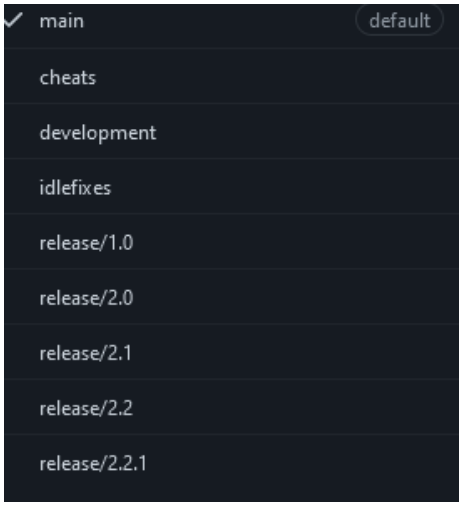
Ratiation Technical Design Document

Written by Daniël Meijs (TortleTurtle) & Thomas De Haas (Detravoir)

Version Control



For version control we are using Git and GitHub. A workflow method we are using is GitFlow. This ensures that all code is created safely without changing code someone else is working on. Our repository exists out of 4 categories of branches: main, development, feature, and release branches. The release branches all contain a version of a release. This can be major or minor releases. Main always contains the latest release and is identical to the last created release branch. Development contains the newest features but is not release ready. Team members create new 'feature' branches from this branch. Once a feature is finished a Pull Request is created to merge the feature into Development. The Pull Request helps find any potential merge conflicts and lets other developers look at your code to see if does not conflict with anything they have written. Once development contains enough new features for a new release a new release branch will be created. In this release branch everything will be checked and implemented. Build settings will be adjusted and once a successful build has been made the release branch will be merged into main and a release will be created on GitHub.

A more in-depth explanation of the workflow can be found here.

UML-Diagram

Throughout the project Daan has designed multiple systems using UML Diagrams and Lucid Chart. These can be found here. The final version is a full overview of the project and has been generated using PlantUML and PlantUML-Generator a text-based way to create UML diagrams that can be integrated into JetBrains Rider.

Scripts

Managers

Event Manager

A static class containing all the custom events. The event manager is static so every script can communicate with it and so it doesn't have to be added to the scene. It also does not inherent from Mono Behaviour since Delegates are built into .NET. The Event Manager does not subscribe methods to these events classes will subscribe and unsubscribe methods themselves. Invoking events is also done by the other classes. For example, when the Rat Manager spawns a rat it will invoke the On Rat Spawn Event.

List of events:

- On Game Loaded
  - Passes an instance of the Save Game Manger
- On Rat Merge
  - Passes the tier of the newly created rat as an integer.
- On Cheese Generated
  - Passes a double which is the amount of cheese generated.
- On Rat Spawn

Rat Manager

The Rat Manager controls the spawning of the rats. To spawn a rat, it first picks a random location within the playing area, after choosing a location it instantiates a rat prefab. It gives it a ratType and a Tier. After spawning the rat, it also puts this data in a list so that it can be loaded. When the game opens back up it knows how many rats of what tiers there needs to be spawned. After all this it runs an event to save the game.
This script makes use of a Coroutine. It checks the number of rats that are in the list, if it isn't on the max amount of 16 it will spawn a rat with the type and tier it requires. This will happen every few seconds and this can be lowered in the upgrades.

```
private IEnumerator SpawnRatTimer()
{
    while (true)
    {
        yield return new WaitForSeconds(ratSpawnInterval - spawnRateUpgrade.Level);
        if (SpawnedRats.Count < maxRats)
        {
            var type = Random.Range(0f, 100f) > 100 - 5 * spawnChanceUpgrade.Level ? ratTypes[1] : ratTypes[0];
            var tier = Random.Range(1, spawnHigherTierChanceUpgrade.Level + 1);
            SpawnRat(type, tier);
        };
    }
}

public void SpawnRat(RatType type, int tier)
{
    //Pick a location to spawn the rat
    Vector2 position = new Vector2(Random.Range(wall.bounds.extents.x, (wall.bounds.extents.x * -1)), Random.Range(wall.bounds.extents.y, (wall.bounds.extents.y * -1)));

    //Spawn a rat
    Rat newRat = Instantiate(ratPrefab, position, Quaternion.identity, null).GetComponent<Rat>();

    newRat.type = type;
    newRat.tier = tier;
    newRat.SetRat();

    SpawnedRats.Add(newRat);
    EventManager.OnRatSpawn?.Invoke();
}
```

It also has a function that gets called when 2 rats merge and 1 needs to be removed.
And a script that will spawn all the rats back when the game is loaded.

Save Game Manager

In this script, multiple variables will be saved in the playerprefs.
- Cheese Amount, here the currency of the player will be saved into the playerprefs.
- Rats, here it will make a list with all the rats based on their type and tier and save this into the playerprefs.
- Shop items, here it will loop through all the shop upgrades and save what tier you are on into the playerprefs.
- Statistics, here it will save all the calculated data and statistics in the Statistics Manager into the playerprefs.

This script does also load the same data it saved before and send it back to its respective script.

Statistics Manager

The statics manager tracks these statistics:

- Highest Tier Reached
- Total Merges
- Total Cheese Gained
- Total Cheese Per Second

It does this by subscribing its methods to specific events. The first three stats are also saved by the Save Game Manager. The last one does not have to be saved because all information needed to calculate it is contained by Rat and is already being saved.

The Statics manager has 2 methods subscribed to the On Rat Merge event: CheckHighestTier and AddToTotalMerges.

The AddCheeseToTotal method is subscribed to the On Cheese Generated event.

Unlock Manager

The Unlock Manager is an abstract class. It needs a list of game objects to unlock. Import is that these objects are set to Inactive since the they get unlocked by setting them to active.

The unlock manager works by subscribing its Unlock method to the On Rat Merge event. The integer that is passed through the event is used to unlock the items in the list.

Loading already unlocked items works similarly. When the On Game Loaded event is invoked. The Unlock Manager will call the Unlock Method but will pass Highest Tier Reached from the Statistics manager. This way it is not needed so save which items have already been unlocked individually.

The Unlock method is the only abstract method. This way it is possible to customise different speeds at which items unlock.

Index Unlock Manager

The Index Unlock Manager inherits from Unlock Manager. It's responsible for unlocking the index entries in the RatDex. It must implement the Unlock method. The index entries are unlocked one on one with the rat tiers. o when a Rat of tier 5 is merged all index entries up to and including rat 5 will be unlocked.

Rat Shop Unlock Manager

The Rat Shop Unlock Manager inherits from Unlock Manager. It's responsible for unlocking the rats you're able to buy in the Rat Shop. It's Unlock Method unlocks Rats three tiers down from the

highest tier reached. This is done so you can't immediately buy the latest rat you've reached and immediately merging to a higher tier.

Currency Manger

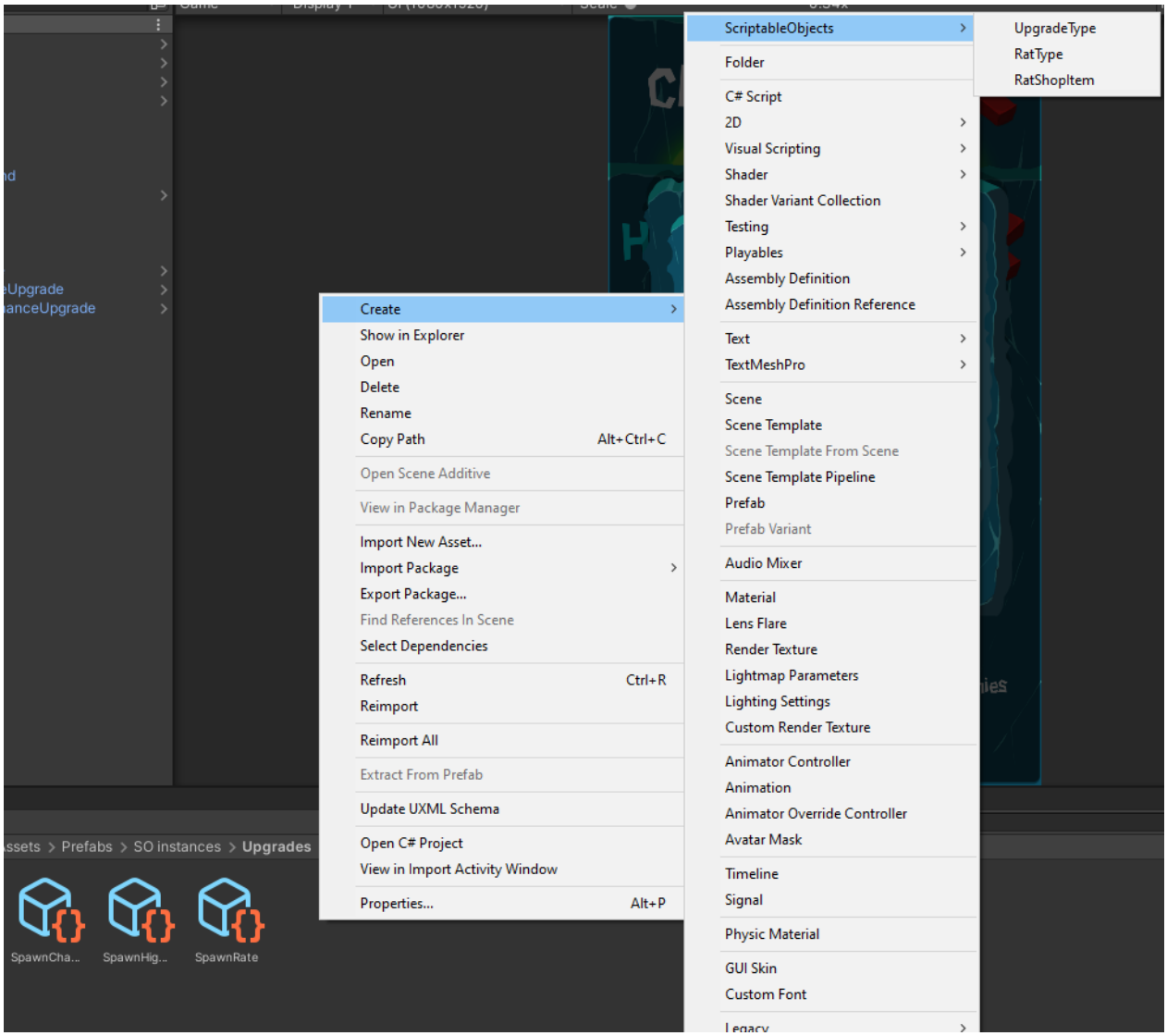
The currency manager keeps track of how much Cheese the player has and adding and deducting Cheese from the total amount. It does this through the usage of the methods DeductRatPower and AddRatPower both these methods are static so can easily be called by any object. For example, DeductRatPower is used by the Shop Item instead of passing a direct reference or creatin the method can now be easily called through the class.

AddRatPower is subscribed to the On Cheese Generated event.

Scriptable Objects

Scriptable Objects are scripts that you can easily create a new instance of and don't have to be added to a scene. Scriptable Objects are great for filling content since you can use them as a blueprint and fill them with the appropriate data. This makes the game also very scalable since it is easy to add new features like upgrades or different types of rats.

To create a new instance right click in the project window go to new -> scriptable objects and pick one. The instances are placed in their respective folders within the prefabs folder.



Can Be Bought

Can Be Bought provides all data that is required by Shop Item. Can Be Bought is an abstract class. HasBeenBought is the only abstract method and must contain the logic to check if an item has successfully been bought and then up the amount of TimesBought.

BuyLimit sets a buy limit if this is set to 0 the limit is infinite.

BaseCost and IncrementCostFactor are used to calculate costs. The calculation of the cost is done by Shop Item.

```
Frequently called  TortleTurtle
public int TimesBought { get => timesBought; set => timesBought = value; }
TortleTurtle
public int BuyLimit
{
    Frequently called
    get => buyLimit;
}
TortleTurtle
public double BaseCost { get => baseCost; }
TortleTurtle
public double IncrementCostFactor { get => incrementCostFactor; }
2 overrides  TortleTurtle
public abstract bool HasBeenBought(); //returns true if success, false if failed.
```

Rat Shop Item

Rat Shop Item inherits from Can Be Bought. Its HasBeenBought methods checks if buying a rat doesn't make the player go over the maximum number of Rats, if not it will tell the Rat Manager to spawn a rat.

```
TortleTurtle
public override bool HasBeenBought()
{
    var ratManager = RatManager.Instance;
    if (ratManager.SpawnedRats.Count >= ratManager.MaxRats) return false;
    timesBought++;
    ratManager.SpawnRat(type, tier);
    return true;
}
```

UpgradeType

Also inherits from Can Be Bought. Its HasBeenBought only checks if the player is not going over the BuyLimit. Also, it adds the accessor Level, this returns the same number as TimesBought. This has been done to make code more readable since usually one refers to the level of an upgrade instead of how many times it has been bought.

Rat Type

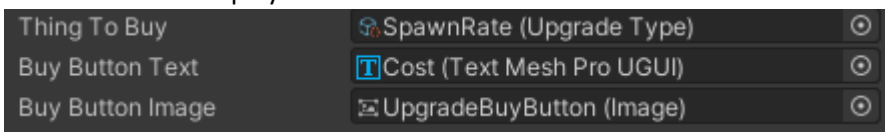
Rat Type provides the information the Rat game object needs. It contains three fields:

- BaseCheesePerSecond
    - Used to calculate how much cheese a rat generates.
- MaxTiers
    - The highest tier a rat can become.
- RatSpriteAtlas
    - Contains the sprites for the corresponding tier of rat.

UI

Shop Item

Shop Item handles the buying of a Can Be Bought instance, calculating its cost. It also displays information to the player. It has three fields that need to be filled.



Thing To Buy is an instance of a Scriptable Object that inherits from Can Be Bought.

The Buy method performs checks using AbleToBuy method and the HasBeenBought method from the Thing To Buy. After performing the checks, it will deduct Cheese from the player.

AbleToBuy looks at the BuyLimit of the Thing To Buy and compares the cost to current amount of cheese the player has.

```
//Returns true if able to buy.
Frequently called  TortleTurtle *
private bool AbleToBuy()
{
    //Check if not going over the buy limit.
    if (thingToBuy.BuyLimit != 0)
    {
        if (thingToBuy.TimesBought + 1 > thingToBuy.BuyLimit) return false
    }
    //Check if cost is not higher then current amount of Cheese.
    if (_cost > CurrencyManager.Cheese) return false;

    return true;
}
```

CalucateCost calculates the cost using a formula. This is handy since this way we don't have to manually set a cost. To calculate the cost, we use TimesBought, BaseCost and IncrementCost factor from the Thing To Buy.

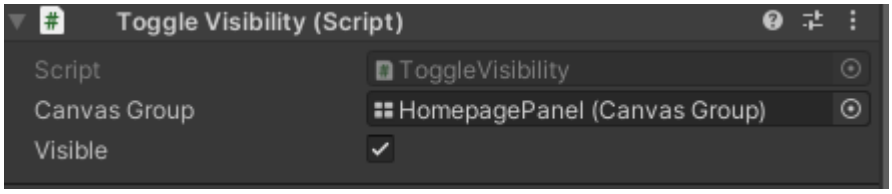
```
_cost = Math.Round(thingToBuy.BaseCost * Math.Pow(thingToBuy.IncrementCostFactor, thingToBuy.TimesBought + 1));
```

The Increment factor is the most impactful attribute. Let's set our base cost to 100, if the Increment factor is 2 the cost will double with every purchase; 100, 200, 400, 800, etc. If the Increment factor is 3 it will triple; 100, 300, 900, 2700, etc.

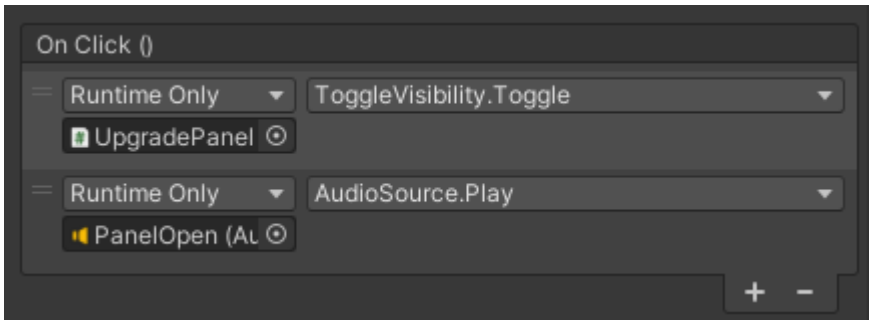
Toggle Visibility

Instead of using SetActive() which disables scripts we use a custom script to toggle the visibility of screens using opacity so scripts can still run.

To use this script the game object you are placing it on must have a Canvas Group component. If something must be visible at start up, make sure the Visible box is checked in the inspector.



You can set the visibility by calling the Toggle() method.



Format Number

A static class that you can use to format numbers to use the units K (Kilo) and M (Mega). For example, if you would pass the number “10000” the result would be “10K”.

Currently it only has the FormatDouble method.

Usage example:

```
_tmpText.text = FormatNumber.FormatDouble(CurrencyManager.Cheese);
```

Display classes

They do what they say on the tin. These classes must be placed on a game object with a Text Mesh Pro Text component.

- ShowCheesePerMinute
  - Displays TotalCheesePerMinute from the statistics manager.
- ShowTotalCurrencyAmount
  - Displays Cheese from the Currency Manager.
- Typewriter
  - Gives text a typewriter effect.

Player Controls

Drag Rats

In this script, the system looks if the screen gets clicked (or touched with a finger). Then it looks if it clicks/touches on a rat's collider. When this is done, it is indicated that this rat is currently being dragged. If the rat is dragged , the next function causes the rat to change position relative to where the mouse or finger is. If the mouse or finger is released from the screen it will drop the rat on that location and says the rat isn't currently being dragged anymore.
If you drag a rat over another rat and drop it, it will compare if the rat you were holding and the rat you are dropping it on are from the same tier. If this is the case it will evolve the rat using a function in the Rat.cs script. Also deleting 1 of the 2 rats from the game which happens in the RatManager.cs script. It also checks if 1 of the rats is shiny, if so it gives you a 50% change to get an evolved shiny.

```
private void CompareRats(Rat thisRat, Rat otherRat)
{
    if (thisRat.tier != otherRat.tier) return;
    //pick a type based on a 50/50 chance.
    if (Random.Range(1, 10) > 5)
    {
        thisRat.type = otherRat.type;
    }
    thisRat.Evolve();
    GetRatManager().RemoveRat(otherRat);
    EventManager.OnRatMerge?.Invoke(thisRat.tier);
}
```

User Input

Mouse and Touch both have different input events which are triggered. So, figuring out if a mouse button has gone down or a finger just touched screen requires its own logic. Instead of writing this into the checks that Drag Rats makes its better for readability to keep these in separate classes. To be able to communicate with these classes an abstract class has been made called User Input.

```
Frequently called  2 overrides  TortleTurtle
public abstract bool Down();
Frequently called  2 overrides  TortleTurtle
public abstract bool Up();
Frequently called  2 overrides  TortleTurtle
public abstract bool Pressed();

Frequently called  2 overrides  TortleTurtle
public abstract RaycastHit2D GetHit();
Frequently called  2 overrides  TortleTurtle
public abstract Vector3 GetPosition();
```

Below an example of how a Down event looks for Mouse Input

```
public class MouseUserInput : UserInput
{
    Frequently called  TortleTurtle *
    public override bool Down()
    {
        return Input.GetMouseButtonDown(0);
    }
```

Below an example of how a Down event looks for Touch Input.

```
public override bool Down()
{
    if (Input.touchCount <= 0) return false;
    var touch = Input.GetTouch( index: 0);
    return touch.phase == TouchPhase.Began;
}
```

Rat

The rat class needs to have a RatType instance assigned.

The SetRat method is one of the more important methods. This sets the sprite and collider and calls the CalculateCheesePerSecond method. SetRat must be called after load and every time the rats tier changes.

```
//sets the rats sprite and collider size.
Frequently called  TortleTurtle +1 *
public void SetRat()
{
    //get the sprite from the sprite atlas
    _spriteRenderer.sprite = type.RatSpritesAtlas.GetSprite( name: tier.ToString());
    //set collider size to the size of the sprite.
    _boxCollider.size = (Vector2)_spriteRenderer.sprite.bounds.size;
    CalculateCheesePerSecond();
}
```

CalculateCheesePerSecond uses the rats tier and the BaseCheesePerSecond provided by RatType. The amount of CheesePerSecond goes up exponentially per tier + a little extra.

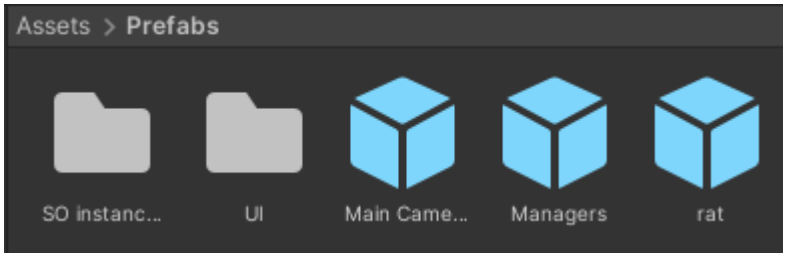
```
CheesePerSecond = (Math.Pow(type.BaseCheesePerSecond, tier) + 0.5 * tier);
```

For example, if BaseCheesePerSecond is set to 2 and the rat would be tier 5 the following calculation would be made: 2^5 + 0.5 * 5 = 34.5.

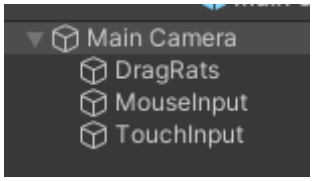
The GenerateCheese method contains an interval that fires the On Cheese Generated event every 10 seconds.

```
var cheeseGenerated :double = CheesePerSecond * 10; //times 10 because timer is every 10 seconds.
//Fire On cheese event and pass amount of cheese generated.
EventManager.OnCheeseGenerated?.Invoke(cheeseGenerated);
```

Prefabs

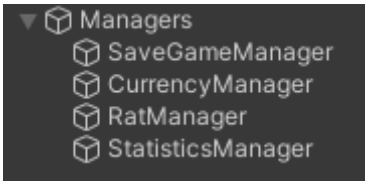


Main Camera



Besides containing the main camera, the prefab also contains an object with the Drag Rats script and objects with the User Input scripts. The reason for this is because those scripts make use of the main camera for different functionalities such as Ray Casting. They've been put there to signify the importance of the camera to these scripts.

Managers



The manager prefab contains all managers. They've been grouped like this since some managers need references to another manager. By putting them into a prefab it is certain they'll always have a reference. An added benefit is that adding new managers is easy since one can just add a new manager to the prefab and it will automatically be added in every scene Since the Index Unlock Manager and the Rat Shop Manager are only responsible for items on one screen they've been placed on their respective panels.

Rat

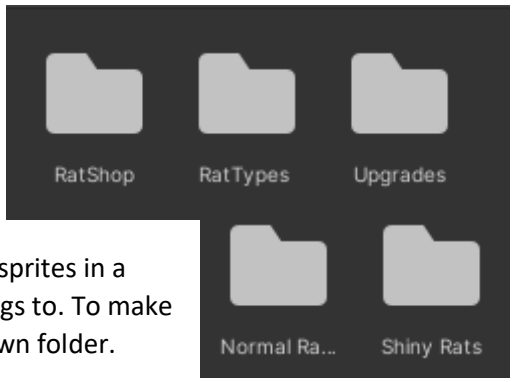
This prefab contains the Rat script. This prefab is used by the Rat Manager to spawn new rats.

UI

Inside the UI folder you'll find prefabs of often used elements in the UI. These allow for quickly making a new screen. If you want to make a change to an element in every screen you only need to adjust the prefab.

The UICanvas prefab is special. It contains the whole UI. This is done for implementing UI changes. In the UI scene changes are made to the UI. Once these are approved those changes get pushed to the original prefab.

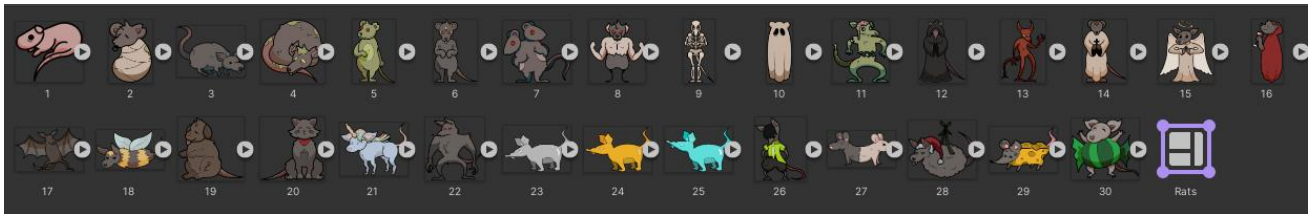
SO Instances



In this folder all instances of Scriptable Objects are placed in their respective folder. This makes it easier to find the instance specific instances.

Creating a sprite atlas

The rat script makes use of a sprite atlas to get the right sprite corresponding to its tier. To make this possible, the sprites in a sprite atlas must be named the number of the tier it belongs to. To make this possible, the sprites of a type of rat must be in their own folder.



Information on creating a Sprite Atlas can be found here.