# DAY 01: 16/01/2022

Developed a problem description for each class, a class diagram, and created a github for the project. Next step: Write pseudocode.

# DAY 02 17/01/2022

Started working on pseudocode. Good progress.

Re-structured the Simulation class and added some ideas for generating limbs based on the parent creatures.

Jumping power of the creature is a random number between the creature’s parents’ jumping power.

For each limb, the creature has higher probability(80%) of generating the same limb type as the parents if they both have the same limb type, otherwise the limb generation is random.

Other creature characteristics will be calculated smiarly

All of this is not final and I will have to perform tests and check if this approach succeeds in generating successful creatures.

Other limb, wheel and leg characteristics will be simulated similarly.

# DAY 03:

Pseudocode is good. Review it and see if anymore progress is needed. If not, START UP UNITY!

Started working a bit more on pseudocde. Mind is unclear so I will stop here, I believe most of my code is straight bs so I’ll do it with a clear head.

# DAY04(02/02/23) Continued pseudocode.

Completed pseudocde for back end: Leg and Wheel class completed. Also fixed up Simulatin class

Added mutations in the Creature class where the creature has a 20% chance of not following the parents’ path.(This is simply done using Math.rand and the value can easily be changed). Decided hat it is not necessary to have this on the lower classes as well

Next time: Start working on Unity. Must have Stuart something t show by next week!!!

# DAY05(06/02/23)

Palayed around with Unity a bit. Made a simple platform and saw how wheels can work. But didn’t get much done tbh.

Next time:

* Find ou how creatures can actually move
* Backup my shit

NEXT NEXT TIME:

* Start generating creatures

Unity to do list:

To do:

- Make very simple environment DONE

- Find out how to simply generate a single creature. Maybe using scripts too DONE

- Find out how to generate multiple creatures that dont interact...

- Make creatures actually move( legs move, wheels move)

- Find out how to pass on data from generation go generation(scene to scene?)

-probaly like you would do on eg a mario game

- How do we follow the best creture?

# Day06(12/02/2023)

Lots have changed. Have figured out how to implemented movement in Unity. With cars its actually not that bad. There is a txt file for some notes.

I will now focus souly on cars, and will need to create a new environment for testing. Genetical aglrotihms will:

* Change speed? But MUST have a max
* Change wheel size? But must have a max
* Change wheel location? But need to figure out some values, eg transform and wheel joint connected anchor
* Change wheel number? But will have a max

Next time:

* Create a simple environment
* Figure out transform and stuff
* Generate a SINGLE creature with random values

# DAY07 (14-02-2023)

Did simple research ons cirpt. It’s actually not that hard to generate things. Just need to figure out the sprite editor. Not much done cuz no time( Airport)

# DAY08(16-02-2023)

Figured out how to summon the car body with a lot of assets! Next up is summoning the wheels. I NEED to do this before I leave Amsterdam. Also, focus on your report that it took you some time to get around Unity

# DAY09(22-02-2023)

Progress on the creature generator. I can generate a fully functioninig creature.

Now need to add in a few things to change the size, and also make sure the creature generates properly.

General to-dos:

* fully generate a car
* Generate a car with selected assets
  + generate with random assets. But need to come up with restrictions
  + like where in the body the wheel can generate, max size of the wheel etc.
* -Build the simulation
  + Follow the best car
  + Cars don’t interact
  + See which car made it further
  + Pass data from generation to generation

# Day10(13-03-2023)

Creatures don’t collide. Add creature layer, then set up in the project settings that creatures with the creature layer don’t inetrect.

1. Create a new layer for your creatures. In the Unity Editor, go to Edit -> Project Settings -> Tags and Layers, and click the "Add Layer" button. Name the new layer "Creature".
2. Assign the "Creature" layer to your creature objects. Select a creature object in the Unity Editor, and in the Inspector window, set the "Layer" property to "Creature".
3. Modify the physics collision matrix to disable collisions between objects in the "Creature" layer. In the Unity Editor, go to Edit -> Project Settings -> Physics 2D. Find the row for "Creature" and uncheck all of the boxes in the "Collide With" column, except for the "Default" layer.

Next to do:

Creatures generated all at once( I think they are)

Follow creatures

Assign them different speeds

# DAY 11(15/03/2023

* Fixed issue where creatures spawned wonky( have great detail abnd videos of this, use it to demonstrate your debugging)
* Added scripts in
* Generated creatures with random speeds and wheel sizes
* Gave slight colors to creatures

Tommorow’s to do:

* Follow best creature
* Count how far a creature has made it
* Good arena

Next Day’s to do:

* Genetic Algoriths

Next Next day,

* Make creatures a bit more complicated. Eg maybe give them different body shapes? Wheel typeS? Legs? I have ad oc somewhere with ideas go find it.

Fixing issue: Creature spawn like this lol

Diagram

Description automatically generated

After checking code, it seemed that the wheels generating pose a problem. If only the body itself generates then its fine.

A picture containing diagram

Description automatically generated

Made changes to code. On the wheel generator, the wheelJoint2D had an issue when adding the conenctedAnchor. The vector we were creating would move the wheel at the wrong spot, and that isbecause we had wrong x value. It is fixed now and we can move on

I recorded some videos on my phone about this and they are in a folder. These videos are to show the debugging process

Next to-do:

* Car Movement Script DONE
* Pass random values for speed DONE
* Pass random values for speed size DONE
* Platform DON’T NEED YET
* Tracker for which car is the best

Next next

* Change the creature so that it has more places to generate wheels
* Have it so it passes from generation to generation
* Remember I wrote some things down for this somewhere so. Like what variables we can change on the wheel.
* Make it look a bit sexier

Added script in the car so it moves vroom vroom

**A big part of this assignment was figuring out how Unity works.**

**Used scenes to save progress**

Okay…. So let me save.

Now we will create a script following the best creature.

# Day12 16/03/2023

Renamed creature Generator to Creature

Lmao actually didn’t do much.

Need to restructure my code. Use the pseudocode and think how ti intergrate it into unity

# DAY13 (17-03-2023)

Combination day! Big day! Acutal unity with pseudocode!

Realized need to do some serious restructuring. Especially with accessing variables and class structure etc. Need to think and structure

Focused on implementing the actual camera following.

Couldn’t implement camera following. Theres something wrong with my Unity simulation I think.

## Wheel Class:

Added generate wheel code

Had to change constructors, remember this!

## Creature Class:

Decided to have mutations happen at the simulation class. The higher the better

Creature class now has to return shit

Realsied need to do some serious resturcuring. Especially with accessing variables and class structure etc. Need to think and structure

# DAY14(22/03/2023)

Lots of progress

Managed to establish classes and objects in my simulkation.

Fixed the camera issue I had, I need to assign the creature BODY not the object to the variable for it to work.

Struggling with automating this in my simulation scrip tbut I think its just a simple mistake and my mind is boggled.

To do:

* Assign the camera variable automatically
* Figure out why numOfCreatures don’t work
* Have it go from generation to generat

Future to-do:

* New arena
* New variables to consider
* Actual experiment\

**Chatpt**: In Unity its Debug.log not Console.writeLine like it is in C#

Notes when coding:

I need an array of type wheel

-Not game object because it also needs attributes like the radius

-So constructor must return a type wheel

-what should the wheelGenerator return>?

-Asked chat gpt

OKAY SO

The constructor doesnt NEED to reutnr anything, nor does the wheelGenerator

In the wheelArray in the Creature Class we just call the wheel cosntructor

We use the private varibles of each class for shit

IT'S DONE

# Day15(23/03/2023)

Tried fixing the getBestCreautre but did not manage. TBF my debugging was braindead and not very critical. Will do it again tomorrow. I did manage to fix the camera thing though, just had to assign the MainCamera gemeObject through the Unity editor interface:  
A picture containing text, screenshot, monitor, black

Description automatically generated

Tomorrow’s Todo: Finally fix this

Get from generation to generation

# Day16(24/03/2023)

Managed to get a scene restarter working! Much help from ChatGpt telling me how to do things. I didn’t need to spend hours googling stuff

## CHATGPT SCENE RESTARTER HELPER(Click me!)

In the unity game engine, how do i restart a scene while passing values to the restarted scene?

To restart a scene in Unity while passing values to the restarted scene, you can use the following steps:

1. Create a new script that will handle the scene restart and value passing. For this example, let's call it **SceneRestart**.
2. In the **SceneRestart** script, create a method that will handle the scene restart. This method should load the current scene again and pass the necessary values to it. For example:

csharpCopy code

public void RestartScene(string value1, int value2) { SceneManager.LoadScene(SceneManager.GetActiveScene().name); SceneManager.sceneLoaded += OnSceneLoaded; PlayerPrefs.SetString("value1", value1); PlayerPrefs.SetInt("value2", value2); }

In this example, we are using **SceneManager.LoadScene** to reload the current scene. We are also using **PlayerPrefs** to store the values we want to pass to the restarted scene. Note that we are using the **SceneManager.sceneLoaded** event to ensure that the values are passed to the new scene after it has finished loading.

1. Create a new script that will handle the values in the restarted scene. For this example, let's call it **SceneValues**.
2. In the **SceneValues** script, create a method that will retrieve the values passed from the **SceneRestart** script. For example:

csharpCopy code

void Start() { string value1 = PlayerPrefs.GetString("value1"); int value2 = PlayerPrefs.GetInt("value2"); // Use the values as needed }

In this example, we are using **PlayerPrefs** to retrieve the values passed from the **SceneRestart** script.

1. Add both the **SceneRestart** and **SceneValues** scripts to an object in the scene.
2. In your game, call the **RestartScene** method from the **SceneRestart** script when you want to restart the scene and pass values to it. For example:

scssCopy code

sceneRestartObject.GetComponent<SceneRestart>().RestartScene("value1", 42);

In this example, we are passing the string **"value1"** and the integer **42** to the restarted scene.

That's it! Now your scene will restart and the necessary values will be passed to it.

Tell me more about ' SceneManager.sceneLoaded'

**SceneManager.sceneLoaded** is an event that is triggered whenever a new scene finishes loading. You can use this event to perform actions after a scene has finished loading, such as initializing objects or passing values between scenes.

Here's an example of how you can use **SceneManager.sceneLoaded** to pass values between scenes:

csharpCopy code

void Start() { SceneManager.sceneLoaded += OnSceneLoaded; } void OnSceneLoaded(Scene scene, LoadSceneMode mode) { if (scene.name == "MyScene") { string myValue = PlayerPrefs.GetString("MyValue"); Debug.Log("MyValue in MyScene is: " + myValue); } }

In this example, we are subscribing to the **SceneManager.sceneLoaded** event in the **Start** method. We are then defining an **OnSceneLoaded** method that will be called whenever a new scene finishes loading. Inside the **OnSceneLoaded** method, we are checking if the loaded scene has the name "MyScene". If it does, we are retrieving the value of **MyValue** that was stored in **PlayerPrefs** and logging it to the console.

Note that when you use **SceneManager.LoadScene** to load a scene, the **SceneManager.sceneLoaded** event will be triggered for the new scene after it has finished loading. This means that you can use this event to pass values from the current scene to the new scene, as long as you store those values somewhere persistent, like **PlayerPrefs**.

What is PlayerPrefs? in simple terms

**PlayerPrefs** is a simple way to store and retrieve data in Unity games. It is essentially a key-value store, where you can associate a value (such as a string or an integer) with a key (a string).

The values stored using **PlayerPrefs** persist even after the game has been closed, which makes it a convenient way to store game settings or player preferences.

Here's an example of how to use **PlayerPrefs** to store and retrieve a string:

cCopy code

// Store a string in PlayerPrefs string myString = "Hello, world!"; PlayerPrefs.SetString("MyString", myString); // Retrieve the string from PlayerPrefs string retrievedString = PlayerPrefs.GetString("MyString"); Debug.Log("The retrieved string is: " + retrievedString);

In this example, we are storing the string "Hello, world!" in **PlayerPrefs** with the key "MyString". We can then retrieve this string later using **PlayerPrefs.GetString("MyString")**. When we log the retrieved string to the console using **Debug.Log**, we should see "The retrieved string is: Hello, world!".

**PlayerPrefs** can be used to store and retrieve other types of data as well, such as integers and floats. To store an integer, you can use **PlayerPrefs.SetInt("MyInt", myInt)** and to retrieve it, you can use **PlayerPrefs.GetInt("MyInt")**. To store a float, you can use **PlayerPrefs.SetFloat("MyFloat", myFloat)** and to retrieve it, you can use **PlayerPrefs.GetFloat("MyFloat")**.

Tell me more about this line of code: SceneManager.sceneLoaded += OnSceneLoaded;

The line of code **SceneManager.sceneLoaded += OnSceneLoaded;** is subscribing to the **SceneManager.sceneLoaded** event, which is triggered whenever a new scene is loaded in Unity.

The **+=** operator is used to add a new event handler method to the event. In this case, the event handler method is **OnSceneLoaded**, which is a method defined elsewhere in the script.

Here's an example of how you might define the **OnSceneLoaded** method:

arduinoCopy code

void OnSceneLoaded(Scene scene, LoadSceneMode mode) { // Do something when a new scene is loaded }

This method takes two parameters: a **Scene** object representing the scene that was loaded, and a **LoadSceneMode** object representing the mode in which the scene was loaded (e.g. Single, Additive, etc.). You can use these parameters to perform actions when a new scene is loaded, such as initializing objects or passing values between scenes.

By subscribing to the **SceneManager.sceneLoaded** event with the **+=** operator and passing in the **OnSceneLoaded** method, you are telling Unity to call the **OnSceneLoaded** method every time a new scene is loaded.

It's worth noting that you should also unsubscribe from events when you are done with them, to avoid any potential memory leaks or unexpected behavior. You can do this with the **-=** operator, like so:

Copy code

SceneManager.sceneLoaded -= OnSceneLoaded;

This removes the **OnSceneLoaded** method as an event handler for the **SceneManager.sceneLoaded** event.

how else could i write that line of code, in simpler terms

The line of code **SceneManager.sceneLoaded += OnSceneLoaded;** means that when a new scene is loaded, the method **OnSceneLoaded** will be called automatically.

Think of it as registering a method to be called whenever a certain event (in this case, the loading of a scene) occurs. The **+=** operator adds the **OnSceneLoaded** method as an event handler to the **SceneManager.sceneLoaded** event, so that whenever a new scene is loaded, the **OnSceneLoaded** method will be called.

So, in simpler terms, this line of code is saying "When a new scene is loaded, call the **OnSceneLoaded** method automatically".

But how is it called, with no parameters?

When you use the **+=** operator to add a method to an event, you are not actually calling the method yourself. Instead, you are telling Unity to call that method automatically when the event is triggered.

In this case, **SceneManager.sceneLoaded** is an event that is triggered whenever a new scene is loaded. When that event is triggered, Unity will automatically call the **OnSceneLoaded** method that was added as an event handler using the **+=** operator.

The **OnSceneLoaded** method takes two parameters (**Scene** and **LoadSceneMode**), but you don't need to worry about passing those parameters yourself. Unity will automatically pass the appropriate parameters to the **OnSceneLoaded** method when it is called as an event handler.

So even though the **SceneManager.sceneLoaded** event doesn't explicitly pass any parameters to the **OnSceneLoaded** method, the method still receives the necessary parameters when it is called automatically by Unity.

Do i have any control of these parameters?

Yes, you can control the parameters that are passed to the **OnSceneLoaded** method when it is called as an event handler for the **SceneManager.sceneLoaded** event.

The **SceneManager.sceneLoaded** event is defined to pass two parameters: a **Scene** object representing the scene that was loaded, and a **LoadSceneMode** object representing the mode in which the scene was loaded (e.g. Single, Additive, etc.).

When Unity calls the **OnSceneLoaded** method as an event handler for the **SceneManager.sceneLoaded** event, it automatically passes those two parameters to the method.

However, you can also define your own method with the same signature as the **OnSceneLoaded** method (i.e. taking a **Scene** object and a **LoadSceneMode** object as parameters), and then pass that method as an argument to the **SceneManager.LoadScene** method to control the parameters that are passed to the **OnSceneLoaded** method.

Here's an example of how you might define your own method to control the parameters passed to **OnSceneLoaded**:

csharpCopy code

void MySceneLoadedMethod(Scene scene, LoadSceneMode mode) { // Do something with the scene and mode parameters } void LoadNewScene() { SceneManager.sceneLoaded += MySceneLoadedMethod; SceneManager.LoadScene("MyNewScene"); }

In this example, we have defined a new method called **MySceneLoadedMethod** that takes the same two parameters as **OnSceneLoaded**. We have also subscribed **MySceneLoadedMethod** to the **SceneManager.sceneLoaded** event using the **+=** operator.

Then, when we call **SceneManager.LoadScene("MyNewScene")**, Unity will load the "MyNewScene" scene and automatically call the **MySceneLoadedMethod** method as an event handler for the **SceneManager.sceneLoaded** event. This means that the **MySceneLoadedMethod** method will receive the **Scene** object and **LoadSceneMode** object for the new scene, which you can then use to perform whatever actions you need to.

I get this error for my code: Assets\Scenes\11Simulation\RestartScene.cs(10,9): error CS0103: The name 'SceneManager' does not exist in the current context using System.Collections; using System.Collections.Generic; using UnityEngine; public class SceneRestart : MonoBehaviour { public void RestartScene(string value1, int value2){ // this function restarts the current scene. It will be used by the simulation class // to restart the scene/simulation when deemed neccessary SceneManager.LoadScene(ScenEmANAGER.GetActiveScene().name);// Gets the currents scene and restarts it SceneManager.sceneLoaded += OnSceneLoaded; // += means that OnSceneLoaded will be called automatically too when a new scene is loaded // note that OnSceneLoaded will be handles by unity and i do not need to give any parameters // SceneManager.sceneLoaded is an event triggered when a new scene finehses loaded // And you use this event to perform actions such as intilaisjg objects or passing values //between scenes. In our case its the Generation number and the Best Creatures PlayerPrefs.SetString("value1", value1); PlayerPrefs.SetInt("value2", value2); // PlayerPrefs is a simple way to retreive data in Unity games. Its a key-value store // These values stay even after the game is closed, so you can store game settings or player prefernece } }

The error message "The name 'SceneManager' does not exist in the current context" means that the **SceneManager** class is not recognized by the compiler because it has not been imported or referenced in your script.

To fix this error, you need to add the following line at the top of your script to import the **UnityEngine.SceneManagement** namespace, which contains the **SceneManager** class:

arduinoCopy code

using UnityEngine.SceneManagement;

Once you have added this line, the **SceneManager** class should be recognized by the compiler and you should be able to use it in your script without any issues.

Here's the updated code with the import statement added:

csharpCopy code

using System.Collections; using System.Collections.Generic; using UnityEngine; using UnityEngine.SceneManagement; public class SceneRestart : MonoBehaviour { public void RestartScene(string value1, int value2){ // this function restarts the current scene. It will be used by the simulation class // to restart the scene/simulation when deemed neccessary SceneManager.LoadScene(SceneManager.GetActiveScene().name);// Gets the currents scene and restarts it SceneManager.sceneLoaded += OnSceneLoaded; // += means that OnSceneLoaded will be called automatically too when a new scene is loaded // note that OnSceneLoaded will be handles by unity and i do not need to give any parameters // SceneManager.sceneLoaded is an event triggered when a new scene finehses loaded // And you use this event to perform actions such as intilaisjg objects or passing values //between scenes. In our case its the Generation number and the Best Creatures PlayerPrefs.SetString("value1", value1); PlayerPrefs.SetInt("value2", value2); // PlayerPrefs is a simple way to retreive data in Unity games. Its a key-value store // These values stay even after the game is closed, so you can store game settings or player prefernece } }

I hope this helps!

Assets\Scenes\11Simulation\SceneRestart.cs(12,37): error CS0103: The name 'OnSceneLoaded' does not exist in the current context using System.Collections; using System.Collections.Generic; using UnityEngine; using UnityEngine.SceneManagement; public class SceneRestart : MonoBehaviour { public void RestartScene(string value1, int value2){ // this function restarts the current scene. It will be used by the simulation class // to restart the scene/simulation when deemed neccessary SceneManager.LoadScene(SceneManager.GetActiveScene().name);// Gets the currents scene and restarts it SceneManager.sceneLoaded += OnSceneLoaded; // += means that OnSceneLoaded will be called automatically too when a new scene is loaded // note that OnSceneLoaded will be handles by unity and i do not need to give any parameters // SceneManager.sceneLoaded is an event triggered when a new scene finehses loaded // And you use this event to perform actions such as intilaisjg objects or passing values //between scenes. In our case its the Generation number and the Best Creatures PlayerPrefs.SetString("value1", value1); PlayerPrefs.SetInt("value2", value2); // PlayerPrefs is a simple way to retreive data in Unity games. Its a key-value store // These values stay even after the game is closed, so you can store game settings or player prefernece } }

The error message "The name 'OnSceneLoaded' does not exist in the current context" means that there is no method called **OnSceneLoaded** defined in your **SceneRestart** class.

To fix this error, you need to define a method called **OnSceneLoaded** in your **SceneRestart** class. This method will be called automatically by Unity whenever a new scene is loaded, because you have subscribed to the **SceneManager.sceneLoaded** event in your **RestartScene** method.

Here's an example of how you could define the **OnSceneLoaded** method:

cCopy code

void OnSceneLoaded(Scene scene, LoadSceneMode mode) { // This method will be called automatically by Unity whenever a new scene is loaded // You can use it to perform any actions that need to be taken after a scene has finished loading // In our case, we will use it to pass the values between scenes string value1 = PlayerPrefs.GetString("value1"); int value2 = PlayerPrefs.GetInt("value2"); Debug.Log("Loaded scene: " + scene.name); Debug.Log("Value 1: " + value1); Debug.Log("Value 2: " + value2); }

Note that this method should be defined within your **SceneRestart** class, but outside of your **RestartScene** method. Once you have defined this method, you should no longer get the "The name 'OnSceneLoaded' does not exist in the current context" error.

Here's the updated code with the **OnSceneLoaded** method added:

csharpCopy code

using System.Collections; using System.Collections.Generic; using UnityEngine; using UnityEngine.SceneManagement; public class SceneRestart : MonoBehaviour { void OnSceneLoaded(Scene scene, LoadSceneMode mode) { // This method will be called automatically by Unity whenever a new scene is loaded // You can use it to perform any actions that need to be taken after a scene has finished loading // In our case, we will use it to pass the values between scenes string value1 = PlayerPrefs.GetString("value1"); int value2 = PlayerPrefs.GetInt("value2"); Debug.Log("Loaded scene: " + scene.name); Debug.Log("Value 1: " + value1); Debug.Log("Value 2: " + value2); } public void RestartScene(string value1, int value2){ // this function restarts the current scene. It will be used by the simulation class // to restart the scene/simulation when deemed neccessary SceneManager.LoadScene(SceneManager.GetActiveScene().name);// Gets the currents scene and restarts it SceneManager.sceneLoaded += OnSceneLoaded; // += means that OnSceneLoaded will be called automatically too when a new scene is loaded // note that OnSceneLoaded will be handles by unity and i do not need to give any parameters // SceneManager.sceneLoaded is an event triggered when a new scene finehses loaded // And you use this event to perform actions such as intilaisjg objects or passing values //between scenes. In our case its the Generation number and the Best Creatures PlayerPrefs.SetString("value1", value1); PlayerPrefs.SetInt("value2", value2); // PlayerPrefs is a simple way to retreive data in Unity games. Its a key-value store // These values stay even after the game is closed, so you can store game settings or player prefernece } }

I hope this helps!

In unity, how can i start a timer?

In Unity, you can start a timer using the **Time** class. Here's an example:

csharpCopy code

using UnityEngine; public class Timer : MonoBehaviour { float timeElapsed; void Update() { timeElapsed += Time.deltaTime; Debug.Log("Time elapsed: " + timeElapsed); } }

This script starts a timer in the **Update()** method by incrementing a **timeElapsed** variable by the **Time.deltaTime** value every frame. **Time.deltaTime** represents the amount of time that has passed since the last frame was rendered, so this value will be very small (e.g. around 0.016 seconds for a 60 FPS game).

In this example, the timer is logged to the console using **Debug.Log()**, but you could also use the timer value to update a UI element or perform other actions in your game.

Is there a built in funciton to use in scripts?

Unity does not have a built-in timer function that you can use directly in your scripts. However, as I mentioned in my previous answer, you can use the **Time** class to implement your own timer.

Alternatively, you could use a third-party plugin or asset from the Unity Asset Store that provides timer functionality. There are several options available, including free and paid assets, that offer features like countdown timers, stopwatches, and more.

round the time to 2 decibels

To round a time value to 2 decimal places in C# (which is the language used in Unity scripts), you can use the **System.Math.Round()** method. Here's an example:

csharpCopy code

float time = 3.14159f; float roundedTime = (float)System.Math.Round(time, 2);

In this example, **time** is the original time value (in seconds), and **roundedTime** is the time rounded to 2 decimal places. The second argument of the **Round()** method specifies the number of decimal places to round to.

Note that **System.Math.Round()** returns a **double** value, so you need to cast it back to a **float** if you want to store the rounded value in a **float** variable.

how do i call a funciton from another scipt in my script

To call a function from another script in your Unity project, you need to follow these steps:

1. Attach the script containing the function you want to call to a GameObject in your scene. You can do this by dragging the script onto the GameObject in the Unity Editor, or by clicking the "Add Component" button in the Inspector window and selecting the script from the list.
2. In your script, create an instance of the class that contains the function you want to call. You can do this by declaring a variable of the class type and using the **new** keyword to create a new instance:

csharpCopy code

OtherScript otherScript = new OtherScript();

Replace **OtherScript** with the name of the class that contains the function you want to call.

1. Call the function on the instance of the class you created in step 2. You can do this using the dot notation:

csharpCopy code

otherScript.FunctionName();

Replace **FunctionName** with the name of the function you want to call.

Here's an example that puts these steps together:

csharpCopy code

using UnityEngine; public class MyScript : MonoBehaviour { void Start() { OtherScript otherScript = new OtherScript(); otherScript.DoSomething(); } }

In this example, **MyScript** is the script that wants to call a function from **OtherScript**. **Start()** is a built-in Unity function that is called when the GameObject that the script is attached to is created in the scene. The **DoSomething()** function is a function defined in **OtherScript**. When the GameObject is created, **Start()** is called, which creates an instance of **OtherScript** and calls its **DoSomething()** function.

what if otherScript has a Start() function, will the script be started twice?

No, the **Start()** function of **otherScript** will not be called twice. When you use **otherScript.FunctionName()** in your script, you are simply calling the **FunctionName()** function of **otherScript** from your script. This does not start the **otherScript** object or its **Start()** function. The **Start()** function is only called when the **otherScript** object is initialized by Unity, which happens only once when the scene starts or the object is instantiated.

does MyScript and otherScript need to be assigned to the same gameObject?

No, **MyScript** and **otherScript** do not need to be assigned to the same **GameObject**. As long as **otherScript** is attached to a **GameObject** in the scene hierarchy and **MyScript** has a reference to that **GameObject** through a public variable or property, you can call **otherScript** functions from **MyScript**.

For example, if **otherScript** is attached to a **GameObject** named **OtherObject**, you can reference it from **MyScript** like this:

csharpCopy code

public class MyScript : MonoBehaviour { public OtherScript otherScript; // reference to OtherScript private void Update() { // Call a function from OtherScript otherScript.FunctionName(); } }

Note that in this example, **OtherScript** is the name of the class that **otherScript** is attached to. You would replace it with the actual name of your script.

NullReferenceException: Object reference not set to an instance of an object Simulation.getBestCreature () (at Assets/Scenes/11Simulation/Simulation.cs:66) Simulation.Update () (at Assets/Scenes/11Simulation/Simulation.cs:44) using System.Collections; using System.Collections.Generic; using UnityEngine; public class Simulation : MonoBehaviour { // ======================================================================================================== // private variables // ======================================================================================================== private int genNum; private int numOfCreatures = 100; private Creature[] creatureArray = new Creature[100];// create an array of type creature and length creatureNum private Creature bestCreature; private Creature secondBestCreature; private Creature currentCreature;// variable that will be used in functions private Creature currentBestCreature;// variable that will be used in functions public CameraFollow cameraFollowScript; public Transform newTarget; private GameObject camera; private float timeElapsed; TimeScript timeScript = new TimeScript();// refrence to the TimeScript // ======================================================================================================== // Start is called before the first frame update // ======================================================================================================== void Start() { // Generation 0 populateCreatureArray(); setCamera(creatureArray[0]); // TEMPorarailly set the new target as the first creature in the array, for testing purposes bestCreature = getBestCreature(); } // Update is called once per frame void Update() { bestCreature = getBestCreature(); // time measurer timeElapsed = timeScript.getTime(); Debug.Log("Time elapsed in Simulation.cs:" + timeElapsed); } // ======================================================================================================== // Getters and setters // ======================================================================================================== void populateCreatureArray(){ // this function is used at the Start() funciton for(int i = 0; i < numOfCreatures; i++){ creatureArray[i] = new Creature(); creatureArray[i].CreatureNoParent(); } } Creature getBestCreature(){ currentBestCreature = creatureArray[0]; Debug.Log(creatureArray[0].getCreatureBody()); return (currentBestCreature); } void setCamera(Creature targetCreature){ newTarget = targetCreature.getCreatureBody().transform; cameraFollowScript.setTarget(newTarget); } } using System.Collections; using System.Collections.Generic; using UnityEngine; public class TimeScript: MonoBehaviour { private float timeElapsed; //This class will be used to measure the time elapsed in a simulation void Start(){ timeElapsed = 0; } // Update is called once per frame void Update() { // time constantly updated timeElapsed += Time.deltaTime; //Debug.Log("Time elapsed: " + System.Math.Round(timeElapsed, 2)); // uncomment if needed for testing purposes } public float getTime(){ return timeElapsed; } }

The issue in your code is that you are trying to create an instance of **TimeScript** using the **new** keyword in the **Simulation** script. This is not the correct way to create an instance of a script in Unity, as all script instances should be attached to a GameObject in the scene.

To fix the issue, you need to attach the **TimeScript** script to a GameObject in your scene and then get a reference to that object in your **Simulation** script. You can do this using the **GameObject.Find()** function, which allows you to find a GameObject in the scene by its name or tag.

Here is an updated version of your **Simulation** script that should work:

csharpCopy code

using System.Collections; using System.Collections.Generic; using UnityEngine; public class Simulation : MonoBehaviour { // ======================================================================================================== // private variables // ======================================================================================================== private int genNum; private int numOfCreatures = 100; private Creature[] creatureArray = new Creature[100];// create an array of type creature and length creatureNum private Creature bestCreature; private Creature secondBestCreature; private Creature currentCreature;// variable that will be used in functions private Creature currentBestCreature;// variable that will be used in functions public CameraFollow cameraFollowScript; public Transform newTarget; private GameObject camera; private float timeElapsed; private TimeScript timeScript; // ======================================================================================================== // Start is called before the first frame update // ======================================================================================================== void Start() { // Generation 0 populateCreatureArray(); setCamera(creatureArray[0]); // Get a reference to the TimeScript attached to the "TimeScript" GameObject timeScript = GameObject.Find("TimeScript").GetComponent<TimeScript>(); bestCreature = getBestCreature(); } // Update is called once per frame void Update() { bestCreature = getBestCreature(); // time measurer timeElapsed = timeScript.getTime(); Debug.Log("Time elapsed in Simulation.cs:" + timeElapsed); } // ======================================================================================================== // Getters and setters // ======================================================================================================== void populateCreatureArray(){ // this function is used at the Start() funciton for(int i = 0; i < numOfCreatures; i++){ creatureArray[i] = new Creature(); creatureArray[i].CreatureNoParent(); } } Creature getBestCreature(){ currentBestCreature = creatureArray[0]; Debug.Log(creatureArray[0].getCreatureBody()); return (currentBestCreature); } void setCamera(Creature targetCreature){ newTarget = targetCreature.getCreatureBody().transform; cameraFollowScript.setTarget(newTarget); } }

Make sure to attach the **TimeScript** script to a GameObject in your scene called "TimeScript" for the updated code to work.

get a random intiger in Unity

You can use the **Random.Range()** function to get a random integer in Unity. This function takes two integer parameters, **min** and **max**, and returns a random integer between **min** (inclusive) and **max** (exclusive).

Here's an example of how to use it to get a random integer between 0 and 9:

csharpCopy code

int randomInt = Random.Range(0, 10); Debug.Log("Random integer: " + randomInt);

This will log a random integer between 0 and 9 to the console.

# A

I also did some testing by pringing things on the console to see if they work, for example testing the timer. Printong the time elapsed on the Simulation.CS class

A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence

Another example of testing is testing to see if the right values are passed. For example, on the SceneRestart.cs scipt:

Text

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

Now… lets try passing some ACTUAL useful values.

**MAKE SURE TO FIX GETSECONDBESTCREATURE**

**MAKE SURE TO FIX dontDestroy PRIVATE PUBLIC ETC**

* I am almost ready with transferring variables, do this.
* Also add interface
* Once that is done start testing and clean up code a bit
* Then add in the different things for the actual experiment, and environment.