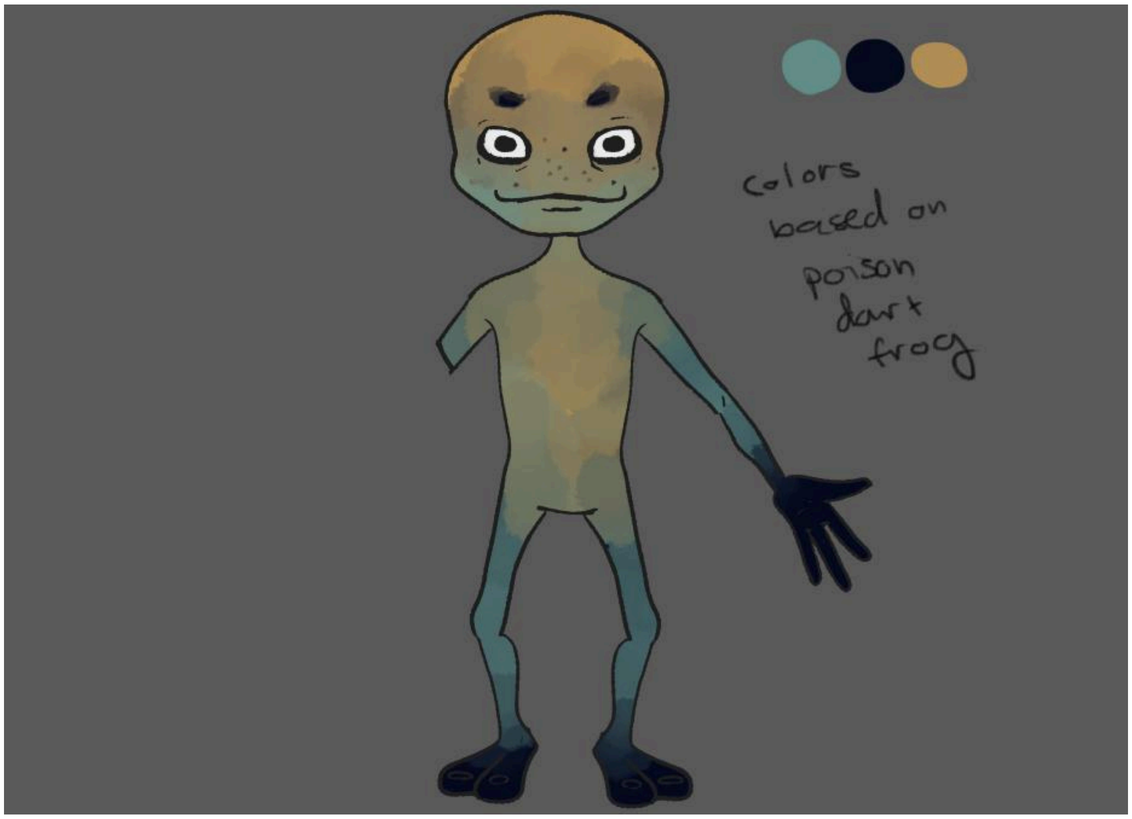
Concept Artwork

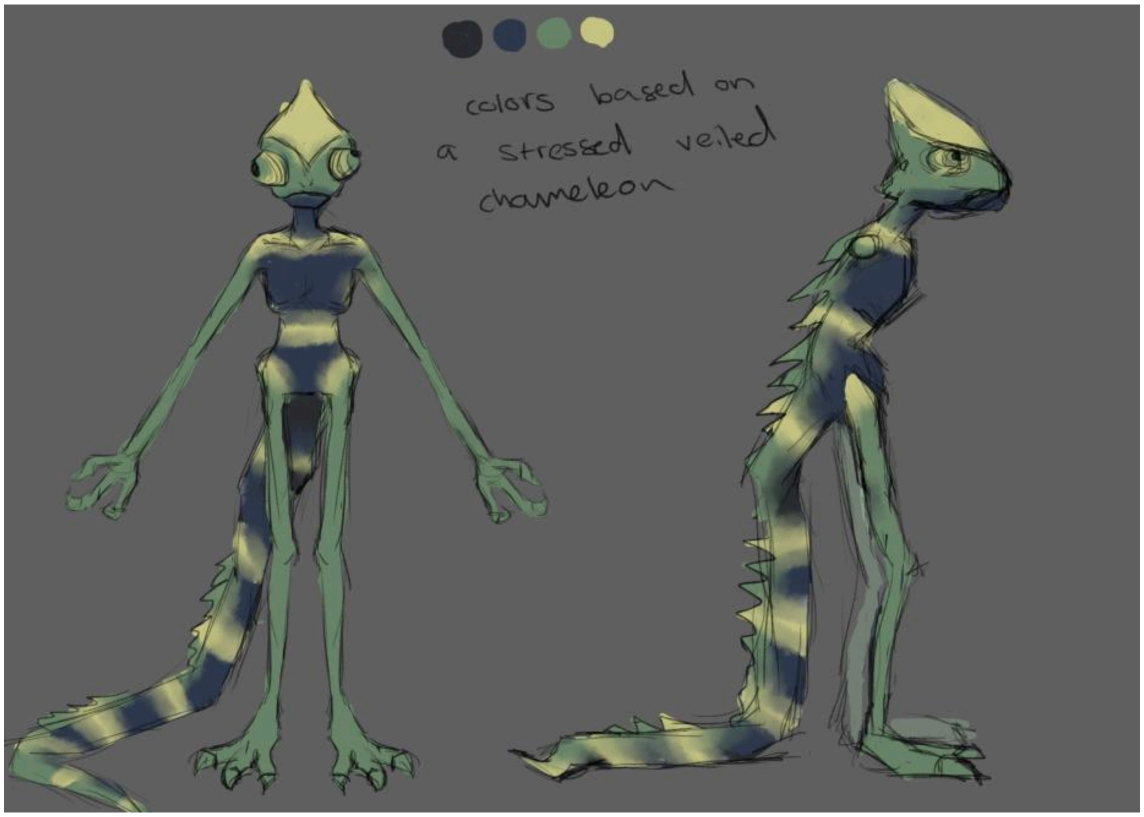
Snatcher’s art style is best described as “gothic horror”.



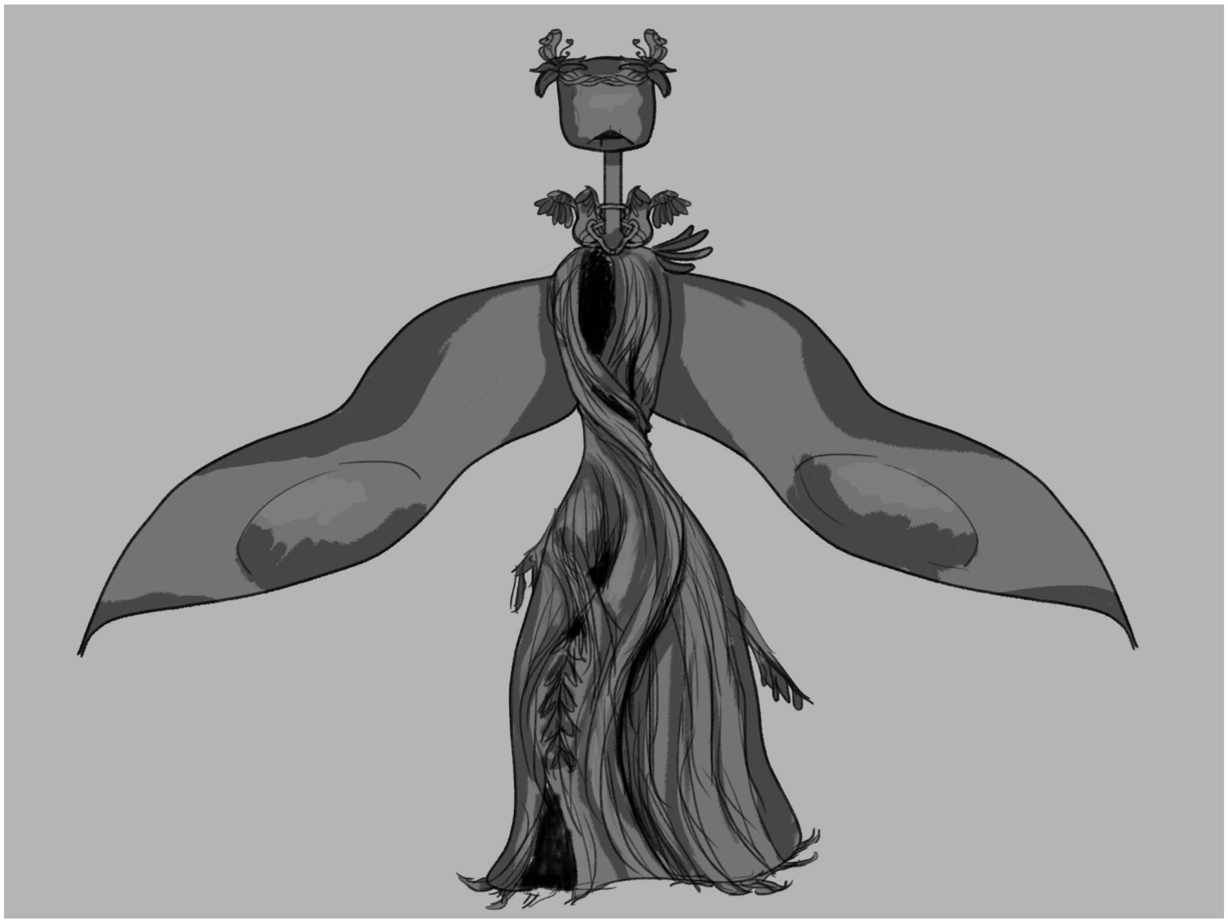



concept 1:
haunted art doll looking for its lost arm
have wooden texture
hook is elastic used for stringing ball-jointed dolls
concept 2:
an alien kid excited to learn about earth + humans
keeps limbs in his backpack
concept 2.5
angry fisherman blob w/crocs
my beloved


colors based on poison dart frog

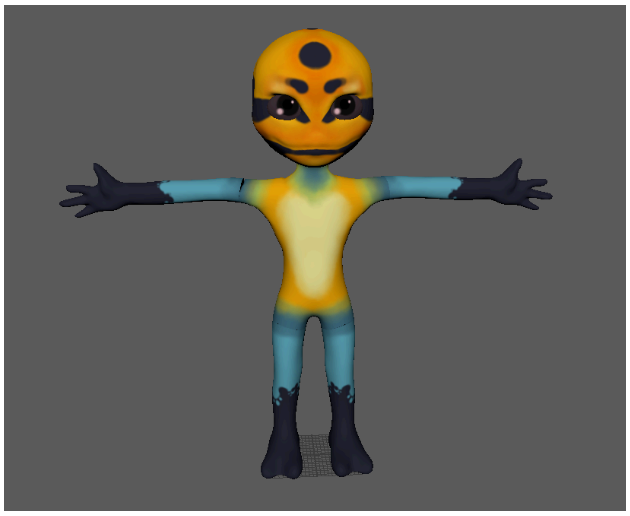

colors based on a stressed veiled chameleon

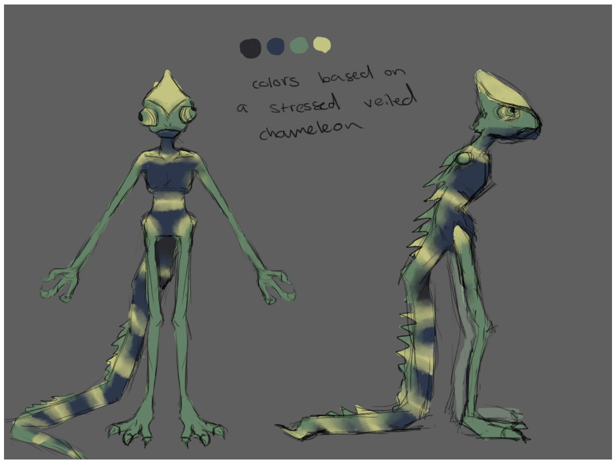
based on mushrooms

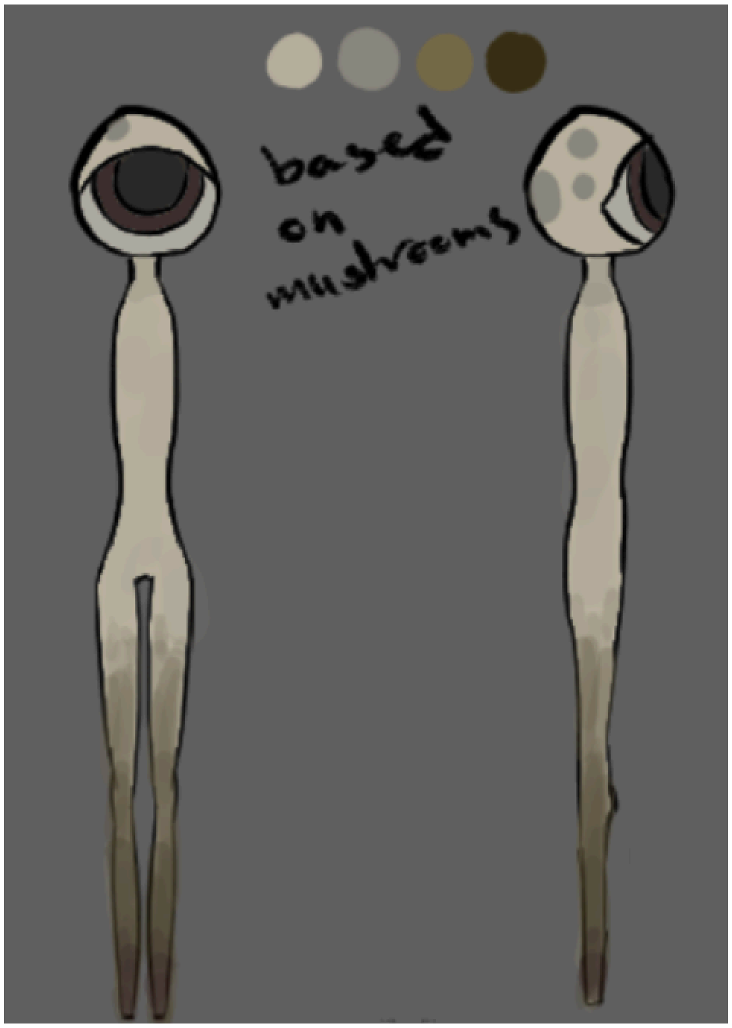


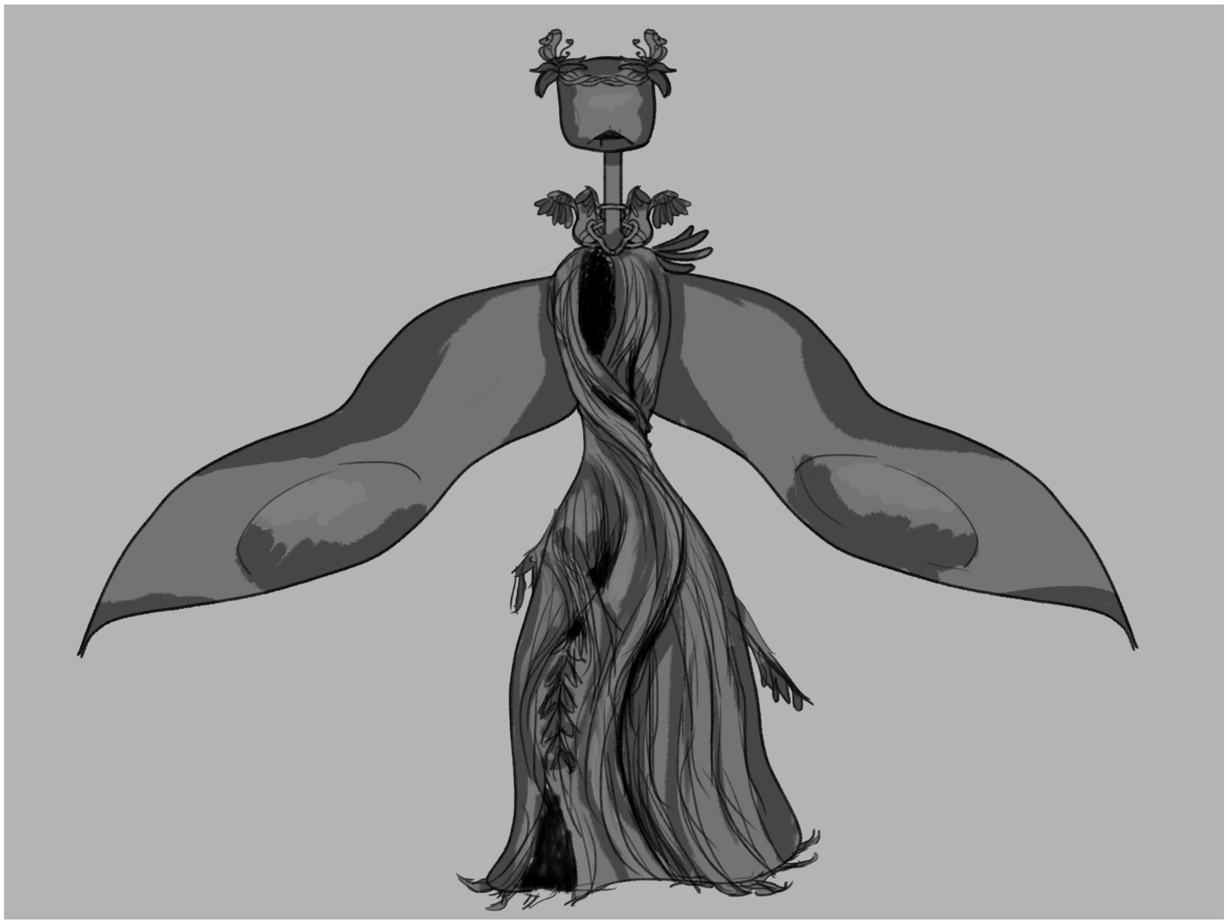
Game Assets

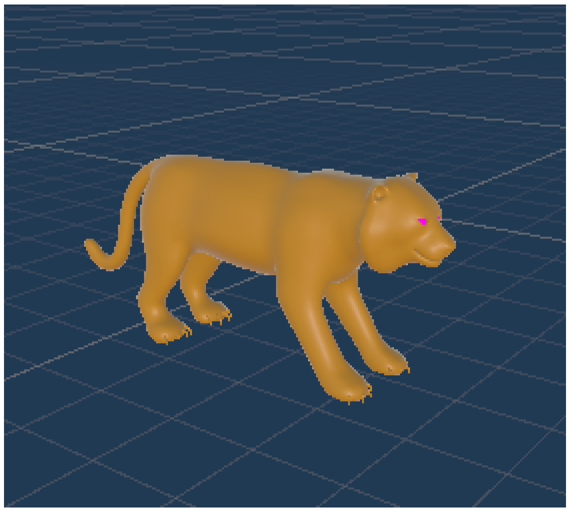
Primary Asset Template

Snatcher	
 colors based on poison dart frog	
Description/Backstory	Not much is known about Snatcher. But Snatcher knows what he wants. Limbs.
Height/Width	1 meter
Parameters	Limb inventory: this determines its ability Dash distance Rotation speed Hook length Max stamina Weight
Animation list	snatcher_dash snatcher_walk snatcher_fall snatcher_idle snatcher_hook snatcher_obtain snatcher_swap snatcher_fly snatcher_invis snatcher_jump snatcher_die

Stinky Chameleon	
	
Description/Backstory	When snatched by Snatcher, it will lose its tail.
Behavior	Standard
Snatchable limb	Tail
Height	1 meter
Parameters	Detection range: 10 Move speed: 3.5
Animation list	enemy_invis_walk enemy_invis_idle

Tall Mushroom	
	
Description/Backstory	When snatched by Snatcher, it will lose its legs. The ability its limb gives to Snatcher is a vault.
Behavior	Standard
Height/Width	3 meter
Parameters	Detection range: 10 Move speed: 3.5
Animation list	enemy_vault_walk enemy_vault_idle

Flying Maple Seed	
	
Description/Backstory	When snatched by Snatcher, it will lose its wings. The ability its limb gives to Snatcher is flight. This enemy does not currently fly when idle or chasing Snatcher.
Behavior	Standard
Height/Width	2 meter
Parameters	Detection range: 10 Move speed: 3.5
Animation list	enemy_fly_walk enemy_fly_idle

Scent-Tracking Tiger	
	
Description/Backstory	There is no limb to be snatched currently. This enemy wanders around on the ground.
Behavior	Scent Tracking
Height/Width	1 meter
Parameters	Detection range: 10 Move speed: 3.5
Animation list	enemy_fly_walk enemy_fly_idle

Asset List

The asset sheet and task assignment can be found here.

Asset Repository

The asset repository can be found here.

Dev Documentations

Coding Standards

General Structure of a File

- Since everything in C# must be in a class, it makes sense to separate the implementation of a class into its own file.
- Remove any unnecessary "using" statement
- After the name of the class, follow the order listed below
    - Properties and fields
        - Constants (public ones should go above private ones)
        - Static (public ones should go above private ones)
        - Public properties and events
            - Events here mean C# event not SO Events
            - SO Events should be treated like [SerializeField] private fields
        - [SerializedField] private fields
        - Private fields
    - Methods
        - Public methods
        - Unity-related methods
            - Start, Update, FixedUpdate, etc.
        - Private methods

Open Curly Bracket

- **Please put it on the next line**

Classes

- Use PascalCase
- Names should be **nouns**
    - "Enemy"

Abstract Class

- An abstract class should be prefixed with the letter "A"
    - "APlayerState"

Public Properties

- Use PascalCase
- Names should be **nouns** or **adjectives**
- Try to use public properties instead of public fields

    - If you want to expose a field in the inspector, use [SerializeField] on a private field.
    - There is an exception for classes that are very **self contained** and **simple**, or if it is a ScriptableObject. For sake of simplicity, use it. However, it should still use PascalCase.
- “Direction” “IsMoving”

Protected Properties

- Use PascalCase
- Names should be **nouns** or **adjectives**
- Try not to use protected fields
    - When using properties or fields that are protected, we are essentially accessing/modifying something from another class. For this reason, we use PascalCase.

Private Fields

- Use _camelCase
- Prefixed with underscore (_)
- Names should be **nouns** or **adjectives**

Methods

- Use PascalCase
- Names should be **verbs**

Method Parameters

- Use camelCase
- Use **nouns** and **adjectives**
    - “movementVelocity”

Local Variables

- Use camelCase
- Use **nouns** and **adjectives**
    - “currentIndex”

Constants

- ALL_CAPS and underscore between words
    - “PI” or “RESOURCE_PATH”
- Use **nouns** and **adjectives**

- Keep the declaration of a constant at the top of the file. If they are constants, that means they are important.

Events

- Use PascalCase
- Prefix with “On”
    - For both the SO events themselves and the subscribers’ callbacks
        - Use **past participle verbs**
    - For the subscribers’ callbacks, you can opt to use simple verbs, too. Just like a call to action (method).

Access Modifiers

- **Always add access modifiers for consistency**
- In C#, if a method/field/class does not have a specified access modifier, it is implicitly marked as private.
- To maintain consistency, always add “private” in front of things that you want to be private
- This means that the Start and Update methods in the provided template of a MonoBehavior needs to be corrected as well. By default, there are no “private” access modifiers in front of them.

Others

- Try not to abbreviate too much
    - Unless it’s something everyone knows, such as *CPU* or *i* and *j* for a for-loop

Player Finite State Machine

Introduction

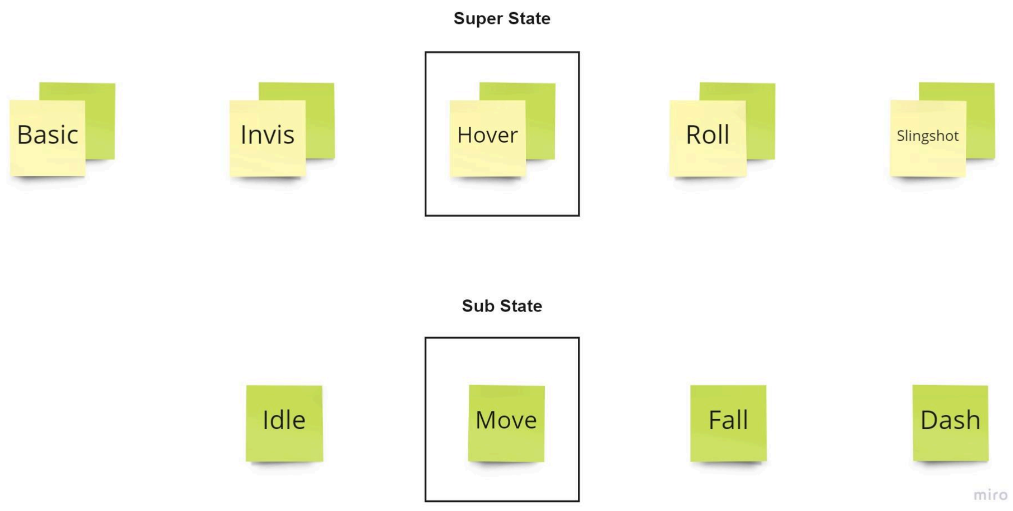


Figure 1. Abstract structure of Snatcher Hierarchical FSM. The Super States and the Sub States are independent of each other.

The implementation of Finite State Machine (FSM) in this project is a hierarchical FSM. Much like a regular state machine, we segment the specific behaviors we envision for a state into its own class. Doing so allows us to reduce conditional checks and thus chances of errors. A swim state only cares about aspects related to swimming, and, when doing its own business, it does not care to check if it is also in the air.

The difference between a regular FSM and our hierarchical FSM is that our implementation requires the state machine to be in two states all the time, one being a Super State and the other a Sub State. Mix and match, and we get multiple combinations of *overall states* the state machine can be in, as shown in Figure 1.

The Super State is directly related to the Limb the player is currently equipped with. It functions as a *configuration consultant* for the Sub States where it provides stats affected by the currently held Limb, such as movement speed, dash distance, and Limb Ability.

The Sub States are the states that actually manifest the behaviors of a state machine. Moving the CharacterController, setting a boolean value in the Animator, and reading player input, are done by the Sub States. There are four universal Sub States that are always present regardless of the currently equipped Limb: Idle, Move, Fall, and Dash. Idle State is a special state which we will discuss later.

There are three major parts to the system: the context, the abstract state, and the concrete states. The context is the Player State Machine, which is a MonoBehaviour. The abstract states are the templates for concrete states. ASuperState is the template for Super States, ASubState is for Sub States, and APlayerState for both ASuperState and ASubState. The concrete states are the concrete implementation of ASuperState or ASubState. To summarize, APlayerState is the grandparent class, ASuperState and ASubState are the parent classes, and all concrete classes are child classes.

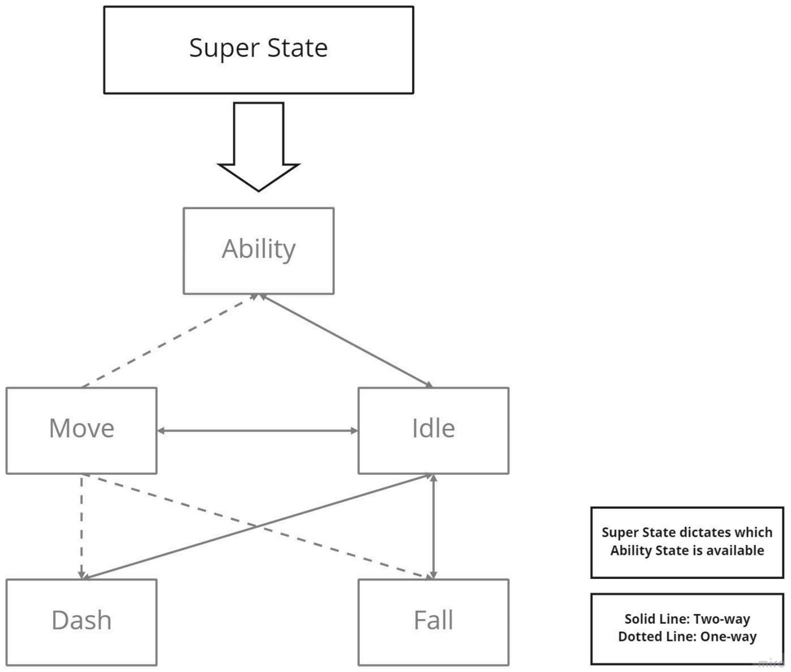


Figure 2. Flowchart of the four universal Sub States and the ability state(s).

The flowchart above demonstrates the relations between the four universal Sub States and the Ability State determined by the current Super State. A solid line means a two-way transition, and a dotted line means a one-way transition. The Super State and the Ability State will be explained later.

Now, take a look at Fall State and we can see that only two states can transition into it: Move and Idle. Such transitions happen when the context becomes *NOT* grounded. We can also see that there is only one state that Fall State can transition into: Idle. The transition happens when the context becomes grounded again. The last observation we can make on Fall is that it has nothing to do with Dash. It is a good example of Fall State only caring about what needs to be cared about for itself.

Finally, the reason why we are able to delegate distinct concrete implementations to each individual state, and still be able to wire them together using our FSM, is that each state (whether a Super State or a Sub State) is a child of the common abstract state *APlayerState* from which they inherit shared common methods and properties. On that note, every Sub State is always able to get relevant information from the current Super State because all Super States inherit from *ASuperState* which guarantees that all Super States meet certain requirements. We can say the system relies heavily on polymorphism to carry out its function.

Player State Machine

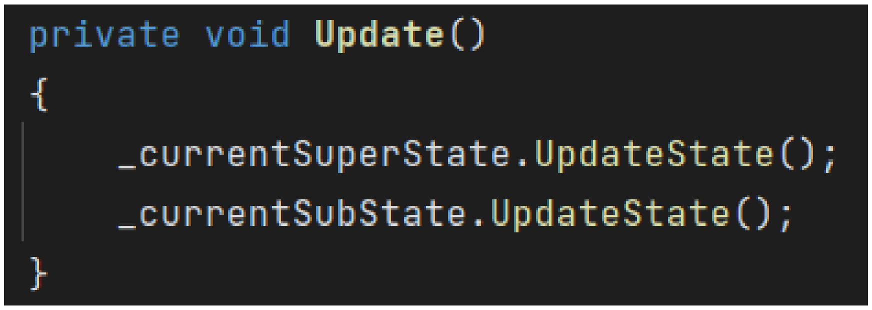
Terminology

The first task to tackle is to clarify some terminologies. In this Guideline, the term *Player State Machine* is interchangeable with the term *context*. They both refer to the MonoBehaviour instance that is attached to the GameObject that we want to act as the player. For consistency throughout the Guideline, and for cohesion with the actual code implementation in the Unity project, the term *context* will be used, unless it becomes necessary to use the term *Player State Machine* for sake of clarity.

Update Method and Passing Context

The APlayerState is a pure abstract C# class. This means every state in this system is not a MonoBehaviour. If a class does not derive from MonoBehaviour, it does not have methods like *Start* or *Update*. This makes an operation in a state that requires an update every frame a tricky task. To cope with that, every state has the public method *UpdateState*, which is a result of deriving from APlayerState. We can call this method from the context in the Update Method, like shown in Figure 3.. Remember, the context is a MonoBehaviour.

The communication between the states and the context are two-way, meaning that the context knows the current state it is in, and the state also knows who is operating on it. It is easy for the context to know its current state. However, it is not so obvious from the state's standpoint which context is acting on it. Some models of FSM pass the context as an argument when calling a method of a state. Our implementation, shown in Figure 3, uses a trick where it does not require passing the context as an argument and still makes the context known to its states. It will be explained in the Factory Pattern section.

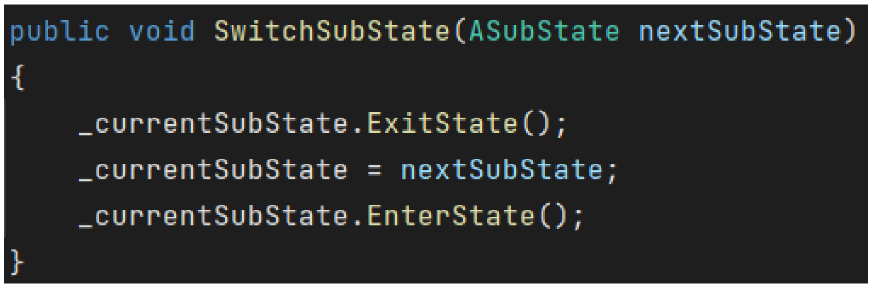


```
private void Update()
{
    _currentSuperState.UpdateState();
    _currentSubState.UpdateState();
}
```

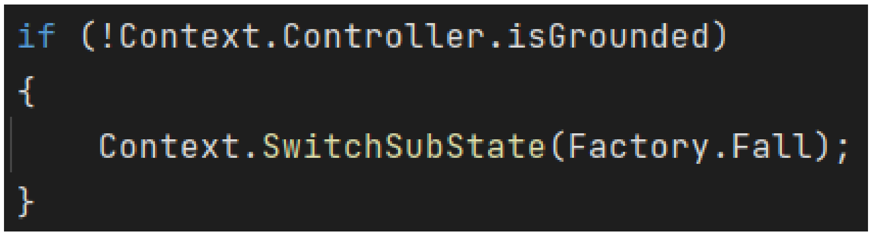
Figure 3. The *Update* method in the context. When calling the *UpdateState* method, we do not pass in the context as an argument.

Switching States

The ability to transition into a different state is a requirement for a FSM. Most models of FSM, ours included, let the states decide when to transition into another state. The states themselves don't have the mechanism to actually make transition happen. However, the context does. Therefore, when a state decides to transition into another state, it calls a method in the context to do so while also specifying which state to transition into next.



```
public void SwitchSubState(ASubState nextSubState)
{
    _currentSubState.ExitState();
    _currentSubState = nextSubState;
    _currentSubState.EnterState();
}
```



```
if (!Context.Controller.isGrounded)
{
    Context.SwitchSubState(Factory.Fall);
}
```

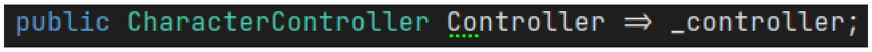
Figure 4. Above is the *SwitchSubState* method in the context. Below is an example of a state calling the *SwitchSubState* method from the context. There is also a similar method called *SwitchSuperState* which handles switching to a different Super State.

The lower image in Figure 4 shows a simplified code snippet from the Idle State. It demonstrates that when the context is not grounded, it will transition to Fall State. The identifiers *Context, Controller,* and *Factory* will be explained in the next sections. However, there is one thing that is important here. This code exists in a state, and it needs to know the context in order for this operation to work. It shows the importance of making the context known to the states, and, in this case, it does.

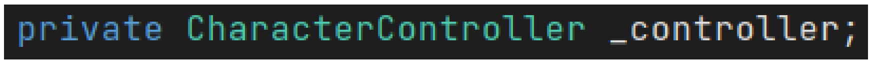
Public Properties & Private Fields

In the lower image in Figure 4, we can see that, within the state, we are calling *Context.Controller*, which is a public property of the context. In this case, it is a reference to Unity's Built-in Character Controller component. This kind of operation is frequently used in this system. If the state needs to do something with the Animator, it calls *Context.Animator*. Of course, these have to be public properties that have already been set up in the context.

The preferred way of setting up this kind of property is through the use of public get-only properties. Properties in C# are like a syntactic sugar of getter and setter methods. By making it get-only, we prevent others from accidentally changing the value of a field or the instance of a class. The image below shows the general idea of how this is set up.



```
public CharacterController Controller ⇒ _controller;
```



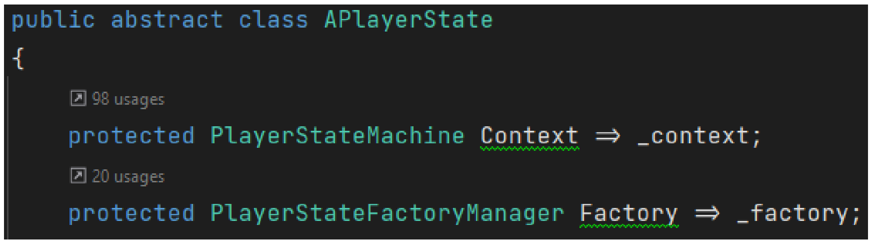
```
private CharacterController _controller;
```

Figure 5. Both images are code snippets from the context. The above shows when getting *Controller*, it will return the instance of *_controller*. The *_controller* in this case is something we will cache a reference in the *Awake* method using *GetComponent<CharacterController>()*.

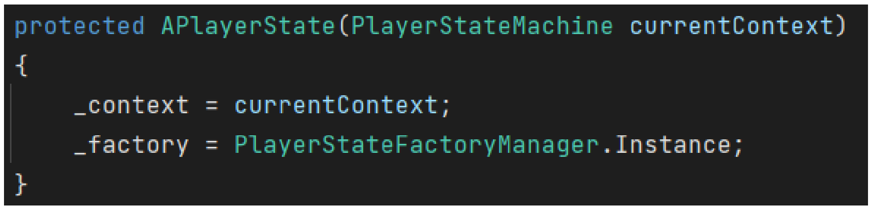
Here is a list of public properties (and their types) currently offered in the context. Note that this list is subject to change as the project expands.

- Debug (bool)
- PlayerInput (PlayerControls; an input-related script)
- Controller (CharacterController)
- Animator (Animator)
- HookController (HookController; custom script for hook behavior)
- GroundCheck (Transform)
- LimbSlot (Transform)
- CurrentSuperState (ASuperState)
- CurrentSubState (ASubState)
- CanGrapple (bool)
- GrappleDestination (Vector3; it is also a setter)

Factory Pattern



```
public abstract class APlayerState
{
    98 usages
    protected PlayerStateMachine Context => _context;
    20 usages
    protected PlayerStateFactoryManager Factory => _factory;
```



```
protected APlayerState(PlayerStateMachine currentContext)
{
    _context = currentContext;
    _factory = PlayerStateFactoryManager.Instance;
}
```

Figure 6. APlayerState has a public getter to the context and another public getter to a factory instance. Both are initialized in the constructor of APlayerState.

Naturally, a state should not be hard-wired to other states. However, it is up to the state to call for switching to other states, and specifying which state to switch to is crucial. Therefore, we need a centralized place, accessible from all states, to manage all the available states and grant access to the states it manages to other states. This is the job of the factory. Quick disclaimer, this is not a typical implementation of the factory pattern. However, the difference here is trivial.

As shown in Figure 6, since every state has a reference to the factory containing all the states, a state can then specify another state and transition to it by calling *Factory.Fall*, for example, like in Figure 4. The main takeaway here is that we don't need to and don't want to hard reference a concrete state to another.

The implementation of the factory can be found in PlayerStateFactoryManager. It is a manager class, and how it is initialized as well as how it initializes states will be addressed in the PlayerStateFactoryManager section.

Super State

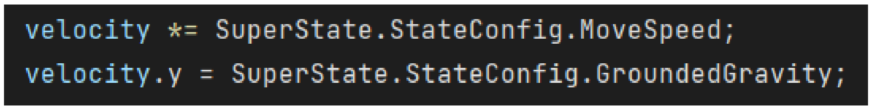
Configuration Consultant

Super States are pretty useless if we consider that they don’t actually define any behaviors, and they tend to have just fewer lines in their files for that reason. However, they are important because the Sub States, the handling of Mecanim, and switching of Limbs rely on the Super States to do the job.

In Figure 1, we can see that the Super States and Sub States are independent of each other. It is as though being in a Move Sub State has nothing to do with being in a Hover Super State. With the way we handle it, if there are five Super States and five Sub States, we can come up with 25 unique combinations of the *overall states* for the context. This does not even take into account some additional Sub States that are specifically related to Limb Abilities, which will be discussed in the Ability Entry State section.

There is a problem with this approach. The Move Sub State will have the same behavior regardless of the Super State the context is in. But what if we want Snatcher to move more slowly when it is equipped with a large arm, making Snatcher in, say, a LargeArm Super State?

The way we handle this is to let the Sub State be aware of the current Super State the context is in. Additionally, we want the current Super State to provide appropriate configurations for the movement speed, turn speed, and dash distance, etc. Stringing the two requirements together, we get something like this:



```
velocity *= SuperState.StateConfig.MoveSpeed;
velocity.y = SuperState.StateConfig.GroundedGravity;
```

Figure 7. Code Snippet from the Move Sub State. We can access the Super State by calling *SuperState*. We can also ask for configuration using the dot notation and specify *StateConfig*.

In Figure 7, the *StateConfig* is a property that has a reference to a ScriptableObject that contains a number of configurations/parameters for a specific Super State. When working in a Sub State and wanting to access a certain configuration, you do not need to worry about which concrete Super State the context currently is in. You simply need to know that the current Super State can be accessed this way, and the configuration in question is queried and provided automatically by the system.

If you’re interested in how this is done, just hit F12 whenever you want to see the implementation of something, and go through the code.

Ability Entry State

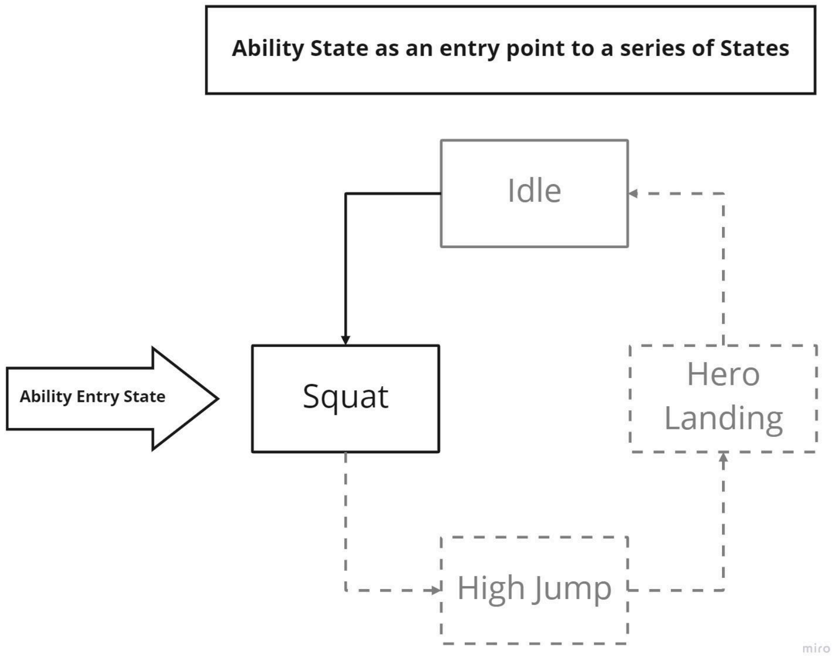
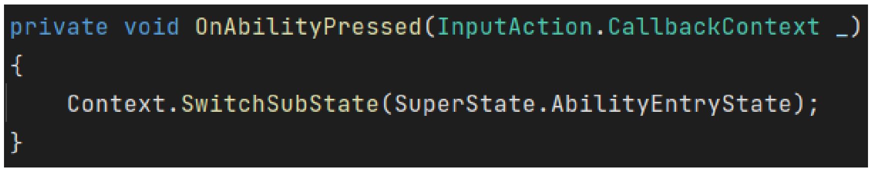


Figure 8. Diagram of entering and exiting a series of Ability States. The structure is similar to a singly linked list where we only care about the head node (the Ability Entry State, Squat State). The rest of the nodes (other Ability States, High Jump State and Hero Landing State) will handle themselves and eventually transition back to Idle State.

Every Super State has a property that holds a reference to a special state, the Ability Entry State. Under the hood, the Ability Entry State is just a Sub State that handles some behaviors for the context. It is called the Ability Entry State because whenever we want to activate the Limb Ability, we enter this state.

There are currently two Sub States in which we can transition into an Ability Entry State: Idle State and Move State.

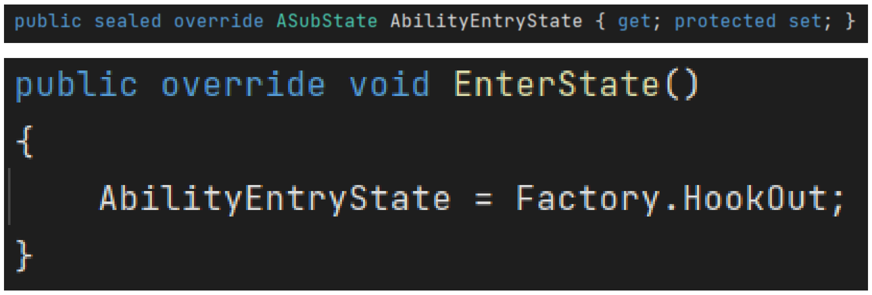


```
private void OnAbilityPressed(InputAction.CallbackContext _)
{
    Context.SwitchSubState(SuperState.AbilityEntryState);
}
```

Figure 9. Code snippet from Move State. *OnAbilityPressed* is a callback to a button press event, which is invoked when the ability button is pressed.

Pay attention to the expression in the middle, *SuperState.AbilityEntryState*. Once again, we delegate the task of finding the appropriate state to the current Super State. The Sub State simply needs to acknowledge that its current Super State knows what state is the right state to transition into.

Inside the Super State, there is something like this:



```
public sealed override ASubState AbilityEntryState { get; protected set; }
```

```
public override void EnterState()
{
    AbilityEntryState = Factory.HookOut;
}
```

Figure 10. Code snippet from a Super State.

We can see that the Super State has a property of type ASubState. In this case, we specify the reference to be the Hook Out State, which is the Ability Entry State for the Basic Limb.

The reason we cache the reference to the Ability Entry State inside the EnterState method instead of doing so in the constructor, is a little complicated. To simply put, if we cache in the constructor by calling *Factory*, we will recursively call the constructor, which will never end and will raise a StackOverflow Exception. Caching inside the *EnterState* method solves that problem. The current implementation is not the most optimized. However, we also don't switch Super States that often.

Now, switching to an Ability Entry State makes sense. However, going back to normal, or in our case, transitioning back to Idle State, is not so obvious. It is actually the responsibility of the person who designs the Limb Ability States to figure out how to transition back to Idle State. In the case of Hook Out State, it simply transitions back to Idle State when nothing is hit. Or, it transitions to Grapple Toward State when a grapple pillar is hit. Grapple Toward State automatically transitions back to Idle State when the character is close enough to the grapple pillar. The implementation of transitioning into a different Super State when hitting an enemy has not been done yet.

The entire process can be thought of as a singly linked list like shown in Figure 8. The idea is that when a Limb Ability is activated, we first transition into the corresponding Ability Entry State. Then, a cascade of Ability States happens as one transitions into another. The process will eventually end up transitioning back to Idle State. It is important that we terminate the process in the Idle State, not Move State or other Sub States. It has something to do with Mecanim and setting parameters. It is also important that if there is branching, like in the case of Hook Out State, all branches should lead to Idle State as well.

Sub State

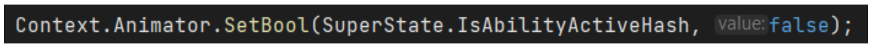
Special Sub State: The Idle State

Idle State handles the behavior for when the character is idling. In a sense, it does not do much except for playing the idle animation and waiting from some input.

Idle State is a very special Sub State because it is sort of like a *default* Sub State. Every other Sub State (except for the in between Ability States) can transition into Idle State. In addition, Idle State can transition to every other Sub State (again, except for the in between Ability States). This makes Idle State an ideal *go-to* state because it will handle the logistics of transitioning to other states with most efficiency.

For example, the current Sub State is Fall State. After falling, we don't need to worry about whether the player is holding down directional input thus switching to Idle State or Move State accordingly. We simply transition to Idle State because if the player is holding down directional input, the Idle State will by itself transition into Move State.

Another reason why we want to always transition back to Idle State is that it resets the animation parameter that controls the Ability animations.



```
Context.Animator.SetBool(SuperState.IsAbilityActiveHash, value: false);
```

Figure 11. Code snippet from the *EnterState* method in Idle State.

Like shown in Figure 11, we call the *SetBool* method with the Animator, and we specify the Animator should exit animations that are related to a Limb Ability. Technically, we can call this method in Move State, too, and don't always have to transition to Idle State. However, we don't want to repeat ourselves, so Idle State naturally becomes the candidate for calling this method. (The hash will be explained in the Parameter Hashing section.)

Move

Move State is the Sub State that handles the movement behavior of Snatcher. There isn't much to look at since the implementation is stable at this point. However, there are still a few points worth mentioning.

Going back to Figure 2, we can see that *Move State* can only be accessed from *Idle State*. This only occurs when the FSM receives a directional input. When entering *Move State*, two things happen.

First, we register the FSM to three input events, each of which, once invoked, will cause the FSM to transition to another state.

- *OnMovementCanceled* event, which happens when the directional input is terminated and the FSM will transition back to *Idle State*.
- *OnDashPressed* event, which happens when the dash button is pressed and the FSM will transition to *Dash State*.
- *OnAbilityPressed* event, which happens when the ability button is pressed and the FSM will transition to the Ability Entry State of the current Super State.

The second thing that will happen upon entering *Move State*, is that the FSM will set the bool parameter "IsMoving" in the Animator to true.

The opposite operations will be executed upon exiting *Move State*. The FSM will unregister from the three input events, and will also set "IsMoving" to false.

In the *UpdateState* method, four things happen, and they execute in this specific order: *UpdateDirection* > *UpdateRotation* > *UpdateMovement* > *CheckSwitchState*. The details are irrelevant in this Guideline, but two methods need special attention to some extent.

The first is the *UpdateMovement* method. Its only job is to combine horizontal movement with vertical movement (gravity), and apply this overall movement to the character by calling the

CharacterController's *Move* method. This step of combining movements is necessary as calling the *Move* method more than one time within one frame can create artifacts. Therefore, we have to account for both types of movement, aggregate their values, and only apply it once per frame, which is exactly what *UpdateMovement* does.

The second is *CheckSwitchState*. It is also called every frame. In this case, it will check if the player is still grounded. If not, the FSM will transition into *Fall State*. There is currently no way to check such an occurrence using an event. Thus, we have to check it every frame.

Fall

Fall State is very similar to *Move State*. In *Fall State*, the character can move around and rotate. It also sets an Animator Parameter to play the fall animation. However, there are two differences between *Fall State* and *Move State*. First, *Fall State* does not observe any input event. It only transitions to *Idle State* when the character becomes grounded. Second, the gravity applied to the character is different than that when it is grounded. This gravity constant is usually larger, but we can configure it otherwise.

One thing to mention is the *isGrounded* property of the CharacterController component. It will only detect correctly if in the last frame update, the *Move* method was called. Therefore, both in the *Move State* and *Fall State*, the *Move* method is always used to move the character. It is never through the means of directly moving Transform.

Dash

Dash State, once entered, will only exit to *Idle State* when the dash has been fully performed. In the future, we will want to stop the dash when hitting something. The implementation is rather simple. However, it also uses DOTween and an async method, both of which can seem like dark magic.

Unity Mecanim

Mirroring The FSM

The Mecanim side of the Player FSM should mirror the structure of the scripting side. It should do so in two senses. One is the Super States being their own small state machines. We will make Sub State Machines for the Super States in Mecanim to reflect that. The other is that the transitioning between Sub States in Mecanim should reflect the actual transitions in the scripts. If *Fall State* cannot transition into *Dash State* in the scripts, it should not have an animation transition in the Mecanim, either.

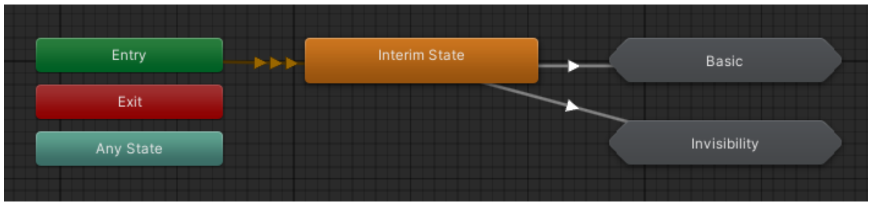


Figure 12. The Super State layer of Mecanim. Interim State is a state to mediate the entry into one Sub State Machine.

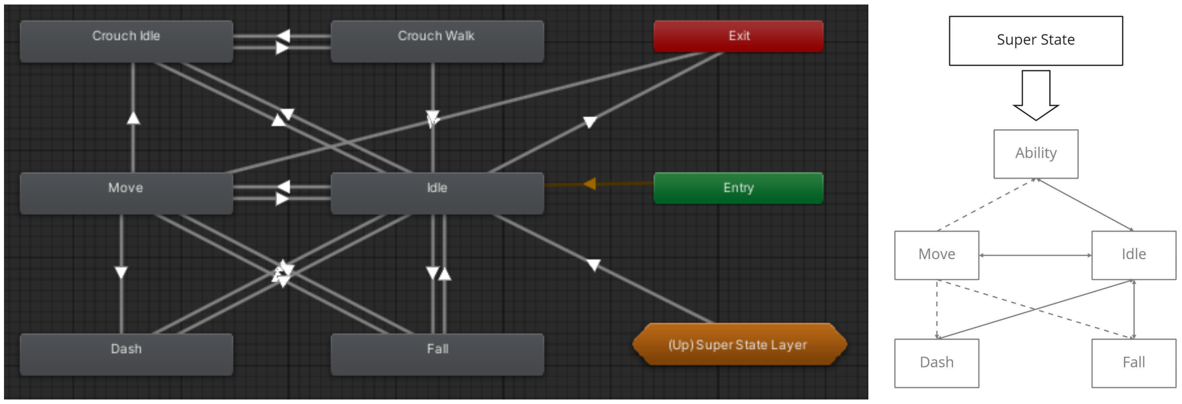


Figure 13. Inside the Invis Sub State Machine. The structure is very similar to the structure shown in Figure 2.

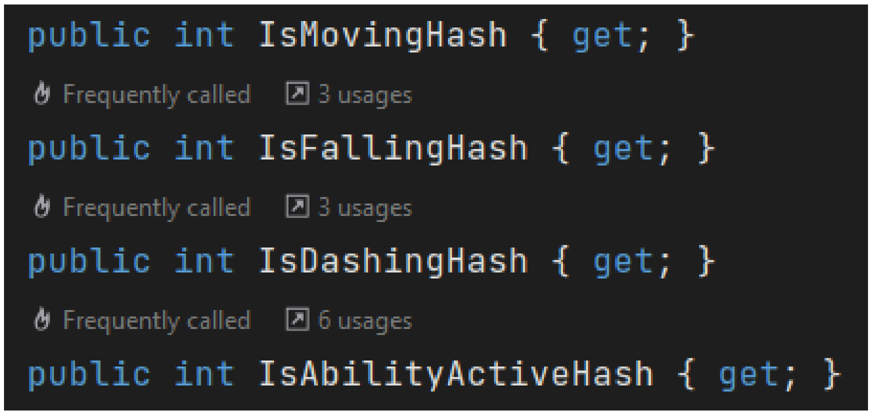
Parameter Hashing

Looking at the Unity documentation on Setting a boolean parameter in an Animator we can see that there are two method overloads; one takes a string and a bool while the other takes an int and a bool. We can use the string, which must correspond to the name of one of the parameters in an Animator, to change its boolean value. However, this is a slow operation and our FSM changes states very frequently. The solution is to use the integer which is an id for a parameter.

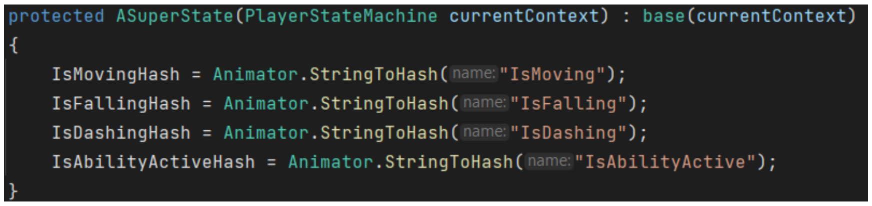
Animator has a static method called StringToHash which takes a string as an argument and returns an integer. The returned integer is an id for a parameter. And with this id, we can access the parameter faster. To use it, we can still call the *SetBool* method. Instead of passing in a string, we pass in the integer, and then we can specify the boolean we want.

ASuperState has already provided a number of id's. Their values are initialized in the constructor of *ASuperState* and can be accessed from outside the class.

Ideally, we do not need to touch this code unless a new parameter is introduced. An exception is the IsInSuperStateHash, which is an abstract int property. The reason why we enforce the implementation in its child class is because, at the level of ASuperState, we do not know the parameter that controls the entry and exit of a Sub State Machine yet. In other words, we have to wait until we create a Super State that we know which parameter we should use for controlling the in and out of the Sub State Machine that is related to that Super State.



```
public int IsMovingHash { get; }
Frequently called   3 usages
public int IsFallingHash { get; }
Frequently called   3 usages
public int IsDashingHash { get; }
Frequently called   6 usages
public int IsAbilityActiveHash { get; }
```



```
protected ASuperState(PlayerStateMachine currentContext) : base(currentContext)
{
    IsMovingHash = Animator.StringToHash(name: "IsMoving");
    IsFallingHash = Animator.StringToHash(name: "IsFalling");
    IsDashingHash = Animator.StringToHash(name: "IsDashing");
    IsAbilityActiveHash = Animator.StringToHash(name: "IsAbilityActive");
}
```

Figure 14. Hashed ids in *ASuperState* and the initialization.

Related Manager Classes

All the manager classes listed below derive from *ASingletonScriptableObject*, which are at the same time singletons and ScriptableObjects. The properties of a manager can be accessed by typing its name with the dot notation followed by the identifier “Instance” and another dot. It is very similar to other implementations of the singleton pattern.

One advantage of using a ScriptableObject to make a singleton is that it is not scene dependent. This is achieved by default without having to call *DontDestroyOnLoad*. Another advantage is that we can more easily reference other assets in our manager classes. They work just like one ScriptableObject referencing another ScriptableObject or a prefab. Note that I said “assets,” meaning that we still cannot reference a GameObject in a scene.

PlayerStateFactoryManger

PlayerStateFactoryManager is a manager that is also a factory. It has the entire collection of all possible states, both Super and Sub States. In the Factory Pattern section, we talked about how we can access this class to grab the state we need. There is still an unanswered question though: How do the states know about the context and the factory? Because at the end of the day, the states need to know both in order for them to carry out their own tasks.

This is partially answered in Figure 6, where we can see that the context and the factory are specified in the constructor of *APlayerState*. But who calls the constructor? The answer is *PlayerStateFactoryManager*. It calls the constructors of all states in its public method *InitContext* which is called by the *PlayerStateMachine* itself in *Awake*.

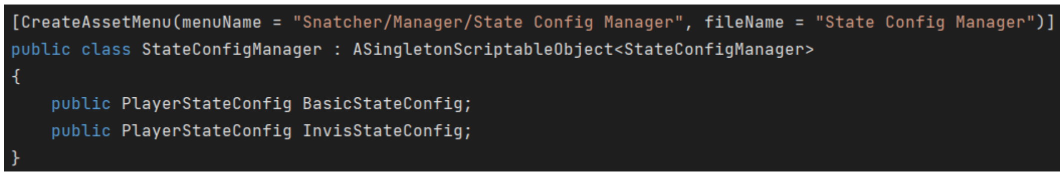
To summarize the process, we start with the initialization of the *PlayerStateMachine*, a MonoBehaviour. When its GameObject is loaded, the *Awake* method is called. In the *Awake* method, it calls *PlayerStateFactoryManager.Instance.InitContext(this)*. In other words, we pass this object to the manager and specify that *this* is the current instance of the context. The manager then initializes all the states with this specific context. As a result, all the states know who the context is. In addition to that, upon initialization, the states also hook up the reference to the manager like shown in Figure 6.

The disadvantage of adopting this implementation is that there can only be one instance of *PlayerStateMachine*. This is because whenever a new instance is created, it will re-initialize all the states with itself and override the old one. This is currently not a problem because there is only one context present at any given moment.

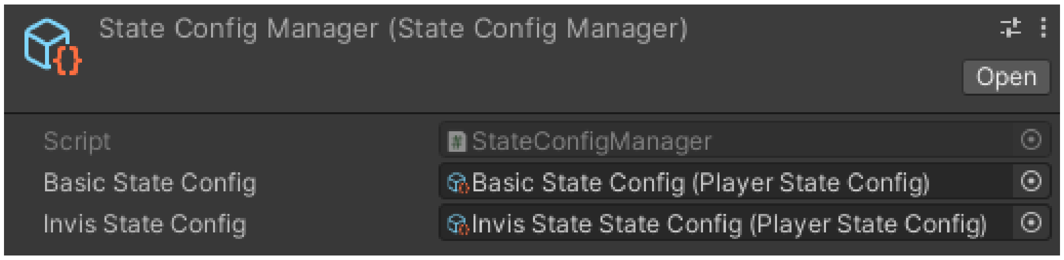
The advantage of this is that the Limbs can also reference this manager. The Limbs are of type *ALimb*, which is a pure C# class. An instance of *ALimb* has to know its corresponding Ability Entry State. This can be easily done with the use of a singleton. The beauty of this is that a Limb will not need to know the context, but will still get access to the Ability State that works with the context. With the manager as the middleman, we eliminate the hard-wiring of *ALimb* to an instance of the context, which is an instance of the MonoBehaviour *PlayerStateMachine*.

StateConfigManager

StateConfigManager is a container that has all the references to configuration ScriptableObjects for each state. The configuration ScriptableObjects are of type *PlayerStateConfig*. To make a new reference to a *PlayerStateConfig*, just declare it as a public field. This is one of the few exceptions where public fields are used. An example is shown in Figure 15. After creating the field, you will need to drag and drop the actual *PlayerStateConfig* ScriptableObject asset into the *StateConfigManager* ScriptableObject asset.



```
[CreateAssetMenu(menuName = "Snatcher/Manager/State Config Manager", fileName = "State Config Manager")]
public class StateConfigManager : ASingletonScriptableObject<StateConfigManager>
{
    public PlayerStateConfig BasicStateConfig;
    public PlayerStateConfig InvisStateConfig;
}
```



State Config Manager (State Config Manager)

Script	StateConfigManager
Basic State Config	Basic State Config (Player State Config)
Invis State Config	Invis State State Config (Player State Config)

Figure 15. Above is the implementation *StateConfigManager*. Below is the Inspector window view of the *StateConfigManager* ScriptableObject asset. We can see it references two other ScriptableObjects which are of type *PlayerStateConfig*.

LimbManager

LimbManager handles the collection of Limbs Snatcher has at a given moment. It provides public properties where other classes can access information such as the current, next, and prior Limbs. Again, the Limbs are derived classes from *ALimb*. The underlying data structure of the Limb collection is a *List* and an index of type int.

The manager also has public methods *SwitchLimb* and *DecrementLimbDurability*. Calling *SwitchLimb* will cause the internal index to either increment or decrement, and thus change the current, next, and prior Limb. *DecrementLimbDurability* decreases the durability of the current Limb which is an integer kept within the *ALimb*.

Upon calling *SwitchLimb*, the manager also invokes an event called *OnLimbSwitched*. Other entities listen to the invocation of this event. However, it is beyond the scope of this Guideline. For more information about events, see the Guideline on SO Event System.

Calling *DecrementLimbDurability* will also cause the event *OnAbilityUsed* to be invoked. Other entities listen to the invocation of this event.

ScriptableObject Event System Manual

Overview

Rationale

The observer pattern is a good way to decouple the codes that we write. Imagine that we want the UI to know about a player's death so that it can show a pop-up text "You Died." A way to do it is to have the player reference the UI and call a method when it is dead. The problem is that if the UI is not in the scene (maybe you're just testing movement abilities), you now get a null reference exception. In order for you to test the movement abilities, you need to drag in the UI prefab; it is a lot of trouble and it doesn't make sense.

The solution is to utilize the observer pattern. Instead of having the player control the UI, we can have the UI *listen* to the player. The player doesn't care about the UI at all. When the player dies, the player *announces* its death, like how you will announce an event. Everyone that listens to the player's death event will get notified.

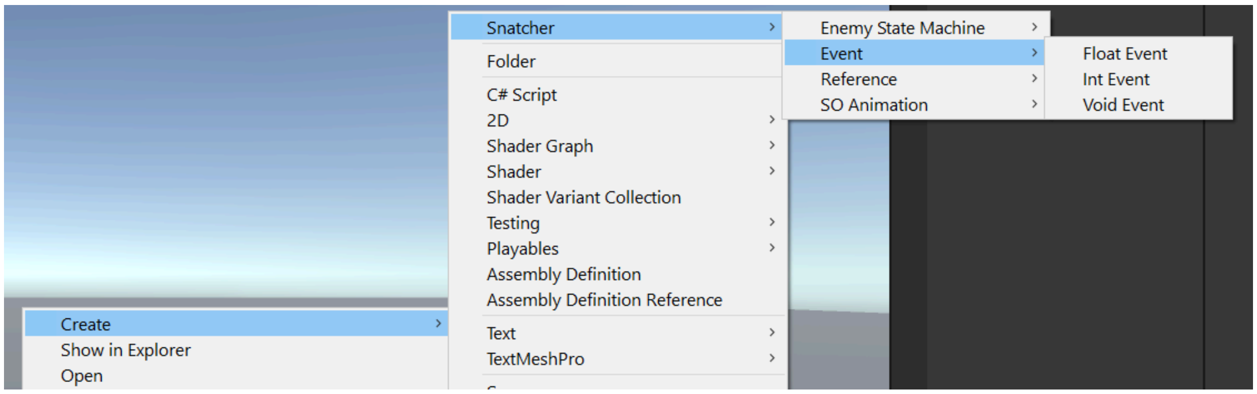
The bad news is that we have to set up a way of communication so that the player does not know or care about the UI. The good news is that, once set up, we can have as many listeners as we want, and the player does not need to know about ANY of them. The player's only job is to announce its death when it happens.
(Go to the Example Section for more detail)

Preface

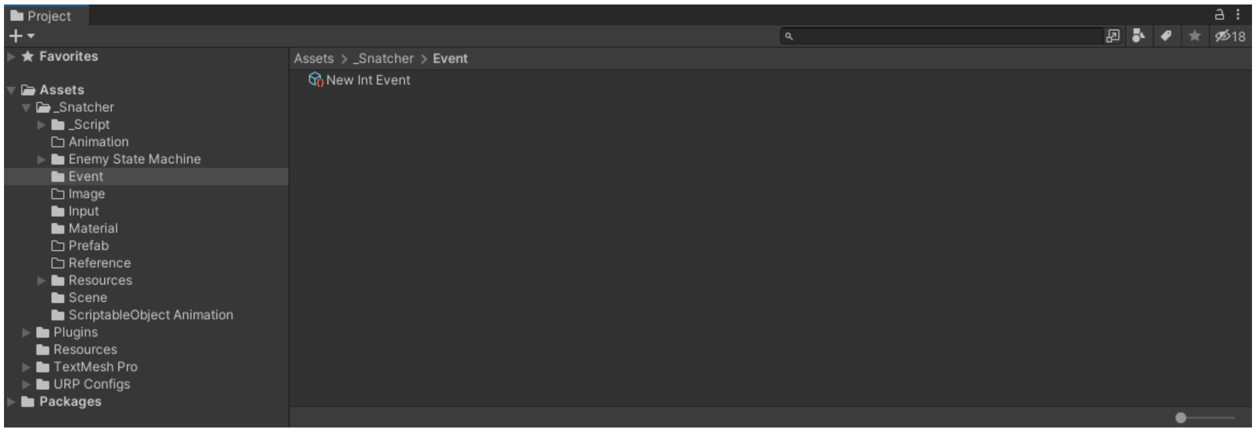
The implementation of the SO event system in our project is largely based on Dapper Dino's implementation. Check out his videos on the topic if you're interested. Part 1 and Part 2 The only difference between his implementation and ours, is that I added an additional option to subscribe to an SO event as you will see later.

Manuals

How to Create an Event?



In the Assets/Event folder (or you can create a subfolder for organization), ***right-click* > Create > Snatcher > Event > *choose one***.



After that, you will get something like this. You just made an event, and it is ready for other scripts to reference it.

What are the types of events?

Commonly seen event types include: Void Event, Int Event, Float Event, and Vector3 Event. What does that mean? The *type* determines what gets *passed down* when the event is invoked. Imagine we want to let the enemies know the player's location when the player lands. What we can do is to have the player invoke a Vector3 Event called *OnPlayerLanded* and pass its current location into the event upon invoking it. What will happen is that all the enemies that have subscribed to *OnPlayerLanded* will receive a notification, not just about the fact that the player has landed, but also the location where the player has landed.

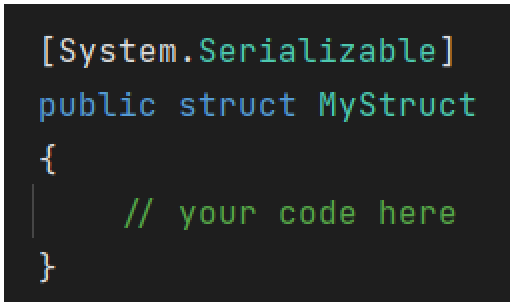
A more abstract idea has to do with the Void Event. What does a Void Event pass around? Actually, nothing. For the subscribers, it's like knowing that something just occurred but no other information is provided. It is similar to subscribing to an Action event in C# style, whereas a Vector3 Event is like Action<Vector3>.

A side-note here: notice that I spelt Void with a capital v. That is because "void" is a keyword in C#, and "Void" is a boilerplate struct that I made for this system to work. You will not need to know the Void struct that gets passed around upon an event's invocation because it will not contain any useful information.

What if the type I need is not present?

Let me know. I will make it for you.

Or watch the videos. Just know that when defining an event whose type is a pure C# class or a struct (that means the object is not a Unity Object like Vector3 or CanvasGroup), make sure you add the attribute *[System.Serializable]* as show below:



```
[System.Serializable]
public struct MyStruct
{
    // your code here
}
```

How to Subscribe?

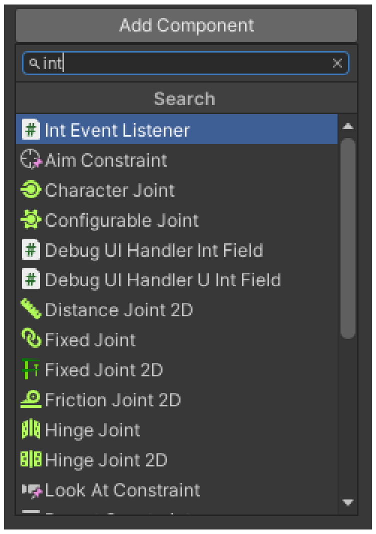
There are two ways you can subscribe to an SO event: through *IEventListener* or through classic C# style *action callback*. Before we start, I need you to know that IEventListener is recommended as it is **safer** and it gets invoked **before** C# style action callbacks.

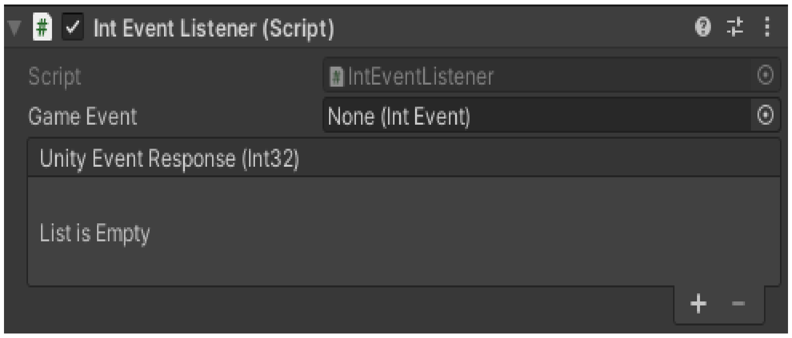
IEventListeners require more steps to set up, but it ensures the subscribing and unsubscribing are done automatically for you. Action callbacks offer more convenience and are quicker to set up. However, you do need to handle the unsubscribing part on your own. Otherwise, there will be errors.

IEventListener

The *IEventListener* is an interface that all the listener scripts in the system implement. That means *IntEventListener*, *FloatEventListener*, *Vector3EventListener*, and *VoidEventListener* (and others we might implement in the future), all have the ability to respond to the invocation of an event.

How they did that is unimportant. What is important is that, if you want to subscribe to an int event, you need to use an IntEventLister. And, you will attach an IntEventLister onto a GameObject. Preferably, it is the GameObject that has the script in which there is a function you want to call. It becomes important when it comes to subscribers that are in prefabs.





Int Event Listener (Script)

Script: IntEventListener

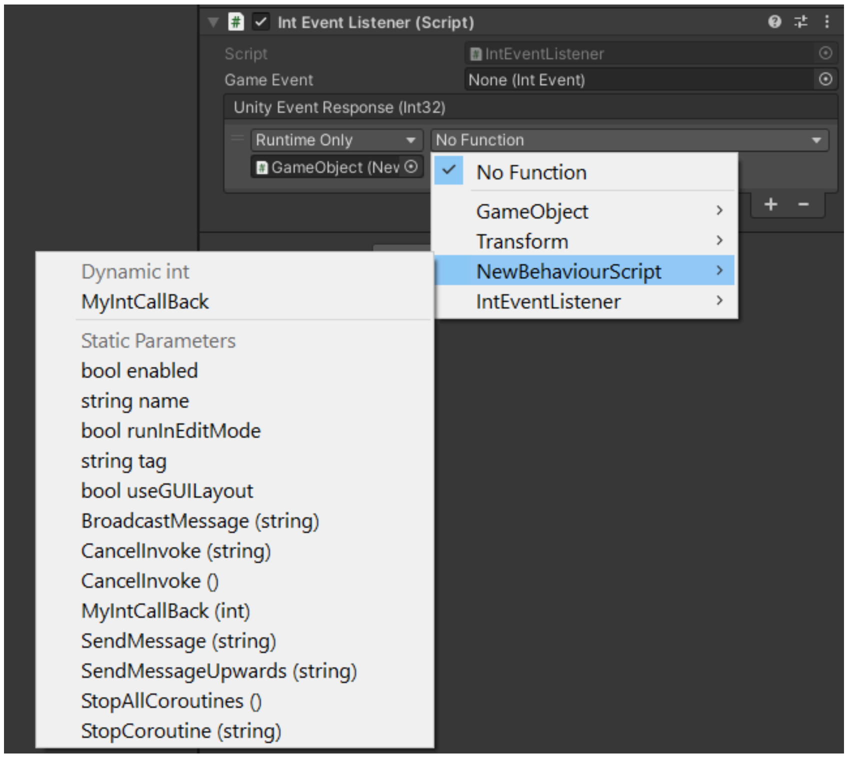
Game Event: None (Int Event)

Unity Event Response (Int32)

List is Empty

The left image shows that you can type in the search bar to look for IntEventListener. On the right, you can see that, after adding the component, it expects a *Game Event* that is of type *IntEvent*, which is an int SO event. You can also see that it expects some UnityEvents. The way this works is the same as that of a UI button's, so I won't go over it here. Just note that the method you want to call needs to be public in order for it to be accessible from the Inspector.

Also, note that to hook up methods for non-Void Events, you probably want to reference it **dynamically** for the method to actually use the value that gets passed down by the publisher. The image below shows there are two options for calling *MyIntCallback*, one under the label "Dynamic int" and the other "Static parameter." The one under "Dynamic int" is usually the one we want.



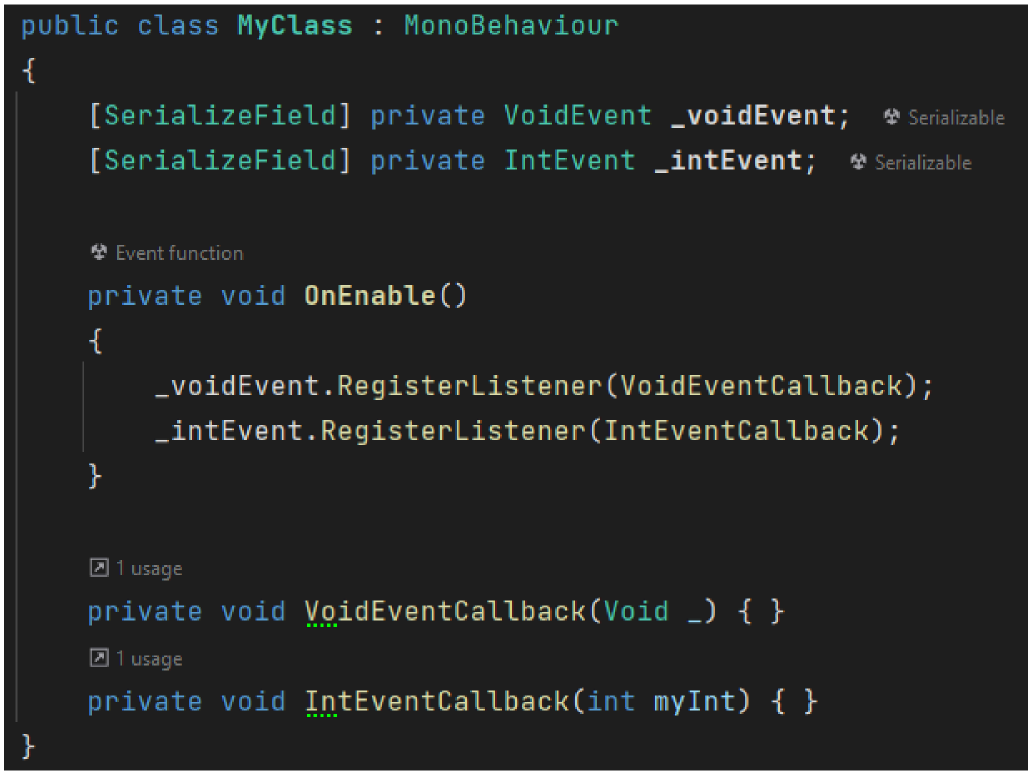
That is the process of setting up an IEventListener, in this case an IntEventListener.

(Remark: UnityEvent has a reputation of being slow and not performant. We might have to deal with this in the future, but for now, it should be fine.)

Action Callbacks

If you're not familiar with C# delegate or event, I suggest you watch the two-video series by Sebastian League, to go over the basics.

How we set up to subscribe to an SO event using C# style action callbacks, is very similar to how we invoke one. Below is the demonstration of how to subscribe to a VoidEvent and an IntEvent. Pay attention to the arguments that are passed in into the *RegisterListener* methods of the two events. Also pay attention to the parameter of *VoidEventCallback* and that of *IntEventCallback*.



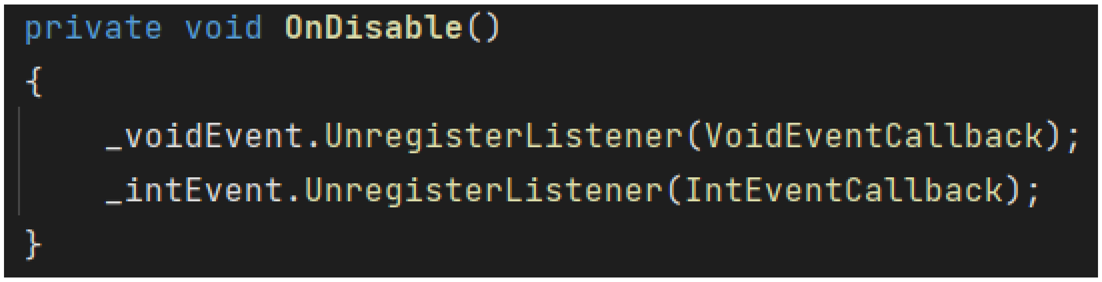
```
public class MyClass : MonoBehaviour
{
    [SerializeField] private VoidEvent _voidEvent;
    [SerializeField] private IntEvent _intEvent;

    private void OnEnable()
    {
        _voidEvent.RegisterListener(VoidEventCallback);
        _intEvent.RegisterListener(IntEventCallback);
    }

    private void VoidEventCallback(Void _) { }
    private void IntEventCallback(int myInt) { }
}
```

Two things are worth mentioning. Firstly, the callbacks are passed into RegisterListener without the *()* at the end. This is because we are not calling them, we are passing them around like variables. Secondly, the parameter types of the callbacks match the types of their events respectively. The "*Void _*" is just syntactic stuff that indicates we won't be using this parameter in this callback. As for IntEventCallback, we do potentially want to use *myInt*.

After we are done subscribing to the events, it is important that we somehow unsubscribe it.

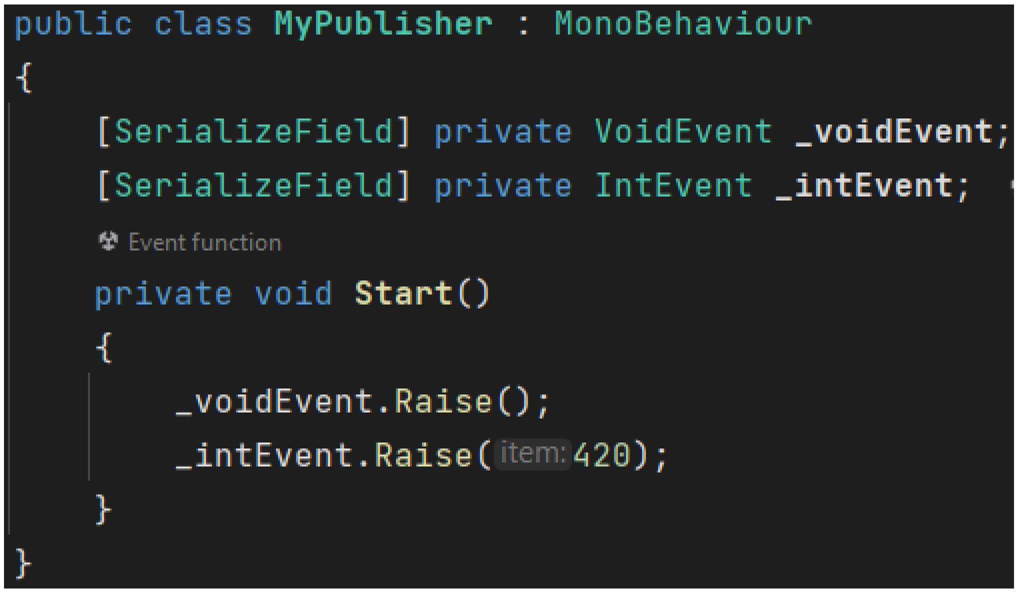


```
private void OnDisable()
{
    _voidEvent.UnregisterListener(VoidEventCallback);
    _intEvent.UnregisterListener(IntEventCallback);
}
```

In this example, I choose to unsubscribe from the event inside OnDisable. This is probably the single most common place to unsubscribe from an event because once a MonoBehaviour is disabled, all the other logics within it should stop working as well.

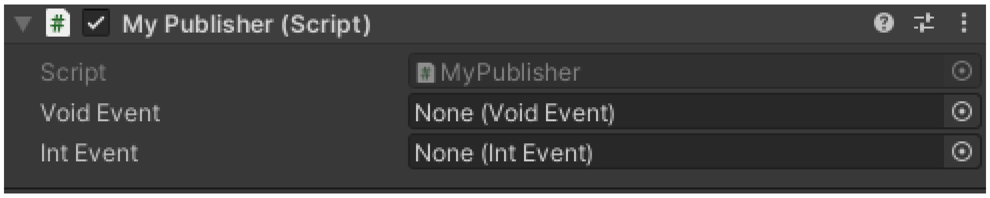
How to Raise an Event?

Raising an event is pretty straight forward. Get a reference to the event that will be invoked. And call the *Raise* method when you want to invoke it.



```
public class MyPublisher : MonoBehaviour
{
    [SerializeField] private VoidEvent _voidEvent;
    [SerializeField] private IntEvent _intEvent;
    Event function
    private void Start()
    {
        _voidEvent.Raise();
        _intEvent.Raise(item: 420);
    }
}
```

Here I have an example of raising two events in the Start method. For the VoidEvent, there is no need to pass in any argument. For the IntEvent, it does expect an integer for input.



```
My Publisher (Script)
Script          MyPublisher
Void Event      None (Void Event)
Int Event       None (Int Event)
```

In the Inspector, we can now add events to it.

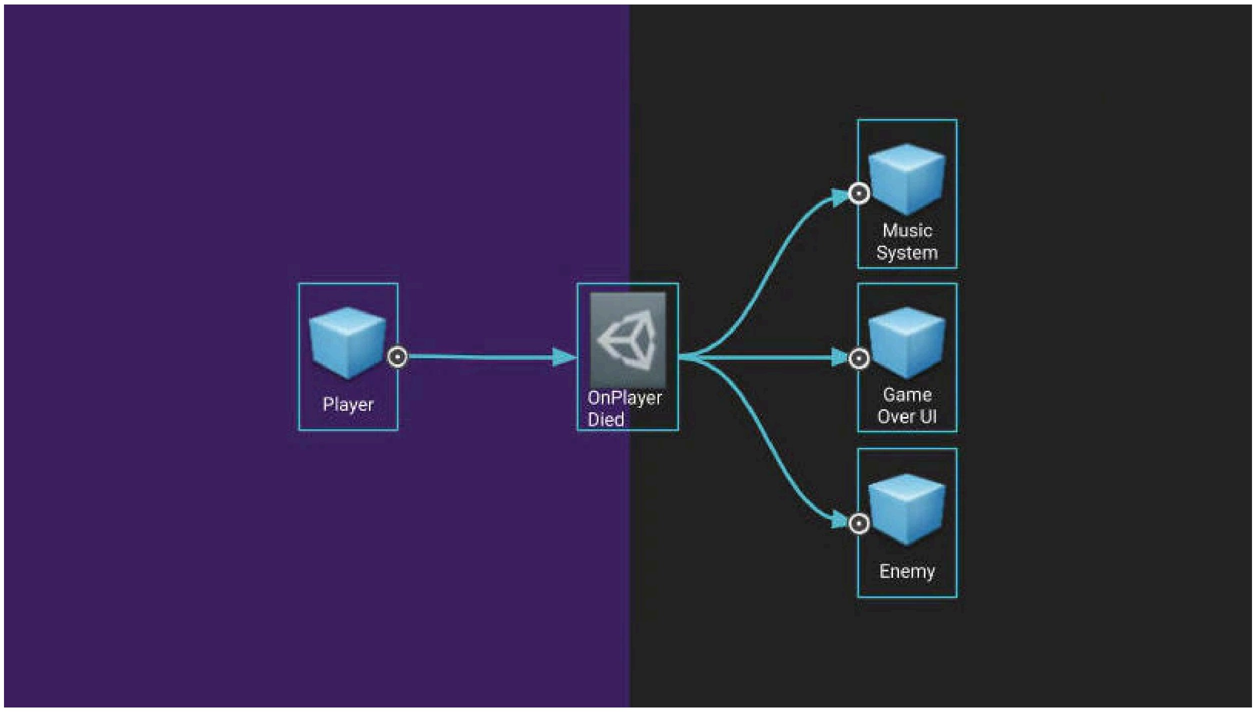
Caveats

Event Dictionary

One thing to keep in mind when using this system is that the Unity Editor isn't helpful in terms of keeping track of who's referencing who. Sometimes, we will lose track of it, and it can be a pain to debug. **You must note the publisher(s) and subscriber(s) of an SO Event.**

The way we will cope with this problem is to use an Event Dictionary. Event Dictionary can be found in the Documentation folder of this Google Drive. Open up the Summary to quickly reference the SO event of interest.

Example



(Source: Three ways to architect your game with ScriptableObjects)

The diagram above demonstrates the basic framework of an event system implemented using Unity Engine's ScriptableObject (SO). The *Player* is the **publisher** of the SO event *OnPlayerDied*, which is the **event** itself. *Music System, Game Over UI,* and *Enemy* are the **subscribers** of the event. The implementation of this pattern in our project is a little more advanced. However, the underlying concept is the same.

It goes like this: *Music system*, *Game Over UI*, and *Enemy* subscribe to the *OnPlayerDied* event. The *Player* holds a reference to the *OnPlayerDied* SO event, which sits inside the Assets folder. When the *Player* **invokes** the event, *Music system*, *Game Over UI*, and *Enemy* all get notified.

The *Player* can be a MonoBehaviour (the type of script you attach to a GameObject), or it can be other things as well. As long as it can hold a reference to the *OnPlayerDied* SO event, it can invoke it. The beauty of it, besides many amazing things events provide us, is that the SO event sits in the asset folder regardless of which scene is currently active. It is essentially an asset. As long as you can hold a reference to it, you can always call it.

Debug Logging

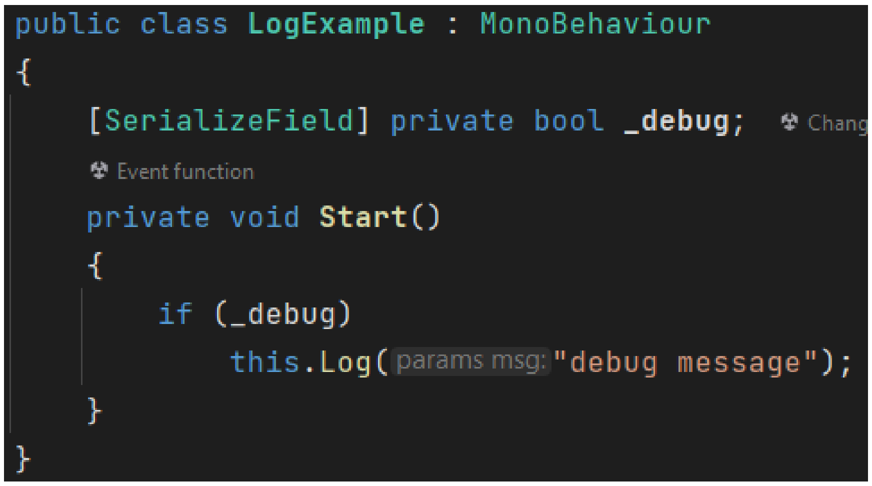
TL;DR

Instead of using *Debug.Log("your message here")*, use *this.Log("your message here")*. Your logging message should appear in the Console as usual, with the exception that it now shows which entity is logging.

Toggling Logs

Try to make a boolean variable exposed in the Inspector that allows you to toggle on or off logging messages. This is pretty easy; you just need to put the log function call in an if-block, and you want to check against the toggle boolean value.

Besides adhering to the coding conventions, you should name that boolean variable _*debug* unless you have multiple levels of details you need to debug. Mark the variable [SerializedField] private, and you can switch on/off logging to the console without having to go to the script.

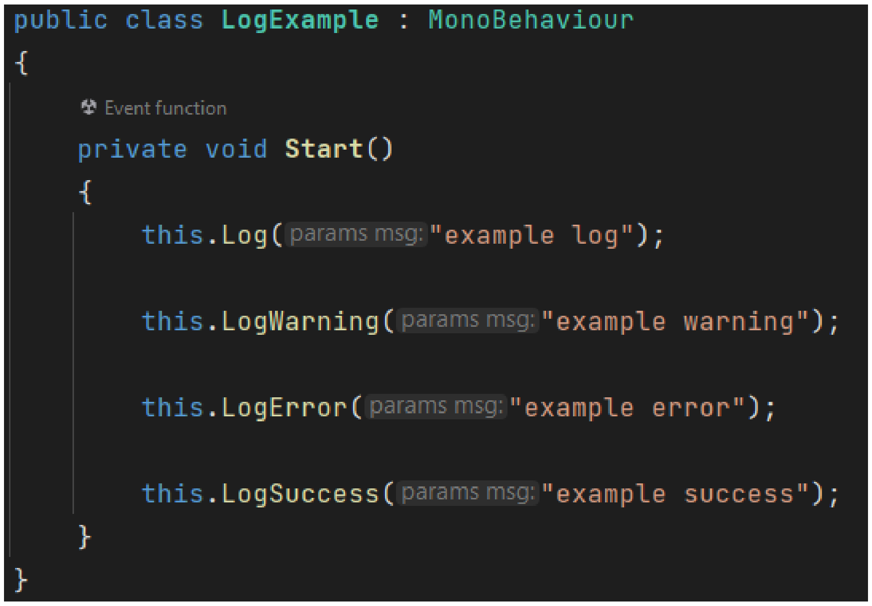


```
public class LogExample : MonoBehaviour
{
    [SerializeField] private bool _debug;  Chang
    Event function
    private void Start()
    {
        if (_debug)
            this.Log(params msg: "debug message");
    }
}
```

Figure 1. Toggle of a Log method.

The new logging system will not compile the logging behavior when making a build. So, we don't need to worry about remnant logging function calls in the code base. That being said, it is still a good idea to remove unused calls to log, unless it is a debugging tool that you constantly need to use. For example, when setting up a new state for the FSM, you probably want to see what state the FSM enters. It is therefore a good idea to leave the call in the script with a toggle like shown above, and use it in need.

Logging Methods



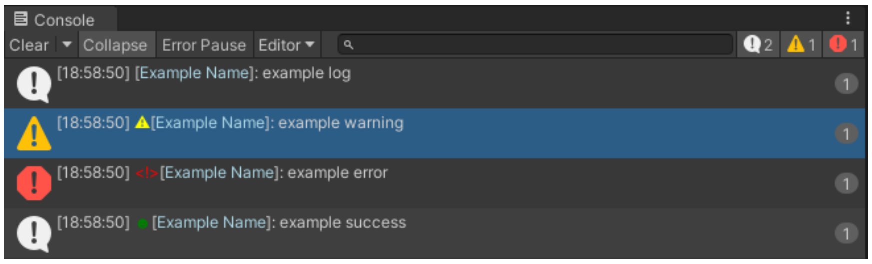
```
public class LogExample : MonoBehaviour
{
    Event function
    private void Start()
    {
        this.Log(params msg:"example log");

        this.LogWarning(params msg:"example warning");

        this.LogError(params msg:"example error");

        this.LogSuccess(params msg:"example success");
    }
}
```

Figure 2. Demonstration on how to use logging methods.



```
[18:58:50] [Example Name]: example log            1
[18:58:50] [Example Name]: example warning        1
[18:58:50] <!>[Example Name]: example error       1
[18:58:50] [Example Name]: example success        1
```

Figure 3. Console results from Figure 2.

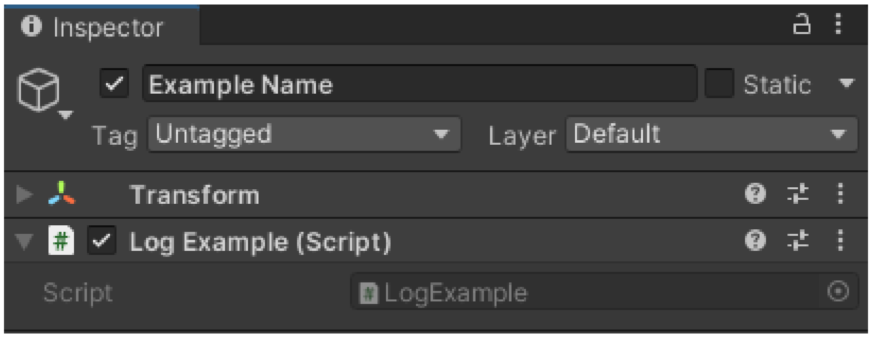
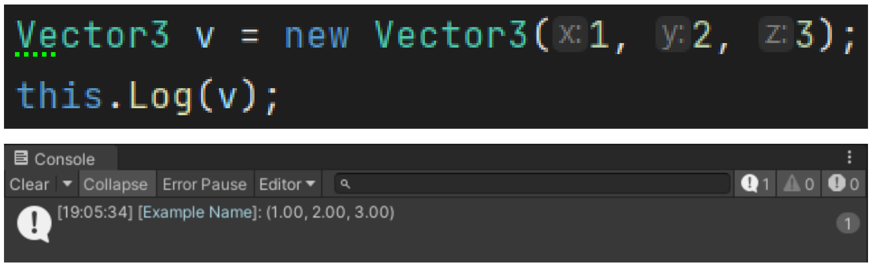


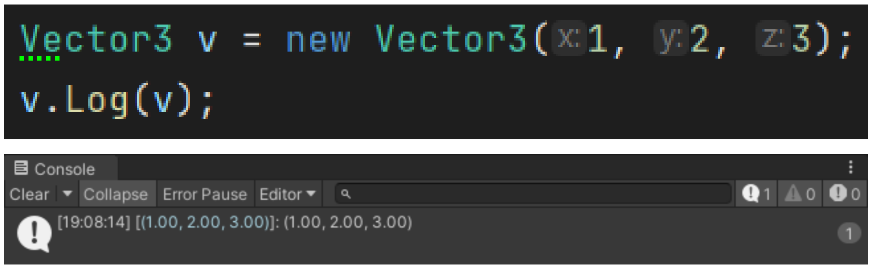
Figure 4. The name of the GameObject is also printed in the Console. In this example, the name of the Game Object “Example Name” is used in the logged messages in Figure 3.



```
Vector3 v = new Vector3(x:1, y:2, z:3);
this.Log(v);
```

```
[19:05:34] [Example Name]: (1.00, 2.00, 3.00)
```

Figure 5. Variables besides strings can also be logged.



```
Vector3 v = new Vector3(x:1, y:2, z:3);
v.Log(v);
```

```
[19:08:14] [(1.00, 2.00, 3.00)]: (1.00, 2.00, 3.00)
```

Figure 6. Logging methods can also be called directly on variables like extended methods.

Identifying Logging GameObject

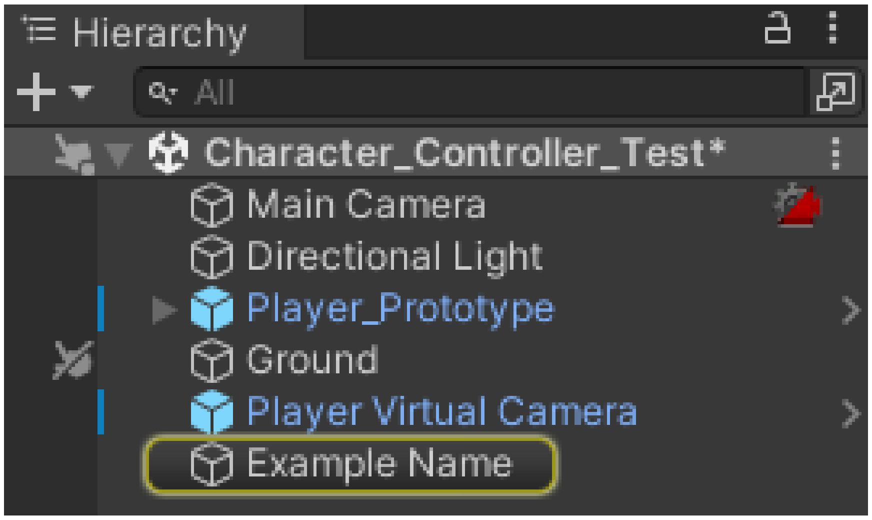


Figure 7. Highlighting the logging GameObject.

If you click on a logged message in Console, it will highlight the GameObject that makes the function call. However, to achieve this, there are two requirements to be met.

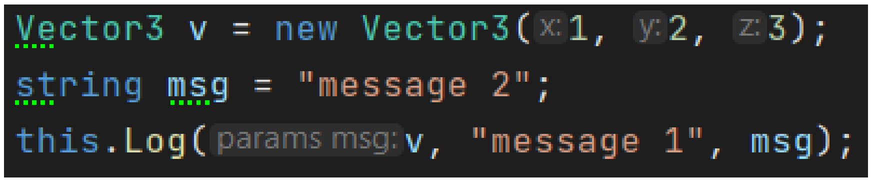
Firstly, it has to be in a MonoBehaviour script. It makes sense because only Monobehaviour scripts can be attached to GameObjects, and everything in the Hierarchy is a GameObject. Therefore, when you want to highlight something in the Hierarchy, it implies that such an entity is a GameObject that has a MonoBehaviour script attached to it.

Secondly, the logging method has to be called in such a fashion as shown in Figure 5. It has to be called by calling on the *this* keyword. It cannot be called like shown in Figure 6.

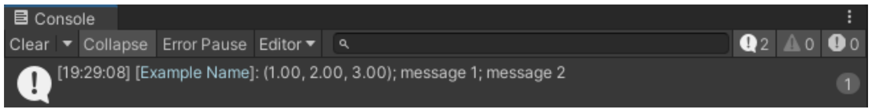
If both requirements are met, you can click on the message, and it will show the GameObject in the Hierarchy that called it.

Concatenation

You can pass in more than one argument into the logging methods. The logging methods will concatenate the arguments and separate them using semicolons.



```
Vector3 v = new Vector3(x:1, y:2, z:3);
string msg = "message 2";
this.Log(params msg:v, "message 1", msg);
```



```
[19:29:08] [Example Name]: (1.00, 2.00, 3.00); message 1; message 2
```

Figure 8. Demonstration on message concatenation. Arguments are separated with semicolons.

Cinemachine Camera Setup

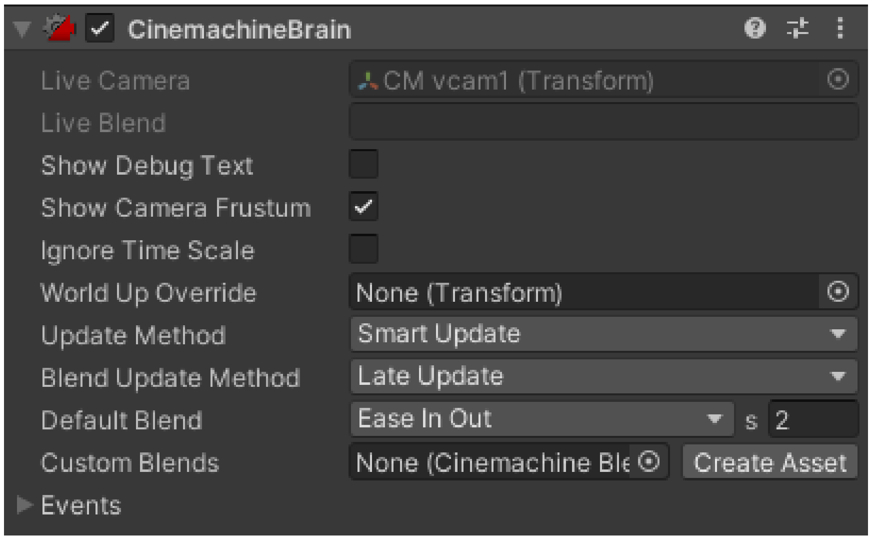
Cinemachine Brain

Cinemachine Brain is a component that gets automatically added to the Main Camera when you add a *Cinemachine Virtual Camera* into a scene. (If you drop a prefab with a *Cinemachine Virtual Camera* component, it won't automatically do so.) You will see an icon on the Main Camera in the Hierarchy like shown in Figure 1.

The component detail itself is shown in Figure 2. There isn't anything we want to change in this component for now.

Two things to note: (a) **we need to update the *Camera* component on Main Camera to use *Orthographic* for the *Projection***, shown in Figure 3, (b) if you added a Cinemachine Virtual Camera in the scene, you cannot move the Main Camera now.

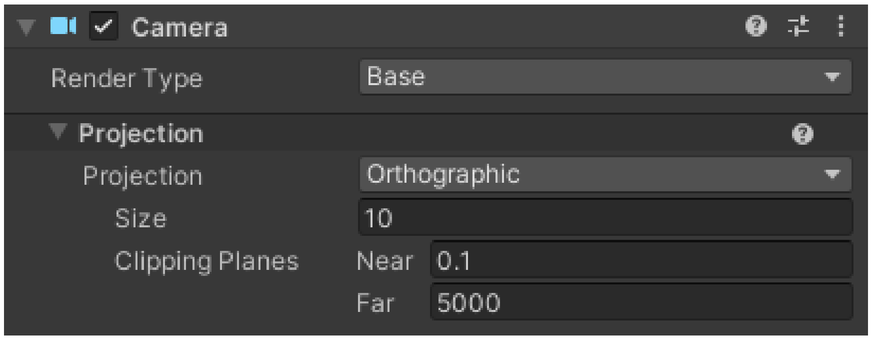
Figure 1. Main Camera in Hierarchy



CinemachineBrain

Live Camera	CM vcam1 (Transform)
Live Blend	
Show Debug Text	
Show Camera Frustum	✓
Ignore Time Scale	
World Up Override	None (Transform)
Update Method	Smart Update
Blend Update Method	Late Update
Default Blend	Ease In Out s 2
Custom Blends	None (Cinemachine Ble Create Asset
Events	

Figure 2. Details of Cinemachine Brain component. The *Live Camera* indicates which *Cinemachine Virtual Camera* the *Brain* is currently using.



Camera

Render Type	Base
Projection	
Projection	Orthographic
Size	10
Clipping Planes	Near 0.1
	Far 5000

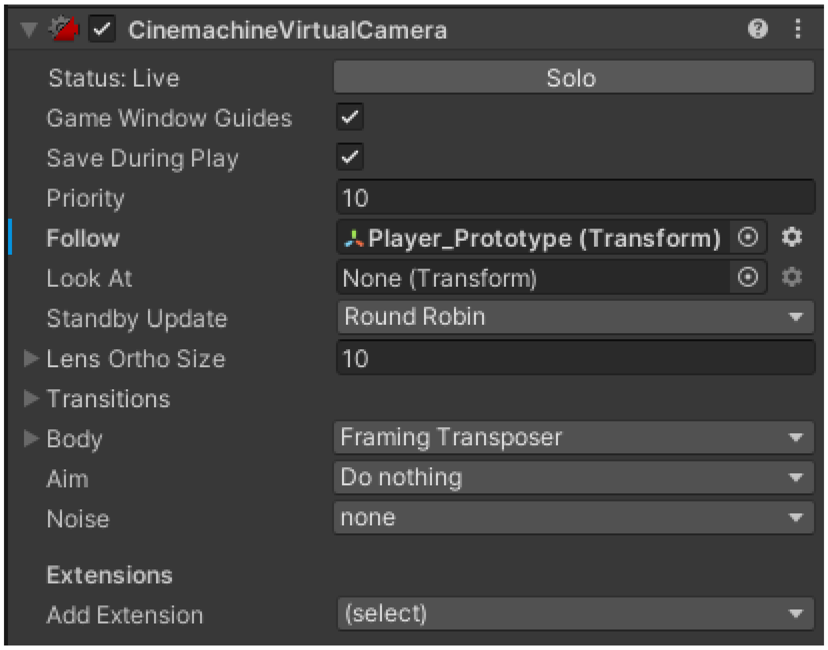
Figure 3. Projection of Camera component needs to be updated to Orthographic

Cinemachine Virtual Camera

A Cinemachine Virtual Camera is essentially an empty Game Object with a Cinemachine Virtual Camera component on it. The reason why the Main Camera cannot move once you add a Cinemachine Virtual Camera to a scene, is that it is now the Virtual Camera's job to move around and act as the camera that we actively use. You will notice that the Main Camera's Transform Position and Rotation are the same as those of the currently active Virtual Camera.

To use a Cinemachine Virtual Camera, we just need to drag and drop a prefab and hook up the reference of our player Game Object.

1. Set the *Projection* of the Main Camera to *Orthographic*
2. Add a *Cinemachine Brain* component to the Main Camera
3. Go to _Snatcher > Prefab > drag and drop *Player Virtual Camera* to the scene
4. In the Hierarchy, select the *Player Virtual Camera* you just dropped
5. In the Inspector, drag and drop the player Game Object into the *Follow* field of the Virtual Camera component, like shown in Figure 4.
6. The Virtual Camera should now follow the player as it moves around


CinemachineVirtualCamera

Status: Live	Solo
Game Window Guides	✓
Save During Play	✓
Priority	10
Follow	**Player_Prototype (Transform)**
Look At	None (Transform)
Standby Update	Round Robin
Lens Ortho Size	10
Transitions	
Body	Framing Transposer
Aim	Do nothing
Noise	none
Extensions	
Add Extension	(select)

Figure 4.

General UI Canvas Setup

Canvas Component

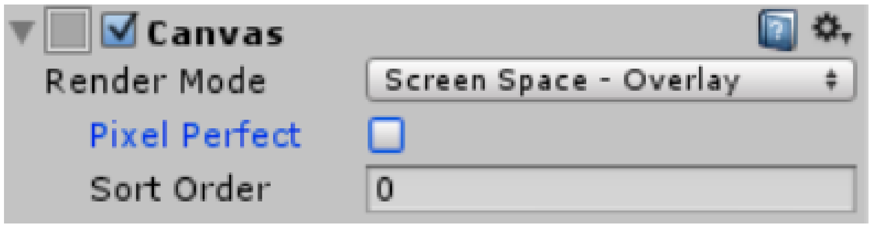

Canvas
Render Mode: Screen Space - Overlay
Pixel Perfect: ☐
Sort Order: 0

Figure 1. Canvas component.

We want the *Render Mode* to be **Screen Space - Overlay**, which is the default anyway. There will be cases where we need popup damage text or overhead health bar displays. In those cases, we will use something else.

Canvas Scaler (IMPORTANT)

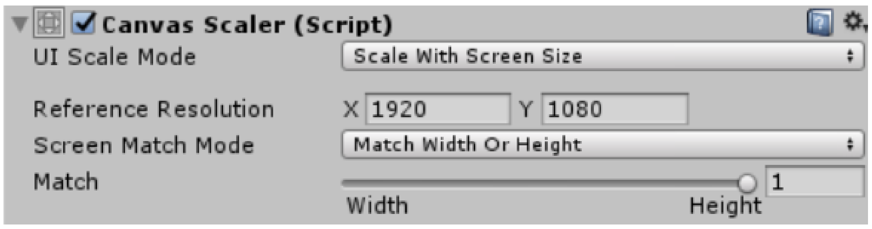

Canvas Scaler (Script)
UI Scale Mode: Scale With Screen Size
Reference Resolution: X 1920 Y 1080
Screen Match Mode: Match Width Or Height
Match: 1

Figure 2. Canvas Scalar.

There are four things we need to make sure they are in place:
1. *Ui Scale Mode* should be set to **Scale With Screen Size**
2. *Reference Resolution* should be **1920 by 1080**
3. *Screen Match Mode* should be **Match Width Or Height**
4. *Match* should be set to **0.5**

Interactable Setup

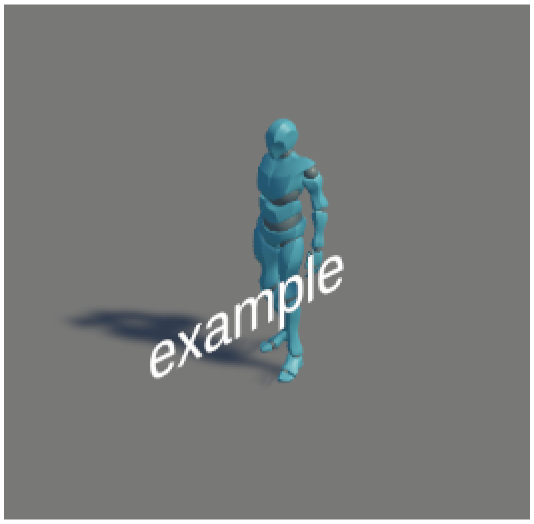
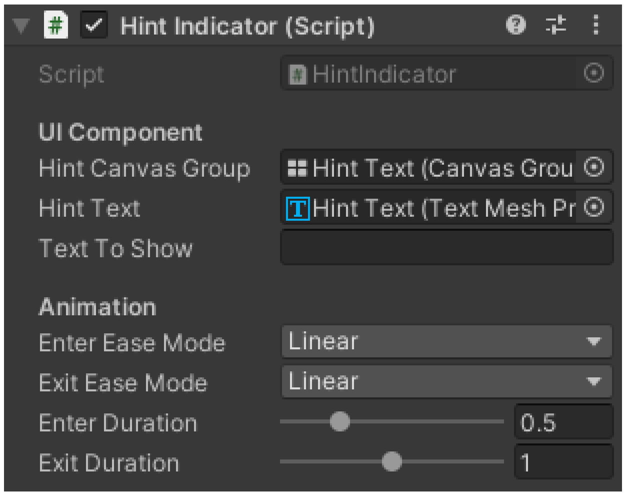


Figure 1. World-space UI Indicator.

Steps to Set Up

1. _Snatcher > Prefab > Interactable > go to *Teleporter* or *Unlocker* > drag and drop the prefab into the scene
2. Adjust the size of the *Box Collider* if needed; **DO NOT** change the scale of the GameObject itself
3. Type in text in the *Hint Indicator* component, as shown in Figure 2.
4. Adjust the Y rotation of the GameObject's Transform if needed
5. Tweak settings in Hint Indicator and *TeleportController* or *UnlockerController*



Hint Indicator (Script)

Script: HintIndicator

UI Component
Hint Canvas Group: Hint Text (Canvas Grou
Hint Text: Hint Text (Text Mesh Pr
Text To Show:

Animation
Enter Ease Mode: Linear
Exit Ease Mode: Linear
Enter Duration: 0.5
Exit Duration: 1

Figure 2. Hint Indicator Component. *Text To Show* input field is where to put in the hint text. Hover over the Animation-related fields to see explanations.

Interactables

Inside Prefab > Interactable, there is a prefab called *Interactable*. This prefab has two prefab variants, *Interactable_Teleporter* and *Interactable_Unlocker*. *Interactable* has a *Hint Indicator* script on it that is responsible for showing UI when the player enters the box collider. Additionally, *Interactable_Teleporter* has a *TeleportController* script attached to it that defines the instance's teleport behavior. Similarly, *Interactable_Unlocker* has an UnlockerController on it and it defines the unlocking behavior of that instance.

Interactable Creation

If you need something that changes an SO reference, you need an *Interactable_Unlocker*. If you need to teleport the player when it presses F, use an *Interactable_Teleporter*. There are some readymade prefab variants of the two kinds. For example, *Interactable_WoodenDoor* is a prefab variant of *Interactable_Teleporter*, and *Interactable_Key* is a prefab variant of *Interactable_Unlocker*.

If you need to make your own, make a prefab variant out of either *Interactable_Unlocker* or *Interactable_Teleporter* because they are the two most common types of *Interactables*. After you have made a prefab variant out of your choice, drop in the model inside that variant.

Never scale the parent GameObject. If the box collider is too small, change the boundaries using the Box Collider component settings. If the UI text is too small, adjust the Canvas settings and TMP settings. If the model is too small, scale the model itself. **Do not 'Apply changes' to parent prefabs. It might accidentally break something.**

Event Dictionary

OnLimbSwitched

Publishers

- LimbManager
  - On the LimbManager ScriptableObject
  - When calling LimbManager.Instance.SwitchLimb(), this event is raised.

Subscribers

- LimbCanvas prefab
- PlayerLimbController on the Player Prefab