1. Argo Metaverse Unity Project

Welcome to the Argo Metaverse Unity project!

The Argo Metaverse v1.0 here referenced is the final version of the first development stage of the Argo Metaverse, an open-world metaverse hub for the Argo community. The Argo Project is an open source tool for Kubernetes to run workflows, manage clusters, and do GitOps right, but the Argo community activities encompass more than just writing software. With the Argo Metaverse, way beyond the usual atmospheres of GitHub and websites, the Argo community engagement is taken, quite literally, to a new world !

As the first step into the development of this amazing community hub, Argo Metaverse v1.0 does not configure as a multi-user and fully interactive online metaverse, but serves as a prototype which showcases an initial vision for the Argo Metaverse and the potential it holds.

This manual is intended to clarify the structure of the project and main assets, as well as the functions of the playable build, and should be consulted for troubleshooting.

2. Compatibility/Requirements

2.1. Unity version

The project was created in Unity version **2021.3.4f1 LTS**, and should only support it by default. Other versions have not been tested.

2.2. Required Unity Packages and Version

- Input System 1.2.0

- Cinemachine 2.8.6
- TextUnity 2021.3.4f1
- MeshPro 3.0.6
- Universal RP 12.1.7 (and dependencies)
- Visual Effect Graph 12.1.7
- Timeline 1.6.4
- Unity UI 1.0.0

2.3. Compatible systems

Development — Although developed from the ground up in Windows, the project has been tested and runs well on MacOS Monterey 12.5.1, running on an Apple silicon machine (M1). It should be compatible with most Mac environments, given that they are running the supported Unity version and maintaining the URP version included in the project.

Builds — Similarly, most test build compilations were conducted in Windows and for Windows, but compiling for Macs on MacOS produced good results. However, it is worth mentioning that testing in the Mac environment was limited (builds compiled on MacOS 12.5.1 / M1 Apple silicon, and tried on similar machines), and that the resulting apps are not registered out of the box. In this way, it might be necessary to bypass OS warnings or circumvent its security measures in order to run them.

Click the link to find more information on how to
Open a Mac app from an unidentified developer - Apple Support

3. Project Structure (Assets)

The bulk of the project consists in a collection of assets divided in a few base folders within the default Unity “Assets” folder. In each of these folders you will find multiple files/assets that share a common purpose. According to their format/type, assets are used by corresponding functions or settings of Game Objects (items which serve as pieces to construct scenes; they are laid out in the Hierarchy panel when a scene is opened. See 4. Hierarchy).

Not only may assets be used by Game Objects but they may also connect with each other when functioning. In this way, deleting a single file or asset may break one or more functions in scenes or in the project in general. However, renaming and/or moving files using the Unity “explorer” (the “Project” panel in the editor) does not cause links to break and should not generate any issues. Thus, arranging files within the editor can be used for further organization and sorting of assets.

It should also be noted that assets of a certain type may be found in a different main folder than the one for its particular type in case that it helps organization or the project structuring.

Full vs Open source Project

The full project includes diverse assets from third-party sources, mainly the Unity Asset Store, which fall under certain end-user agreements that prevent open distribution (see 3.3 - Plugins). To abide by such terms, these assets have been removed from the open source version of the project, and different scenes were created to facilitate development without them (see 3.6 - Scenes).

Exclusive third-party assets (not available in the open source version) can be found under similar folder structures inside the folders dedicated to each of those sources (see 3.3 Plugins).

3.1. Art

This folder is mainly dedicated to animation files, textures and materials used in the project, particularly those created by the development team rather than imported from third-party sources such as the Unity Store assets (see 3.3 Plugins).

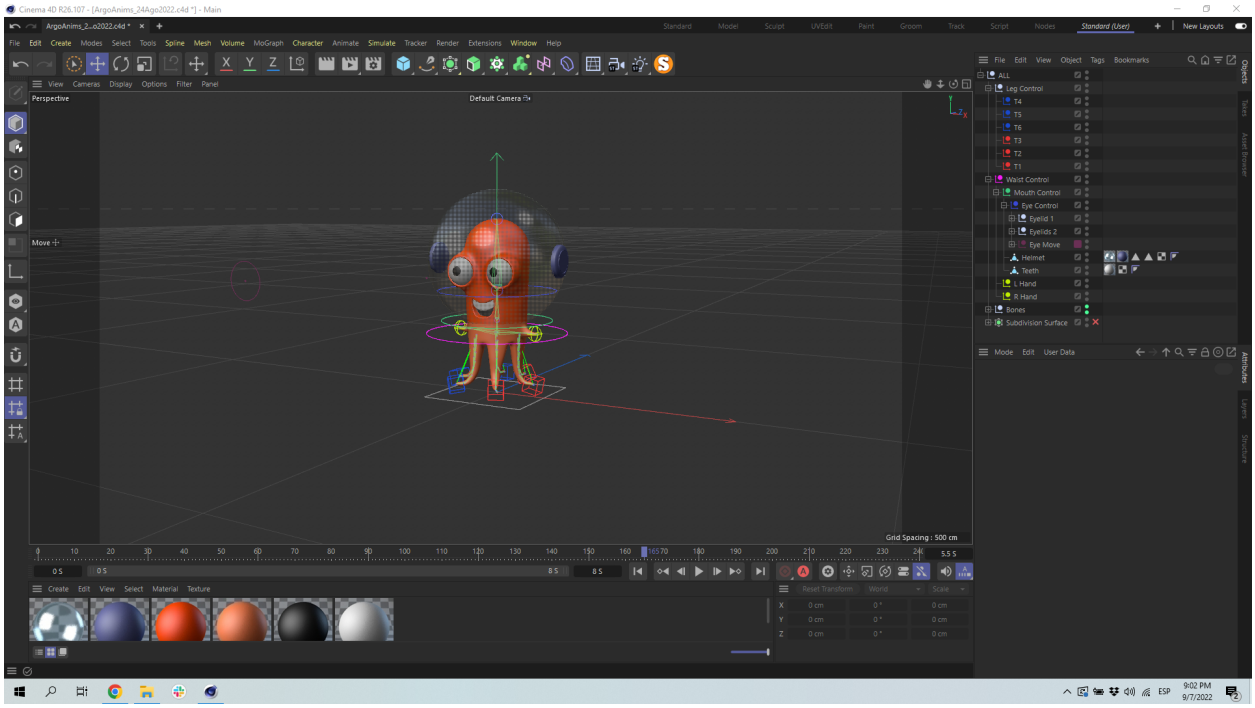
3.1.1. Animation

Here you can find animation assets (both .controller and .playable files) which are being used to generate animations for certain Game Objects.

3.1.1. A - Argo Character Animation

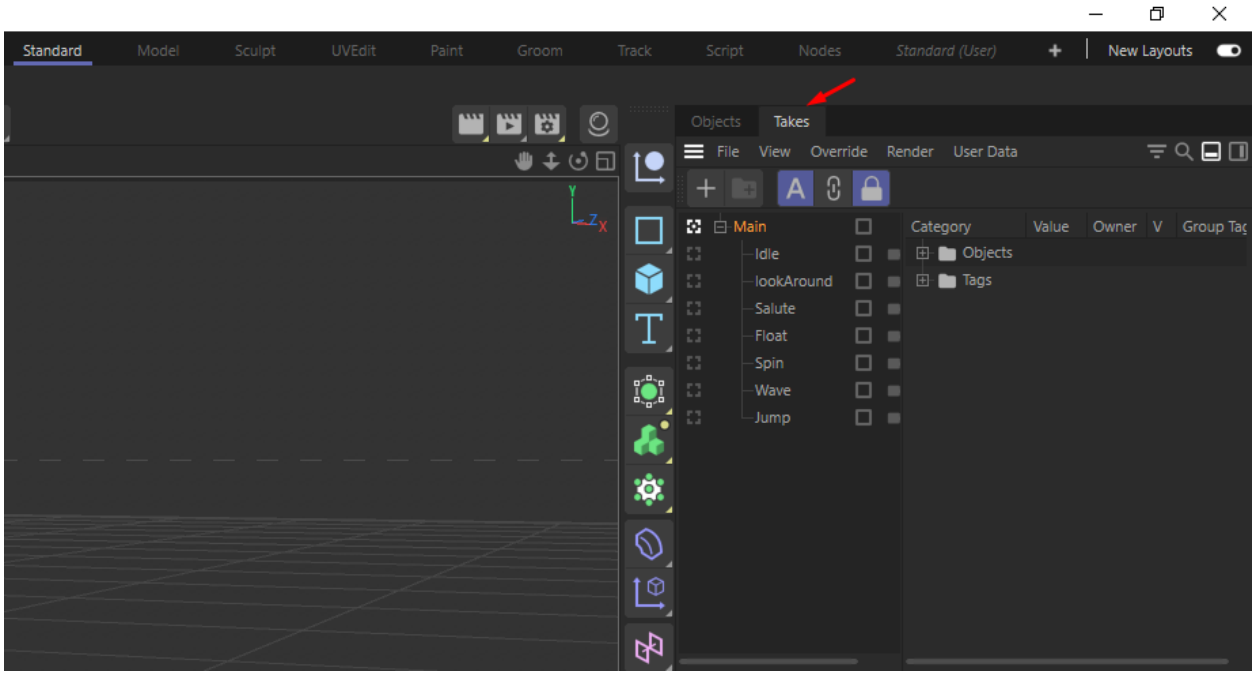
Our character rig was built with Cinema 4D S26, we will highlight the main animation controllers for ease in finding such properties, the rig was structured as follows:

- All Controller
    - Legs Controller
        - 6 Controllers for each tentacle
    - Waist Control
        - Mouth Control (Controls body rotation around the mouth area)
        - Eye Control
            - Eyelid 1
                - Top
                - Blink (Controls blinking with rotation pitch)
            - Eye Move
                - Iris Move (Controls pupils with XY Position)
        - Right Hand and Left Hand Controllers
    - Bones
        - Base
            - Waist
            - R Arm
                - Bicep
                    - R Hand
                        - Check Constraint tag to turn on/off IK
            - L Arm
                - Bicep
                    - L Hand
                        - Check Constraint tag to turn on/off IK
    - Subdivision Surface
        - Body
            - Pose Morph Tag
                - This property controls when argo opens his mouth.



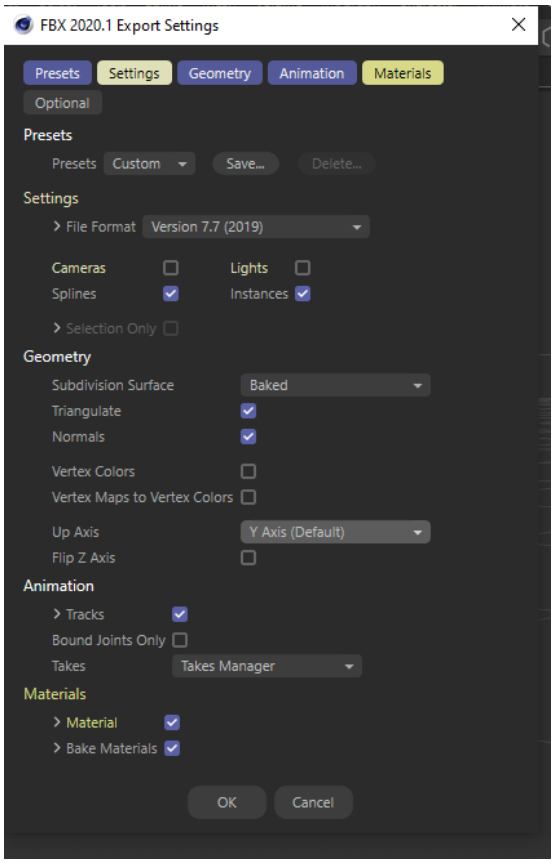
Argo Character Rig

All current animations can be found on the same C4D file, we built it using C4D takes.



Cinema 4D Takes

When exporting the FBX, make sure your C4D file (Ctrl/Cmd + D) project scale is set to 1 on the centimeters unit.



Make sure on your FBX export settings cameras and lights are turned off.

Subdivision surface set to baked, triangulate & normals enabled.

It's also very important that when you are exporting the FBX you go back to the Main take view and export your FBX then, if you export the FBX while you are on a take that is different from the default leg position and rotation (for example float animation) you will affect all the other animations.

- Materials are substituted within unity (see 3.2 Extract materials)

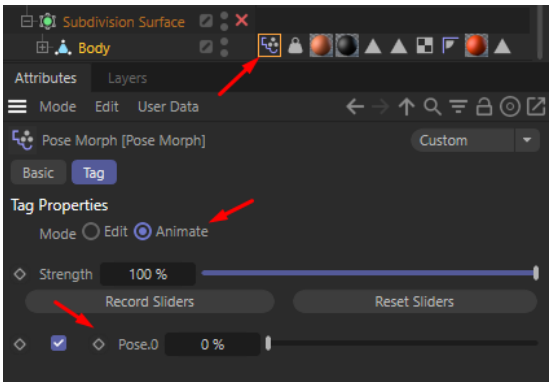
- Keep in mind that any changes on the bone structure, mesh and weighting will likely break the model and animations.

- To make it easy to update Argo with new animations, please name your FBX export **"ArgoAnims"** then replace the according file located in \Argo Metaverse\Assets\Art\Animation.

  Save a backup of this FBX in case something goes wrong.

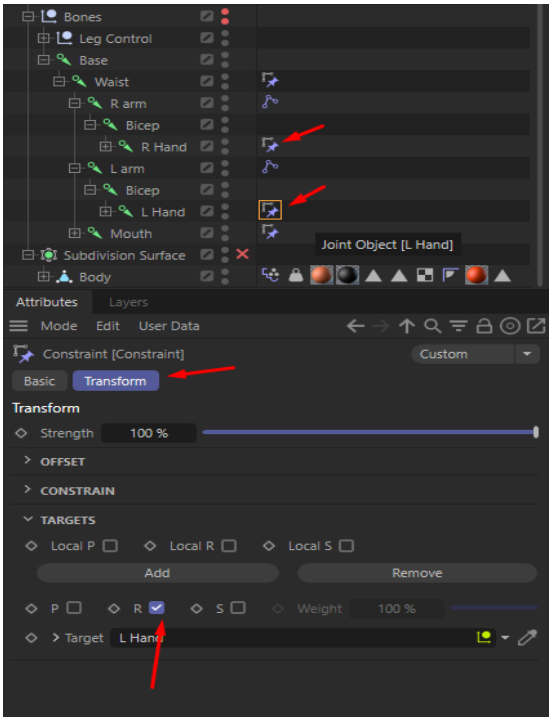
Additional Notes

For animation, Argo's default mouth position is open, when creating new animations you should go to the mouth pose morph tag; which is inside the subdivision surface group; drag the slider up to 100% and set a Pose.0 keyframe.



Pose morph tag

For animation, Argo's default hand settings are set to IK turned off, this means if you move the arm, the hand's rotation angle will not be affected; thus giving creating gesture opportunities. If you do not need this amount of control for the arm you can turn on IK again by heading to Bones> Base> Waist> R Arm/L Arm> Bicep> R Hand/L Hand and locating the constraint tag. Click on the Transform tab and uncheck rotation.

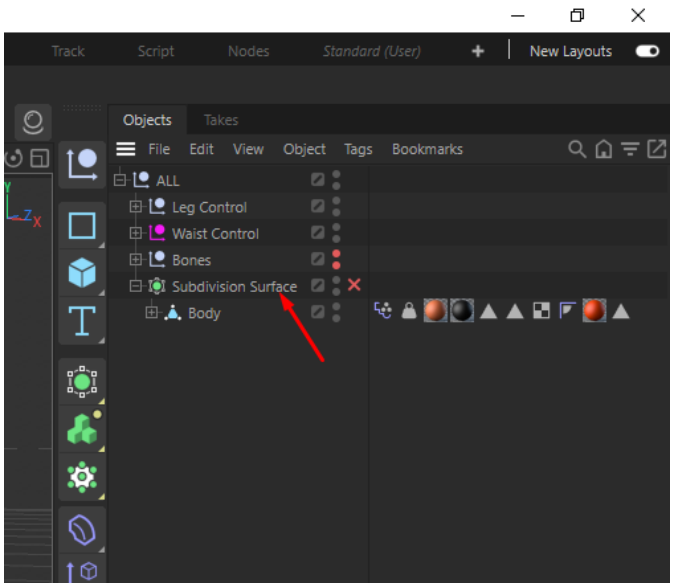


Constraint tags

Known Issues

- The rotation property applied (for example using the “All” controller) on the whole skeleton/rig does not transfer well to unity.

- On the “Main” take you must always keep the subdivision surface unabled, otherwise mesh glitches will appear on unity such as a spiky mouth on argo or bumpy texture on his skin. This does not happen with the other takes where the main animations reside.

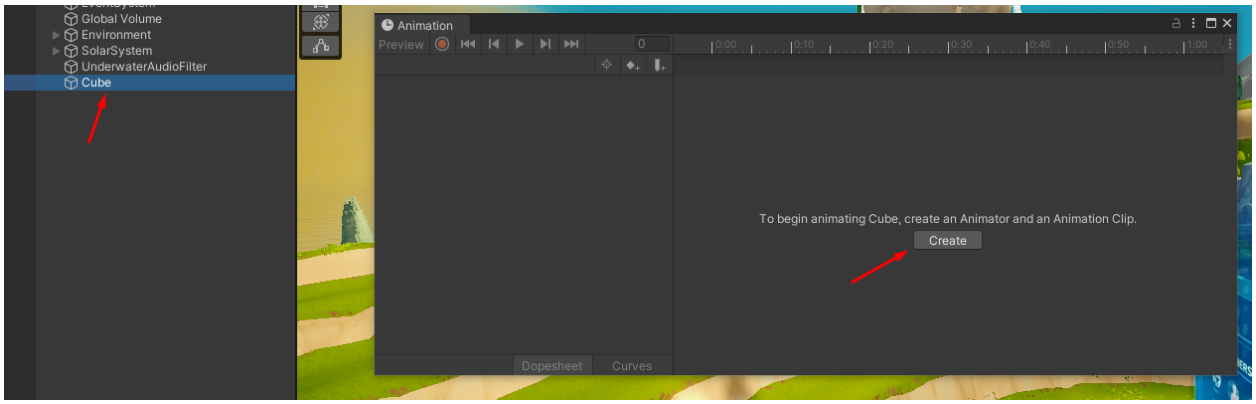


Unabled subdivision surface, The red X lets us know this was deactivated.

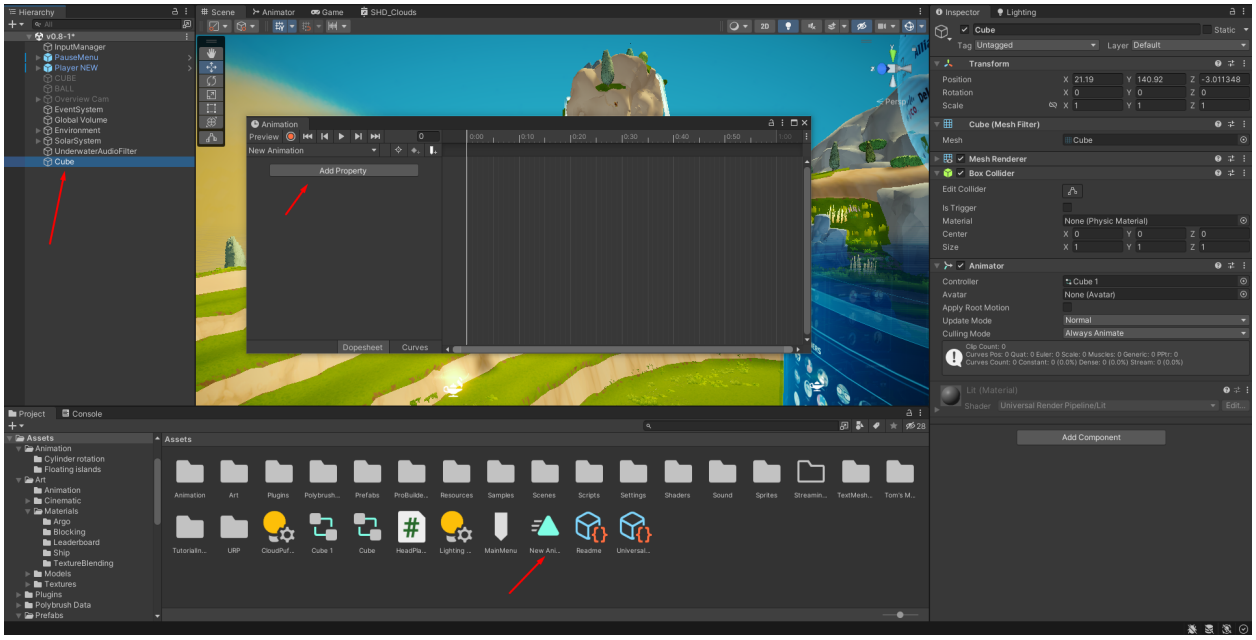
3.1.1. B - Simple Animations in Unity

Locate Window> Animation> Animation (Shortcut Ctrl + numPad 6)

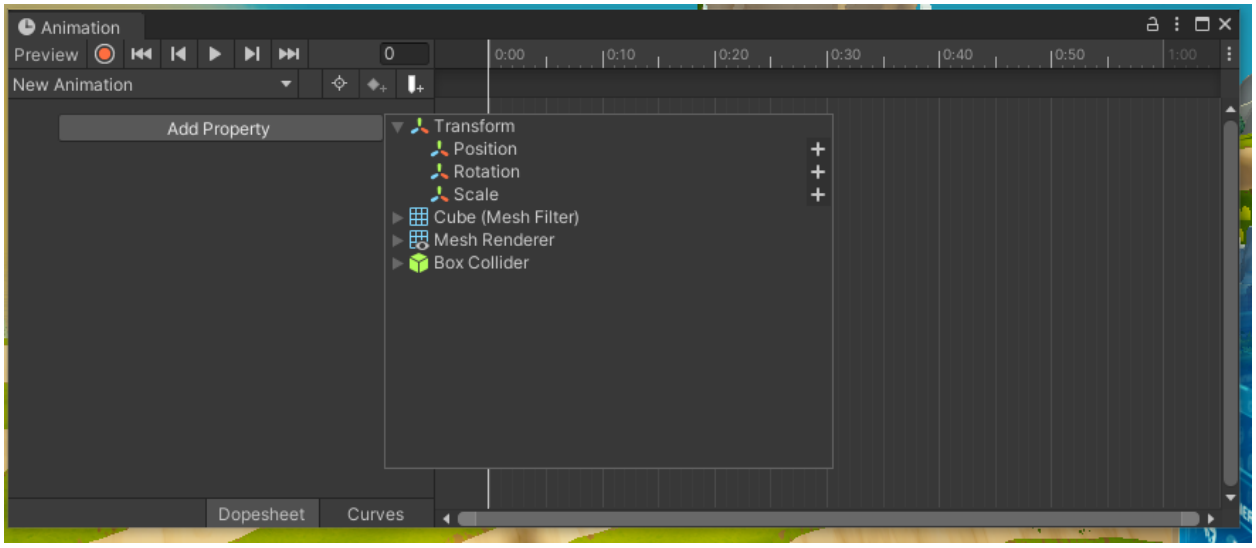
Click on any 3D asset on the hierarchy tab, go back to the animation window and click on “Create”. This will pop up a new window where you decide where the animation controllers will be saved.



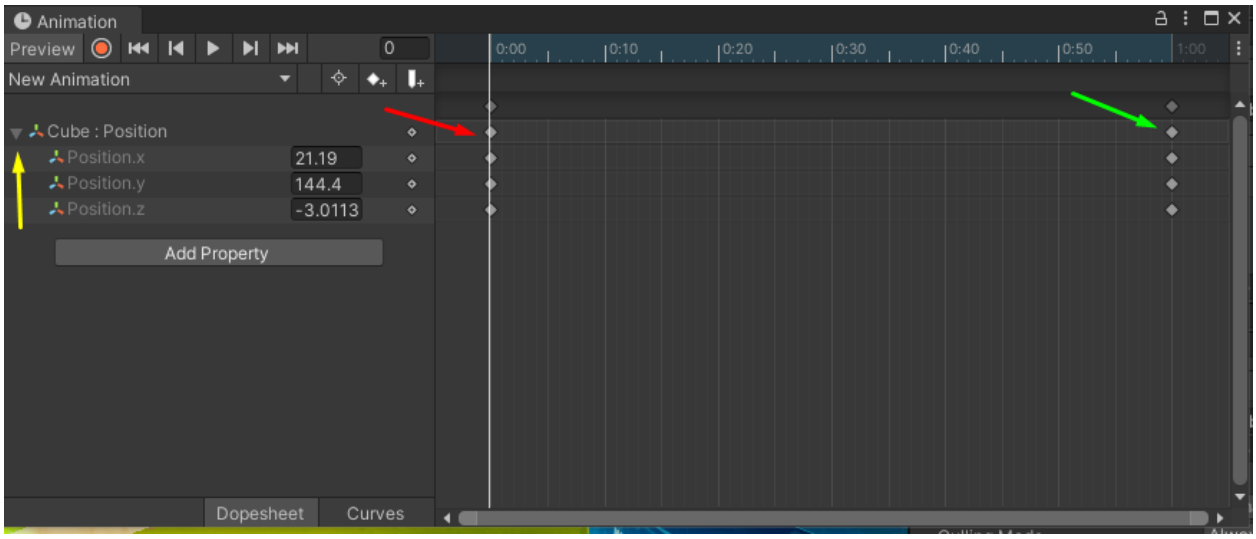
Thus an animation clip will be created (cyan triangle); select your 3D asset.



Click on “Add Property” and here you can control the position, scale and rotation; among other options, press on the “+” button to include said property.



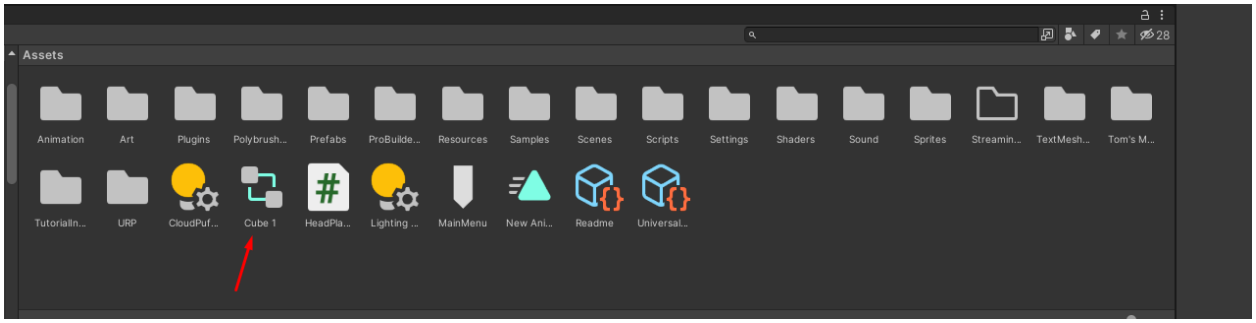
Unity will generate one initial keyframe (red arrow) and one final keyframe (green arrow), click and drag the vertical white line and position yourself over said keyframes, click on the little triangle next to your selected asset’s file name to display the drop down menu, here you can modify the parameters previously enabled. Unity will automatically loop this animation for you.



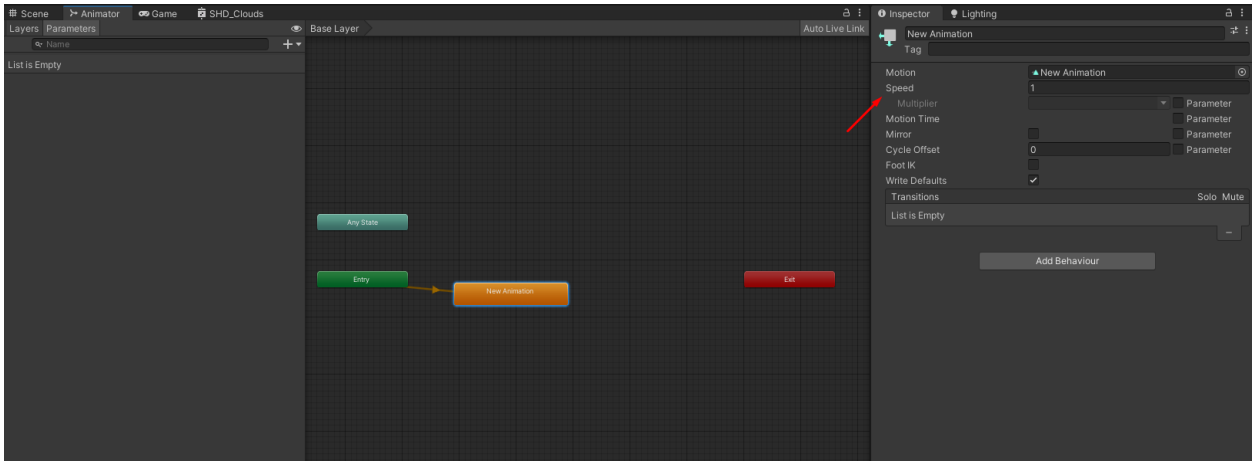
To modify the speed of the animation use the scroll wheel to extend the timeline's length (previously one second, now it's ten seconds), hide the drop down menu or click the top most end keyframe and drag to your desired location.

This is how we set up the community board rotation and the ship flying around it. We created an empty game object and placed it high above the community board, we then aligned the ship to fly around a designated radius, we placed it under the empty game object and on that same game object we gave it rotation keyframes from 0° to 360°; a full rotation to make it loop.

Another way to modify the animation speed without altering keyframes is to use the animation controller which is automatically generated once you assign a property.



Double click this animation controller and you will open the animator tab, click on the orange box and locate the speed parameter on the inspector tab. You also have the option to reverse the animation (mirror), offset in time and more.



3.1.1. C - Simple animated FBX setup in Unity

Unity allows the use of animated FBX from third party programs such as Cinema4D, Blender, Maya, 3DS Max, etc into the game world.

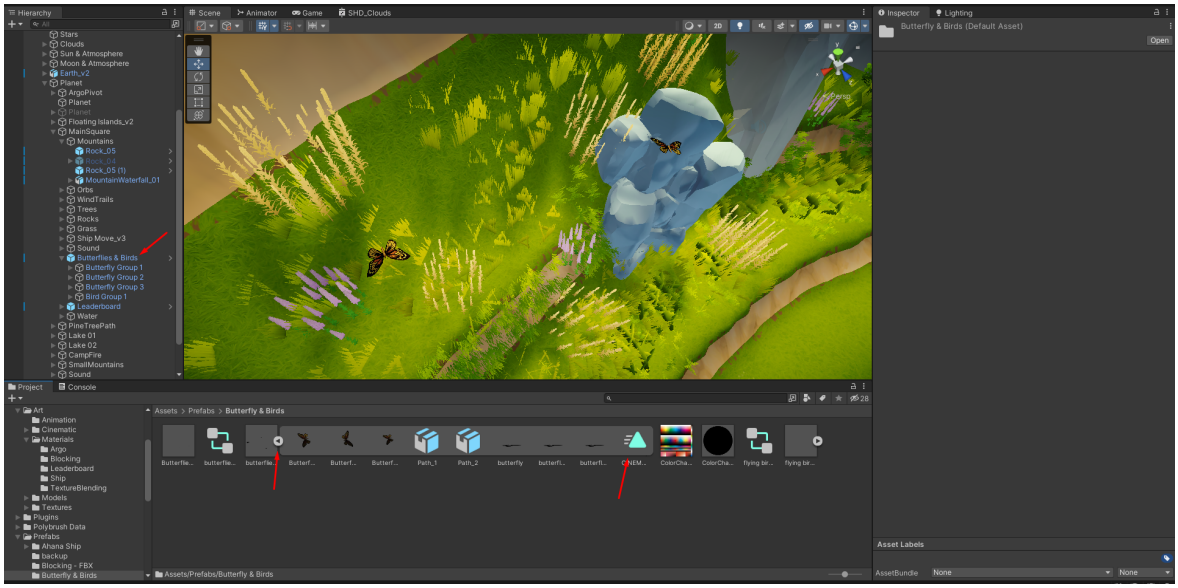
This is how we brought flying butterflies, birds, swimming fish and chest opening animation into the Argo world.

Be mindful not all properties from these programs will transfer perfectly into Unity, a safe option is to use simple position, scale and rotation. The following method is for translating looping animations from such programs into the Argo project.

Cinema 4D

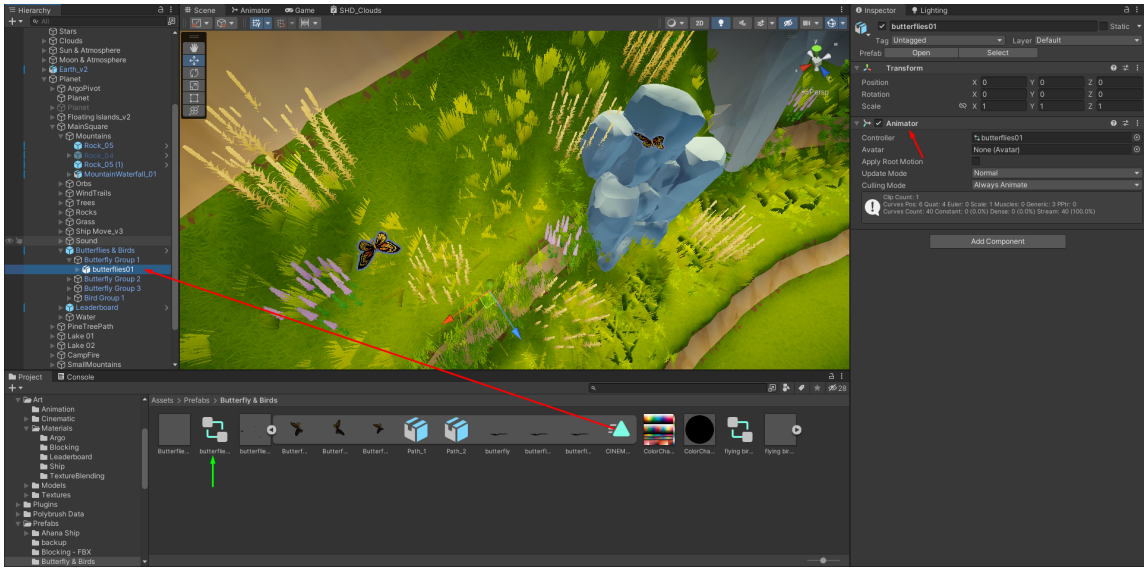
Drag and drop your FBX into unity's asset tab, place your 3D asset on the world.

In the asset tab, open the generated prefab, locate the cyan triangle (animation clip) which contains animation information.

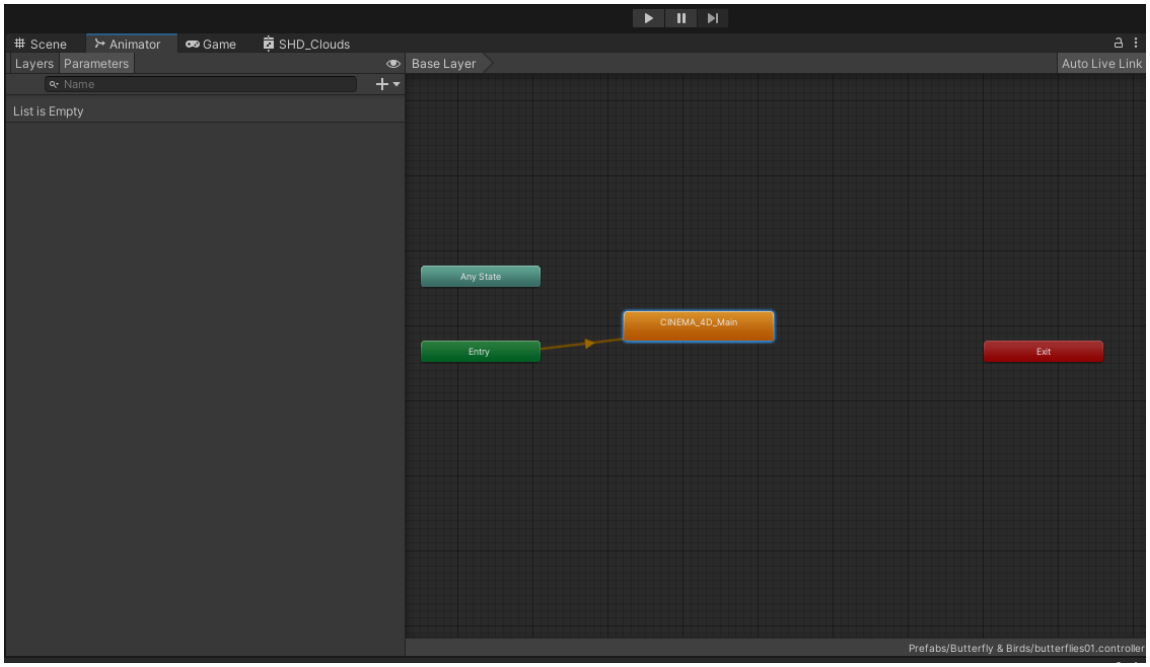


For this case, we have a group of 3 butterflies from the original FBX which were duplicated and rotated inside unity to create variation and make this animation more random and believable.

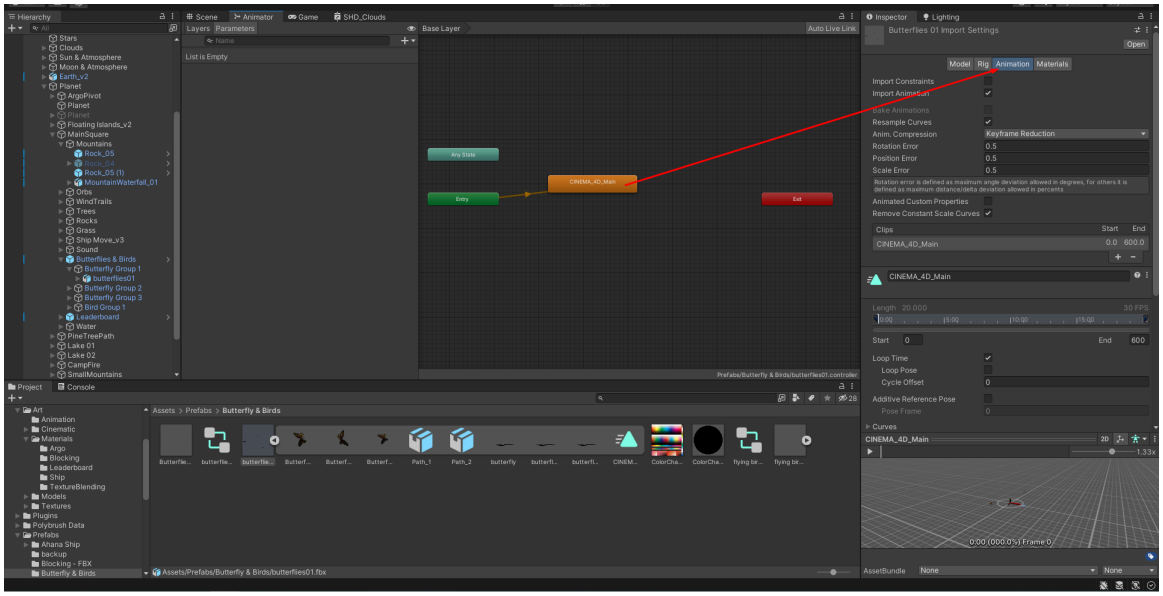
Drag and drop this cyan triangle onto your FBX located in the hierarchy tab, an animator controller will be created (green arrow). We placed the butterfly group on an empty game object for ease of moving around.

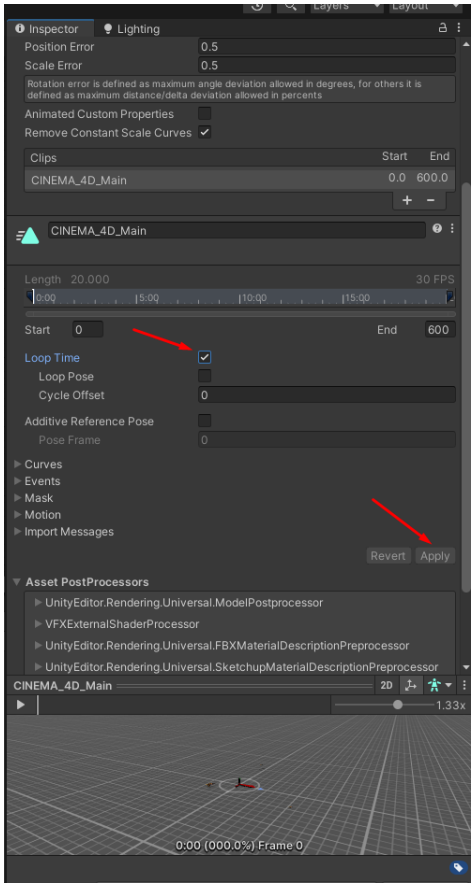


Double click on the animation controller, the animator tab will pop up, double click the orange box to open your animation settings.



By doing so you will go to the Animation tab of your inspector, on the bottom right corner you will get a preview of your animation, it the play button to check if everything is working as it should.





Scroll down a bit on the animation tab; to make your animation loop, check Loop Time, and then hit apply.
If you want your animation to play only once keep loop time unchecked.

3.2. Misc

In this folder you can find diverse types of assets which for some reason were not moved to a specific location. They might not fit within any of the categories of assets present in the structure here explained, or they might serve a particular purpose which makes it easier to have them singled out in this location.

3.3 Plugins

Here is gathered all of the third-party assets sorted by their respective main folders. Each of them offers a folder structure similar to the one laid out here, and may be understood as a sub-project which helps achieve a certain project need.

In the Argo Project, third-party assets have been used mostly for improving staging and aesthetics, by adding Game Objects such as trees, bushes, 3D grass, shaders and particles that make up windtrails, dust, water splashes and clouds, as well as arranging better structure for sound. Unfortunately, such features could not be recreated for the open-source release yet, but they do not interfere with the game's foundations. In this way, although the difference in looks and feel is blatant, their removal from the open-source project should not impact the overall experience of gameplay and game mechanics on which the project is based.

For clarification or troubleshooting related to these assets, please refer to their websites listed below to find further documentation. Also, find below more information on distribution restrictions and what aspects of the game can be affected by not having the assets in the project.

- Argo Metaverse - Unity Store Assets

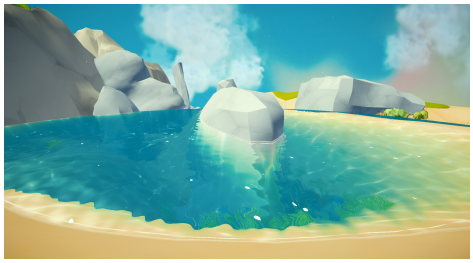
The full project includes or uses diverse assets from the Unity Store. According to the Unity Store End-User License Agreement, found at https://unity.com/legal/as-terms, and particularly to its sections "2. END-USER's Rights and Obligations", "5. Duplication Rights/Back Up Copy" and "9. Intellectual Property", reproduction and distribution of such assets is restricted and open source licensing is standardly not available.

Here is the list of the assets which fall under the EULA and would not be allowed for open source distribution, along with examples of respective use in the project:

- Stylized Grass Shader / Staggart Creations


Grass ground texture and all grass and flowers assets (dark green bush included)

- Stylized Water 2 / Staggart Creations

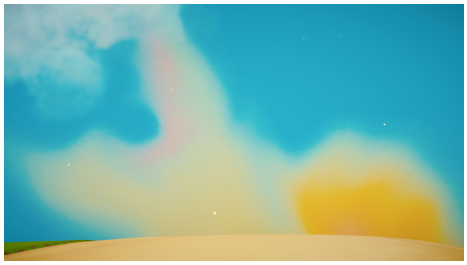



Both still/running water (shader) and splashing/spraying (particles)

- Stylized Nature - Low Poly Environment / GAMEBYTE


All trees and the light green bushes

- Floating Dust Particles / Rivermill Studios

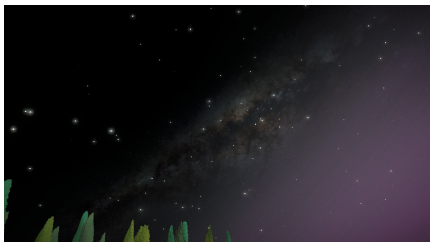


Not the sky, just the moving dusts “dots”(particles)

- VFX Graph - Loot Drops - Vol. 1 / Gabriel Aguiar Prod

Orbs and fire glow/sparks (particles)

- Milky Way Skybox / Adam Bielecki (FREE)

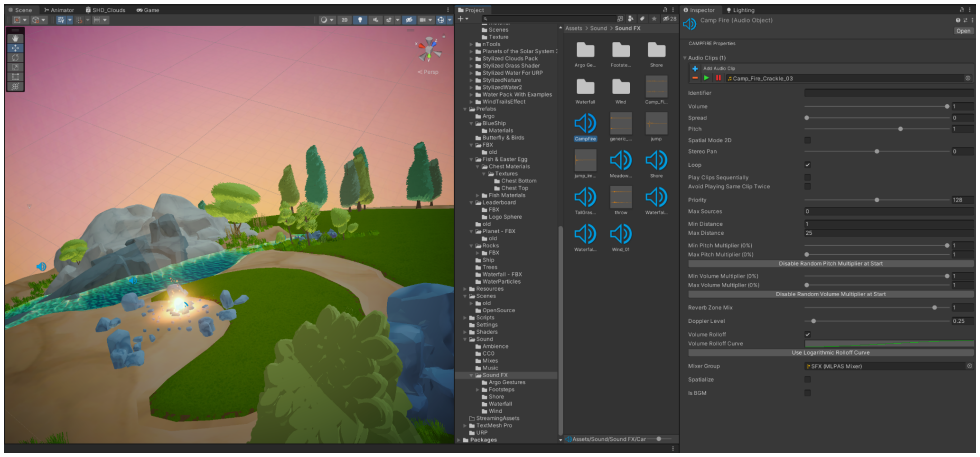


Skybox asset

- Wind Trails Effect / Matt Szymonski

White moving trails (particles)

- MLPAS - Multi Listener Pooling Audio System (FREE)



(see 3.9. Sounds)

3.4. Prefabs (and FBXs)

In this folder you can find two types of assets, FBXs —a 3D bundle whose own assets (be it mesh, material or animation) can be used by Game Objects (including Prefabs)— and Prefabs —assemblies of Unity Game Objects and their Components to serve as root assets, not particular to each Scene but the project in general.

If the assets within a FBX (*filename*.fbx) can be used by Game Objects' components to generate the Objects' main shape or define their materials and animations, Prefabs (*objectname*.prefab) can be a Game Object itself or multiple ones assembled, all with determined settings and components, many times defined with FBXs info/assets themselves. That is why they share this same folder: their use is closely connected, Prefabs making use of FBXs, and FBX behaving like prefabs sometimes (see 3.4.1 FBX / Use / As "fake Prefabs").

3.4.1. FBX

- **Location**

FBX files are not only located in the specified FBX folder inside the main Assets/Prefab folder but also in the other folders inside this main one, (each folder particular to each prefab or type of prefab). Because of organization purposes during development, it made sense keeping FBX files beside their prefab counterparts, making it straightforward to locate and associate an FBX to a corresponding Prefab.

In the specified folder ("Assets/Prefabs/FBX") you will find mostly FBXs which are not used by actual prefabs. These FBXs have been either imported into each scene as a "fake prefab" or used to fill standard Game Objects' components.

- **Use**

In general, using a FBX in a scene will have the purpose of providing its mesh, materials, and animation information to a Game Object (Prefabs included). Clicking the small arrow in their icons (using the Project panel), will show all the assets bundled in that FBX and which can be dragged to a compatible component setting.

Standard use:

Some FBXs in the project, such as *Planet_v0-7.fbx* and *WaterLevel_v0-7.fbx* are used like this, in the standard way, providing their mesh information to fill mesh components of certain Objects in the scenes (In this case: respectively in just one object, *Planet*, and in multiple ones, *WaterLevel.001*, *WaterLevel.003*, *Lake.001*, *Lake.002* and *Lake.003*).

The FBX can also provide materials, but, as noted above in 3.4 - Art, the result is not only unsatisfactory but also limited in terms of customization. Check the mentioned section to see further information, particularly on how to extract materials and then work with them inside unity.

Note that Game Objects' components do not use the FBX file per se, but rather the FBX's own assets, which come bundled together within it and which can be meshes, materials or animations. So, right-clicking a FBX file and selecting "Find References in Scene" may not generate any results even if the FBX is indeed used by the scene. In this case, it is the particular FBX asset which is referenced.
For example, *Planet_v0-7.fbx* is not referenced in the Scenes, but by clicking the small arrow and "opening" the FBX we can find a *Planet* mesh asset within it, and that is what is referenced.
A FBX itself is referenced in a Scene when it is fully imported into it as a "fake prefab".

"Fake Prefabs":

FBX files can be dragged and dropped inside a scene, creating in the Hierarchy an associated "prefab" and corresponding Unity assets within it. Making changes to the FBX on a third-party program such as Blender or Cinema 4D (as long as you save and replace with the same file name) will import all changes into the "prefab", even new meshes and materials.

Only two "fake prefabs" figure in the project:

- *Ship_v3* (which reference to *Ship_v3.fbx)*
- *Earth_v2* (which references to *Earth.fbx)*

However, "fake prefabs" such as these cannot be altered to their root, in a way that translates to all of its instances across different Scenes, like real Prefabs can. In our project, fake prefabs were used when the Game Object was being manipulated by only one team member, and mostly in a single scene, and there was no need for its updates to be replicated in other scenes simultaneously.

3.4.2. Prefabs

Prefab assets found in the main Prefabs folder are used across different Scenes (highlighted in blue when displayed in the Hierarchy panel) and can be modified to their root if opened either through the Project panel or the arrow next to the corresponding Prefab Object in the Hierarchy.

For such reasons, which can greatly improve workflow, it made sense making many of the Project's key objects as Prefabs, keeping their integrity and consistency, as well as facilitating their updating across all Scenes.

The most important Prefabs in the project are :

PauseMenu: Can be found in the root section of the Hierarchy.
Provides an interactive menu when Esc is pressed while the Main Scene is playing (see 4. Hierarchy for technical details)

PlayerNew: Can be found in the root section of the Hierarchy.
Places the user and provides them with first-person camera view and movements when the Main Scene is playing. (see 4. Hierarchy for technical details).

It is deactivated both in Cine and MainMenu scenes. It does not have a function in these Scenes and also needs to not interfere with their MainCameras (see 3.9 Scenes).

Argo: It constitutes Argo Project's main character, from its materials —found in the Art/Materials/Argo folder— to its meshes and animations —both found in the Prefabs/Argo/**ArgoAnims.fbx** FBX— to its animation controllers —found in the Art/Animation/Argo folder— (see 3.1.1. A for further information on Argo Character Animation).

It is used as different instances across scenes to place and give life to Argo for different purposes. In the main scene it is used as an interactable NPC (non-playable character), being found in the Hierarchy under the SolarSystem/Planet/**ArgoPivot** object (a pivot object to allow the Argo prefab's transforms to remain zero'd).This instance of the Argo prefab is deactivated in Cine and MainMenu scenes.

While the prefab does not appear during MainMenu, in the Cinematic scene it is used as two instances (both different from the Main scene's one, which is

deactivated here), through which the Argo character is placed in the cinematic shots. The main one is under the Cinematicsv2/Argo_Cinematic pivot object, which adds Argo to the shots where he is by himself. The second is under the Ship_Cinematic/Ship_Fix/Ship_v3 object (which adds the ship to the cinematic), making it possible to add the Argo character aboard the ship, following the ship Object as it is animated (see 4.8. Cinematics).

LeaderBoard: ********

3.5. Resources

Standard Unity's globally accessible folder for prefabs. It is used by the third party MPLAS Audio System (not available in the open source project. See 3.8 Sounds).

3.6. Scenes

Builds of the Argo Metaverse Project are composed of three scenes: intro cinematics, the main menu of the app and the main playable scene.
Each of these scene types can be accessed in the Unity editor through their corresponding Unity Scene (.unity) files located in the main Scenes folder.

Note that such scenes can be either complete (featuring in their Hierarchies all assets added during development) or "stripped" (as marked in their file name, being stripped of all Game Objects which are based on restricted assets from third-party sources: see 3.3 Plugins for more information on what assets are missing from the Open Source project).

3.6.1. Location

In the full project's root Scene folder (Assets/Scenes) you will find 3 scene files: the **latest (complete) version** of each scene type mentioned above.

- **Complete** main scenes:

Cinematics scene: Assets/Scenes/**Cine *"VersionNumber"*.unity**

Main Menu scene: Assets/Scenes/**MainMenu *"VersionNumber"*.unity**

Main Scene: Assets/Scenes/***"VersionNumber"*.unity**

For referencing and access purposes —such as retrieving information on Hierarchy structure and what are the missing Game Objects— complete scenes were kept in the open source project. You can find them under the folder **Assets/Scenes/Backup CompleteScenes**.

Similarly, open source versions of the scenes can be accessed in the full project if necessary. They can be found under **Assets/Scenes/OpenSource**:

Both in the full and the Open Source projects, you will find in the **OpenSource** folder the **latest "Stripped" version** of each of the 3 types of scene.

- **Open Source ("Stripped")** main scenes:

  Cinematics scene: Assets/Scenes/OpenSource/**Cine *"VersionNumber" Stripped*.unity**

  Main Menu scene: Assets/Scenes/OpenSource/**MainMenu *"VersionNumber" Stripped*.unity**

  Main Scene: Assets/Scenes/OpenSource/***"VersionNumber" Stripped*.unity**

3.6.2. Functionality

All scene types are based in a similar Hierarchy structure (developed from the **Main Scene**), which has the SolarSystem as the main root Game Object. So actually, **Cine** and **MainMenu** are modifications of the Main Scene, having excluded and/or added different assets and objects necessary to achieve their own roles in the build. (see 4. Hierarchy for more information on Objects)

- **Intro Cinematic**

Based on the main scene, it preserves almost all of its complete Hierarchy structure and its objects for keeping the whole Argo World scenario available for cinematic shots.
For the same reasons, a few objects were deleted or disabled for they had no role in the scene, or to prevent conflicts, both technical and regarding the script.

Disabled/Deleted Game Objects in bold (Hierarchy panel, when compared to Main Scene):

- Cine v1.0 > **InputManager**
  Cine v1.0 > **Pause Menu**
  Cine v1.0 > **Player NEW**
  (the function of the 3 objects above do not have place in the Cine Scene)

- Cine v1.0 > SolarSystem > Planet > **ArgoPivot > Argo**
  (Should not appear in the shots. So it gives place to two different Argo instances, which are the ones we can see on the shots. see 3.4.2. Prefabs/Argo and on “Added GAme Objects” below)

- Cine v1.0 > SolarSystem > Planet > MainSquare **> Ship Move_v3**
  (Should not appear in the shots. Gives place to a different Ship Move_v3 instance (see “Added objects” below)

Added Game Objects in bold (Hierarchy panel, when compared to Main Scene):

- Cine v1.0 > **Input + Cinematic Controls**
- Cine v1.0 > **Cinematics v2**
- Cine v1.0 > **SignalToMenu**
- Cine v1.0 > **UI**

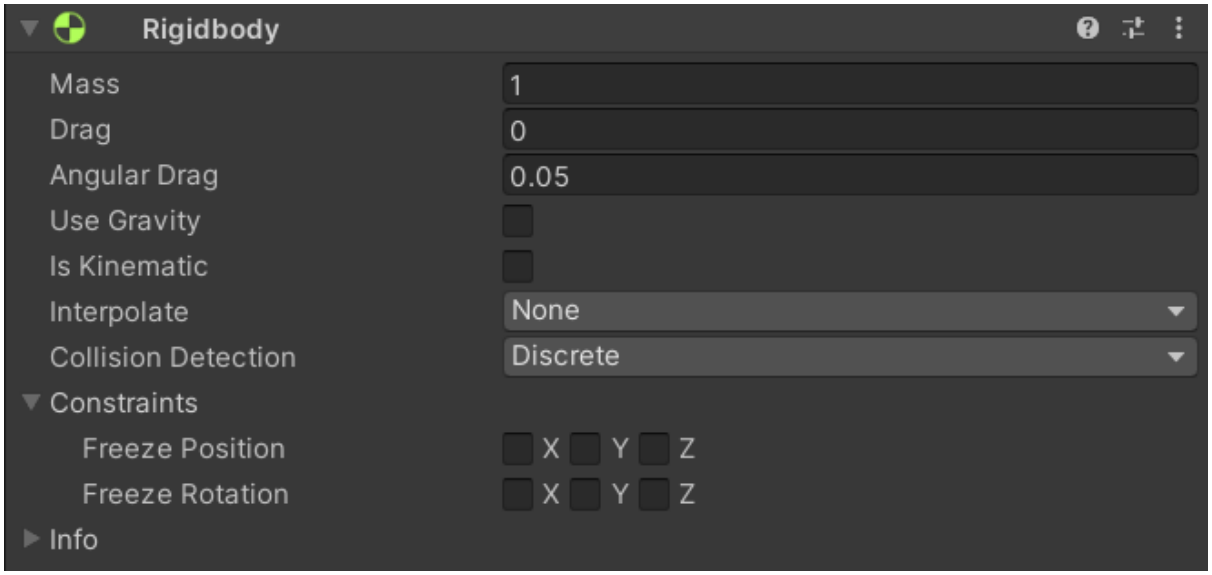
- **Main Menu**
- **Main Scene**

3.7. Scripts

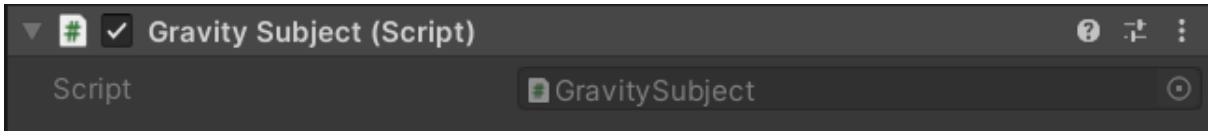
- The Utility folder has a few tools for either scripts or GameObjects.
- Player Stuff has all the player controller scripts and auxiliary script/input assets.
- Interactables has all scripts related to interaction with objects. The leaderboard, pickable objects, NPCs.

3.7.1. Setting up gravity for a GameObject

1. Add a rigidbody and set Use Gravity to off (false):



2. Add a Collider so the object doesn't go through solids.
3. Add the Gravity Subject custom script:

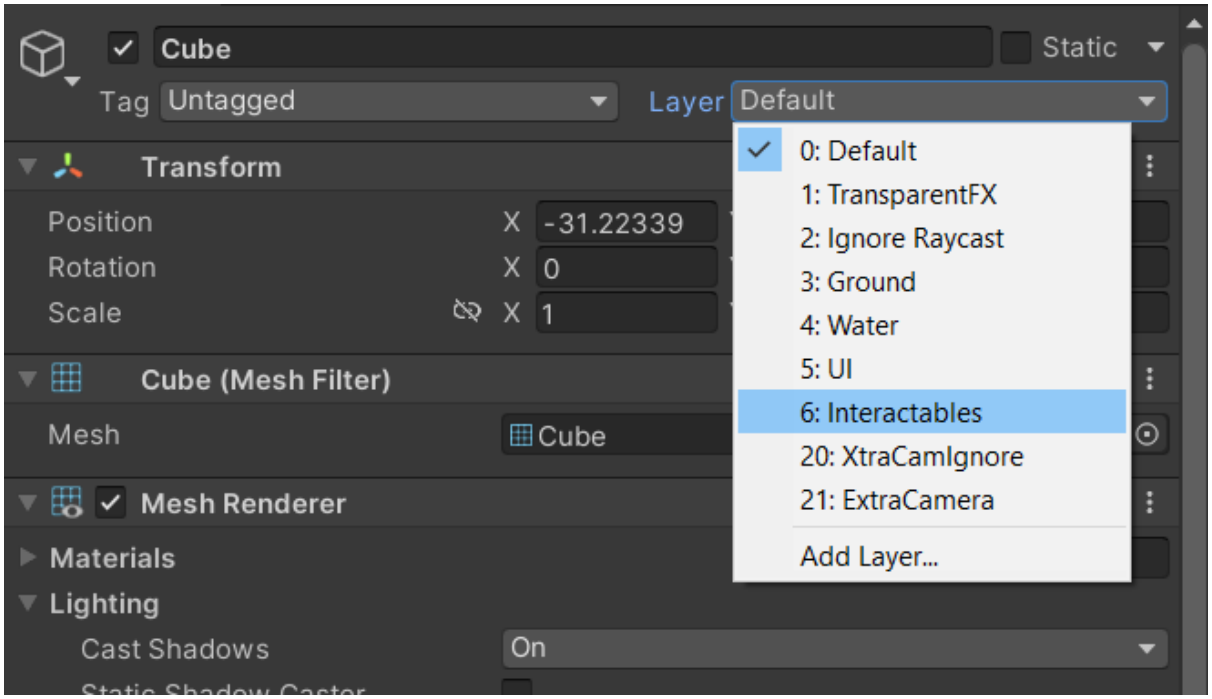


Notes:

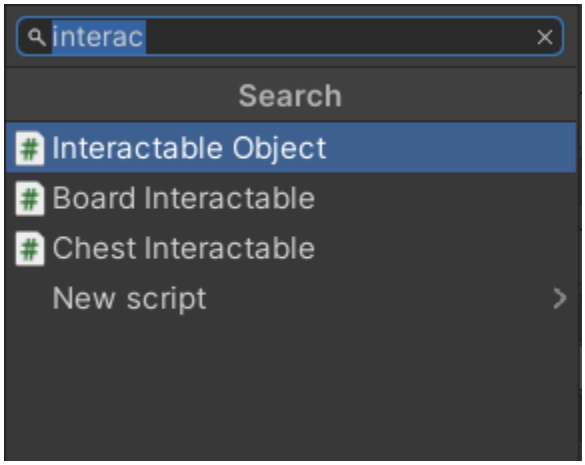
- All Gravity Subjects will look for the only (singleton) Gravity Source GameObject that exists (the planet). There must be one and only one Gravity Source GameObject in the scene.
- Gravity Subjects will have the acceleration of X towards the Gravity Source center point, X being the value set (default 9.7 m/s²). This value can be updated at runtime via the Pause Menu or Unity's inspector.

3.7.2. Making a pickable/throwable GameObject

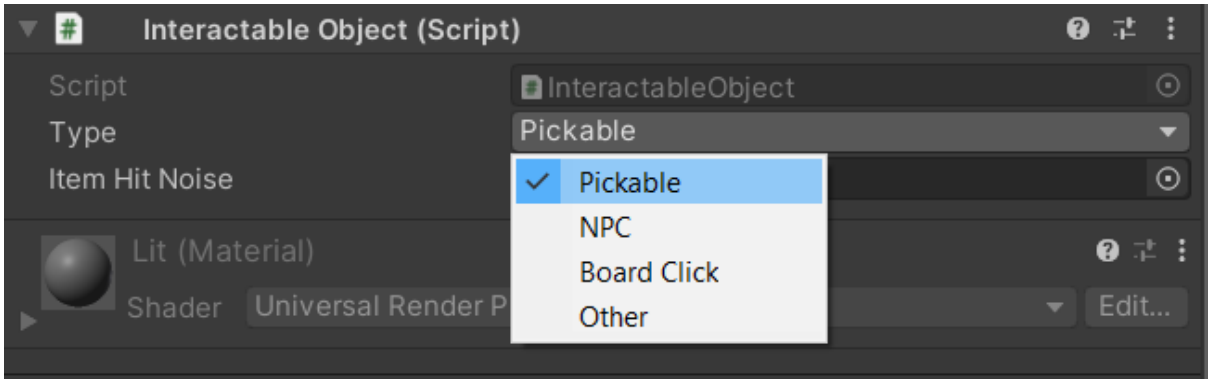
1. Set up the Gravity Subject script for the GameObject as instructed.
2. Set the layer for the GameObject that contains the collider (and all else) to 6 - Interactables:



3. Add the Interactable Object component:



4. Leave the Type set to Pickable:



Notes

- The Player can pick/drop objects with left mouse click and throw held objects with the right mouse click.
- The Item Hit Noise is not fully implemented and not functional.
- The pickable GameObject needs to have the following components: Collider, RigidBody, Interactable Object and be set to layer 6 in order to be pickable by the player.

3.7.3. Setting up an NPC Interactable GameObject:

If you look at the Argo octopus NPC ingame, you'll notice in its components a custom script named Argo NPC instead of the Interactable Object script. Other than that, it has:

- A Collider (can be a trigger)
- Layer 6 (Interactables)

The Argo NPC script inherits from Interactable Object but has new things like all sounds for its animations and such. It also is activated via an exclusive method (Poke()). It could be refactored to support other NPCs but right now it's exclusive to this model/animations. Please keep in mind the Player's interaction method needs to be changed as well as you can see here:

```
switch (targetedInteractable.type)
{
    case InteractableObject.InteractableType.Pickable:
        if (Vector3.Distance(transform.position, targetedInteractable.transform.position) <= INTERACT_DISTANCE)
        {
            if (currentlyHeldItem != null)
                DropHeldItem(2.2f);
            PickItem();
        }
        break;
    case InteractableObject.InteractableType.BoardClick:
        if (_clickingBoard)
        {
            _clickingBoard = !BoardPopup.Instance.Unclick();
            break;
        }
        if (Vector3.Distance(transform.position, targetedInteractable.transform.position) <= INTERACT_DISTANCE * 2)
        {
            _clickingBoard = true;
            targetedInteractable.CLickOnThis();
        }
        break;
    case InteractableObject.InteractableType.Other:
        if (Vector3.Distance(transform.position, targetedInteractable.transform.position) <= INTERACT_DISTANCE)
        {
            targetedInteractable.CLickOnThis();
        }
        break;
    case InteractableObject.InteractableType.NPC:
        if (Vector3.Distance(transform.position, targetedInteractable.transform.position) <= INTERACT_DISTANCE)
        {
            var temp = targetedInteractable as ArgoNPC;
            if (temp.wasClicked)
                return;
            temp.Poke();
        }
        break;
    default:
        break;
}
```

(Image from the Player script, method “Interact()” inside the Input Callbacks region)

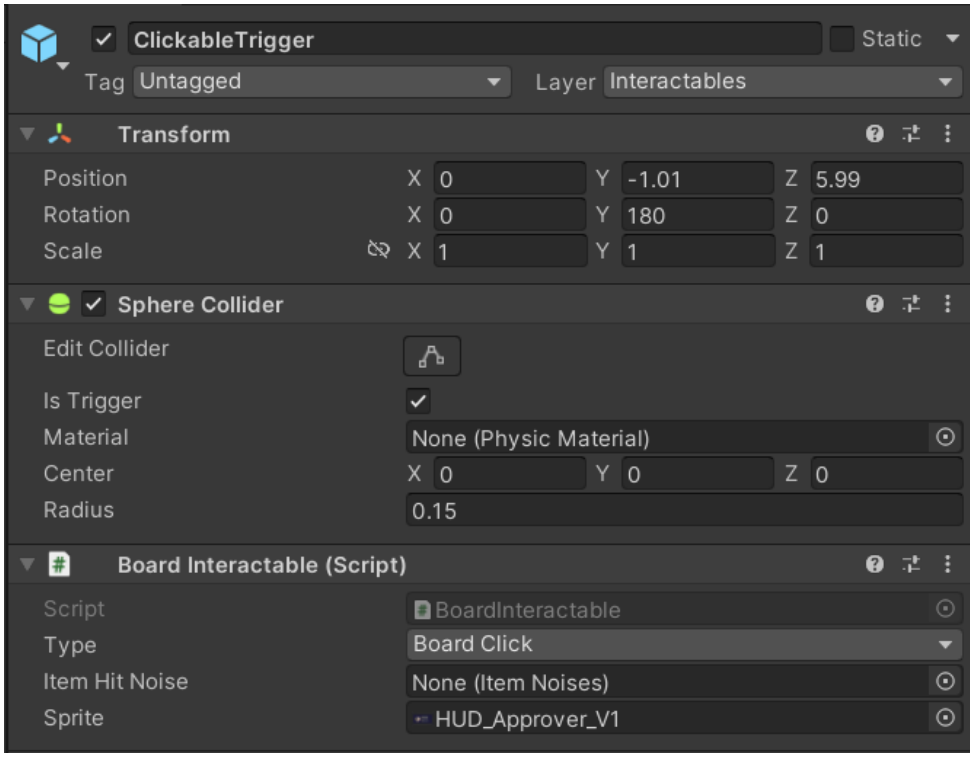
The best way to do it would probably create another intermediary class Interactable NPC and then inherit each specific NPC singular script to that, all being called by a common method. The player would read the targetedInteractable as the intermediary class and call its common method.
It’s also possible to use and call from Player the unused virtual base method ClickOnThis() which can be overridden (and it’s used by other less specific scripts that also inherit from Interactable Object).

Notes

- The .wasClicked boolean visible in this image above is there to forbid the player to click and trigger animations twice. It’s set to true when first clicked and then only returns to false by an animation event in the NPC’s animator, which then calls a method inside the Argo NPC script resetting it to false.
- Each animation was manually set to fire the method inside Argo NPC that plays its corresponding sound file. This is why each NPC would need its own custom script.

3.7.4. Setting up a Leaderboard Popup Interactable

The way it's set for the prototype is just a functional placeholder. Information displayed in the popup should come from an online database, so this script needs further development. However, to set up new clickable areas in the Leaderboard as they are now, follow these steps:



1. Set the Layer to 6 - Interactables.
2. Add a Collider (using sphere as it's the fastest, could be any) and set it to IsTrigger = true.
3. Add the Board Interactable component, which inherits from Interactable Object.
4. Set the Board Interactable Type field to Board Click.
5. Add the image to be displayed by that object into the Sprite field.
6. Position the object in the region of the leaderboard you want it to be (remember to put it as a child of the leaderboard so it moves with it).

3.7.5. Creating other custom Interactables

As long as GameObjects are in the Interactables layer, have a collider (so the Player's raycast hits it) and have a component script that inherits from Interactable Object, it will work.

Inside the project you can see a good example in the underwater chest with its Chest Interactable script:

```
Unity Script | 0 references
public class ChestInteractable : InteractableObject
{
    private Animator _animator;
    private float _timestamp = -5f;
    Unity Message | 4 references
    protected override void Awake()
    {
        base.Awake();
    }
    Unity Message | 0 references
    private void Start()
    {
        _animator = GetComponent<Animator>();
    }

    3 references
    public override void CLickOnThis()
    {
        if (_timestamp <= Time.time - 5f)
        {
            _timestamp = Time.time;
            _animator.SetTrigger("ChestOpen");
        }
    }

}
```

1. It inherits from InteractableObject.
2. Its Awake method can be overridden in case you need to set internal references there. (In this case the Animator component was set on Start but could be on Awake).
3. If the GameObject's InteractableObject (or inherited) component is set to Type: Other, when the Player clicks it, the ClickOnThis method is what's called. So include your specific logic there, overriding the base method.

4. In this particular example all that's happening on the ClickOnThis method is setting a trigger for the animator to start playing the chest opening animation. It also has a cooldown of 5 seconds.

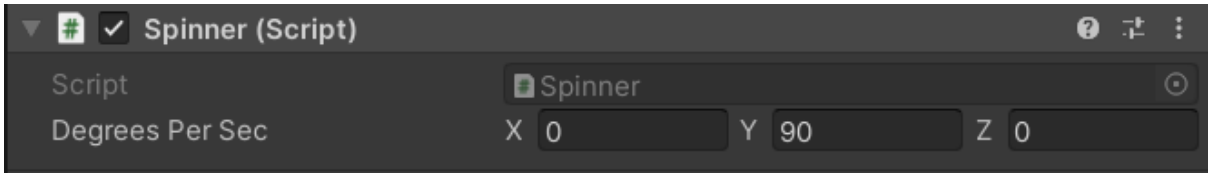
Notes

- If you need different behavior from the Player, say, a different interaction distance for your custom interactable, or a different sprite for the UI (other than the hand), you'll not be able to use the Type: Other. All it does is show the hand sprite if in the default range, then when clicked, call the ClickOnThis method. You should create a new Type and create the new player interaction logic inside the Player script, Input Callback region, Interact() method's switch/case (new case for your new Type).

3.7.6. Using the Spinner component

This simple script was made to spin an object on its axis nonstop.

1. Add the Spinner component to your GameObject.
2. Set the DegreesPerSec vector3 to spin that value on the X, Y and/or Z axis.



(in the example above, the object spins 90 degrees per second, 4 seconds for a full turn, clockwise).

Notes

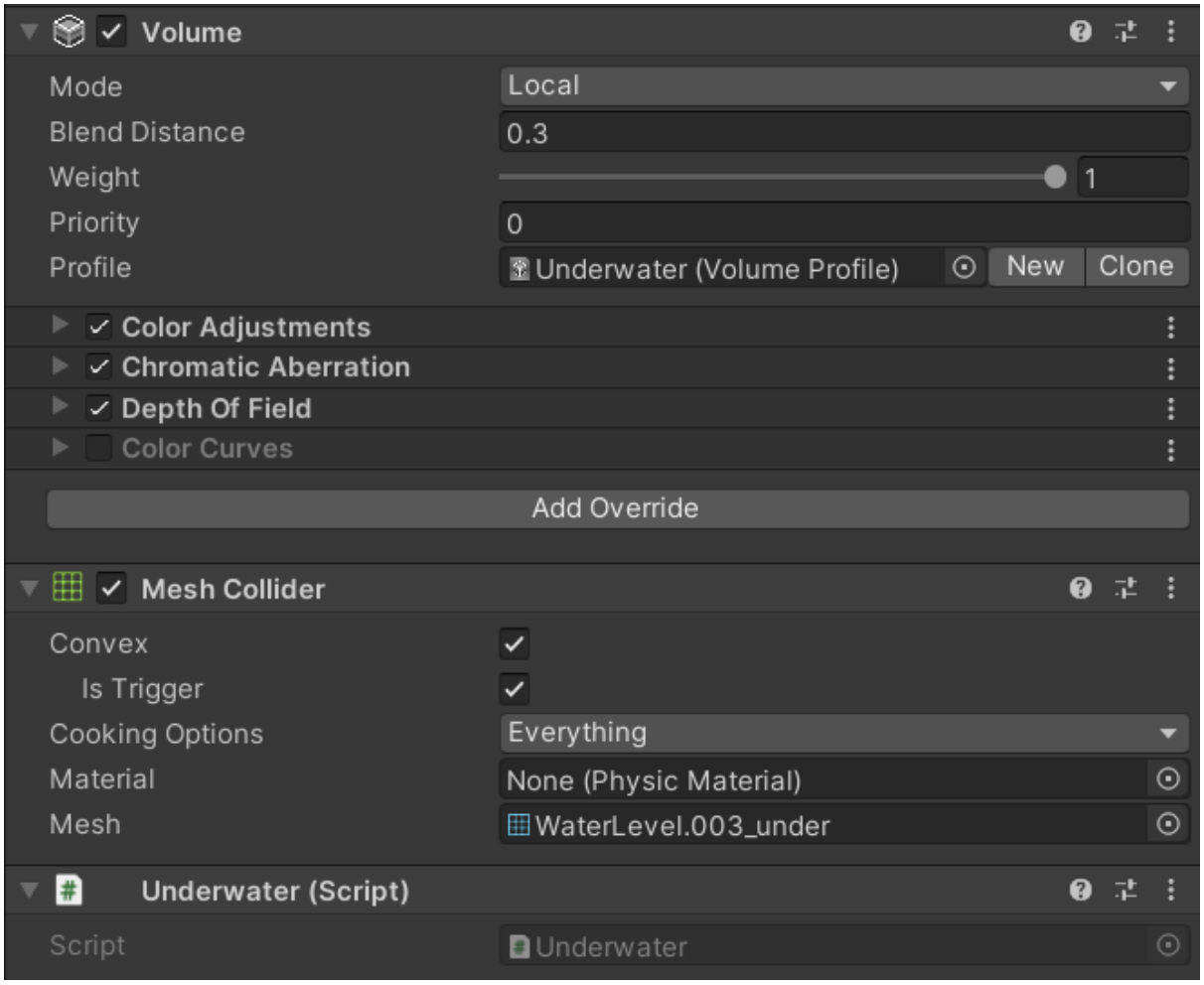
- You can set negative values.
- Higher absolute values spin faster.

3.7.7. Setting up Water bodies

Every water body GameObject in the project is composed of 3 parts:

1. The actual rendered mesh with the water material.
2. An inverted normal mesh with different material for when the player looks up while underwater.
3. Another simpler (less triangles) mesh that goes deeper, to the underground. This acts as a convex Mesh Collider (IsTrigger = true) that will detect the player and camera collisions when inside. And is further divided in two:

- The Volume component is responsible for the post-process effects that will affect the Camera and what the user sees. The “underwater” visual effects.
- The Underwater script component is responsible for communicating to the Player and Gravity Subject scripts that it’s currently underwater and movement should act differently. It’s only a bridge, no actual logic.



(the underwater invisible mesh with it’s required components)

Notes

- Multiple underwater colliders stacking into each other will not work properly. Contiguous bodies of water should have one single Mesh Collider. If the player leaves one underwater zone while still inside another, it will understand it’s no longer underwater and movement and visual effects will turn back to normal.

3.7.8. Setting up walkable terrain

All walkable parts of the map need to have:

1. Collision
2. Layer set to 3 - Ground. If not, it will be walkable but not **jumpable**. Meaning rocks and other solid surfaces should also be set to Ground.

3.7.9. Atmosphere Day/Night Cycle

The Atmosphere script needs 4 requirements to work properly:
It must be on a GameObject which is centered on the planet.
It needs the reference to the atmosphere material, the one whose texture it will offset (from 0.0 to 0.5). So half the texture is day, the other half is night.
The sun position (transform) to determine the “noon” position.
The Player (or Camera) transform, to determine its angle relative to the sun (noon) and find out the current “time of day” and consequent material offset.

Notes
You’ll find the script has many more variables exposed in the inspector screen. It’s a discontinued function that was used to also alter the clouds material, but those were removed from the project at some point in the past. All actual logic in the script was commented/removed so it doesn’t consume processing needlessly.

3.8. Sounds

Up to the v1.0 of the project we used Almenara Games’ Multi Listener Pooling Audio System (MLPAS), which is a package currently discontinued and made free, but still licensed under Unity’s standard licensing; which means it’s been removed for the open-sourced version of this project.

It’s mostly a tool that facilitates implementation with functions like randomizing pitch and volume inbuilt. But it operates essentially using Unity’s audio system, built from it. So the audio queues in code (which have been commented so the code won’t break) can be easily adapted for Unity’s own audio system using Audio Sources etc.

Worth noting MLPAS also can call sounds directly from its Manager, without the direct reference for the file (or Audio Object), those are the Audio Objects inside

the Resources folder (a default path for Unity to find any files at runtime), in the full project.

4. Hierarchy

4.1. InputManager

Responsible for receiving player inputs from Unity's Input System and bridge to other scripts. PlayerControls Input Action Asset (map Player).

Access through InputManager.Instance (singleton) and subscribe (observer pattern) to events on the Start method (not Awake) (eg. inside the Player script: Inputmanager.Instance.OnJump += yourjumpfunction;).

Best practice to also unsubscribe on the OnDestroy method to avoid null reference errors. (eg. Inputmanager.Instance.OnJump -= yourjumpfunction;).

Some of the invoked methods can receive the InputAction.CallbackContext as needed, or values read from them. The Look method reads and passes through the vector2 with screen coordinates so the Player can adjust the camera properly.

4.2. PauseMenu

Pausing the game is handled by the PauseMenu script. Input comes from the PlayerControls Input Action Asset (map Menu). Pausing is directly linked through a Player Input script and calls the ClickPause method inside PauseMenu. Other functionalities are called directly from their respective Unity UI buttons, sliders and such and linked through the inspector manually.
PauseMenu also handles cursor hiding/showing, the game's timescale, SceneManagement back to MainMenu and also the planet's gravity (GravitySource reference needed in the inspector).

4.3. OverviewCam

Looks for the player at Start() through the singleton pattern and keeps track on Update() adjusting its rotation according to the Player's position on the planet. Will result in errors if references are missing in the inspector.

4.4. Player

Singleton (Player.Instance) for other scripts that need access.

The Player script handles most interactions with the world and physics, very few functions are distributed to other scripts in a modular fashion (most of which are also referenced inside Player, like the GravitySubject script and the Raycaster).
The script is divided by regions and many functionalities are commented and explained there. All that was discontinued/not used is commented, tested and ready for use.
All audio queues are commented (since the plugin used was removed for the

open-sourced project). But they are ready to be relinked inside the game's logic.

4.5. EventSystem

Unity standard, updated for the new Input System (currently in use).

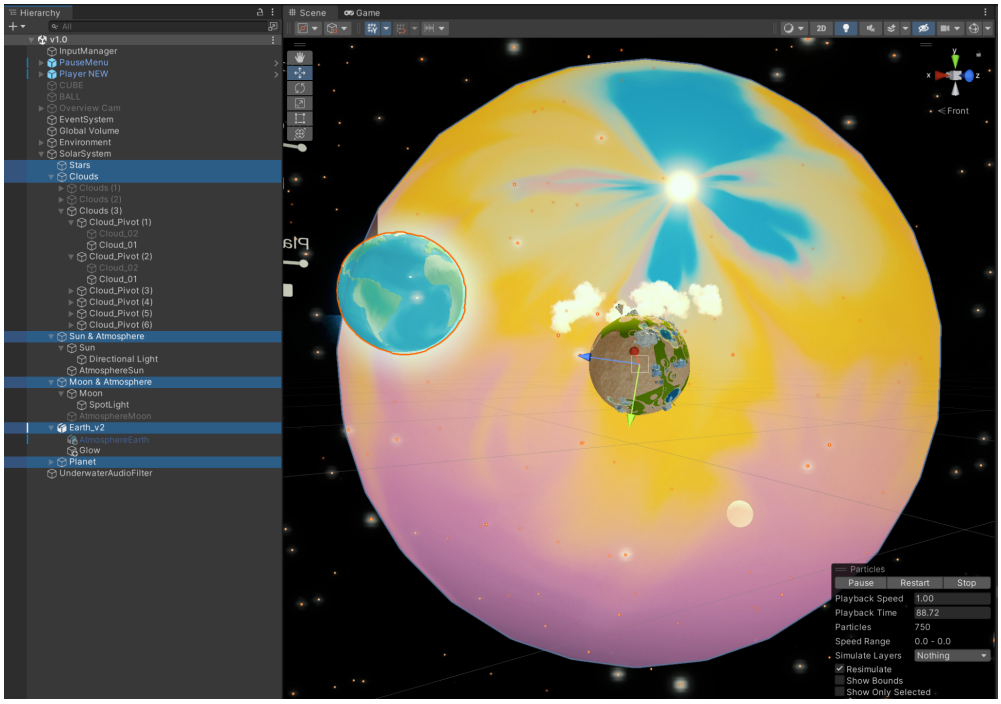
4.6. GlobalVolume

Camera and post-process effects.

4.7. Solar System

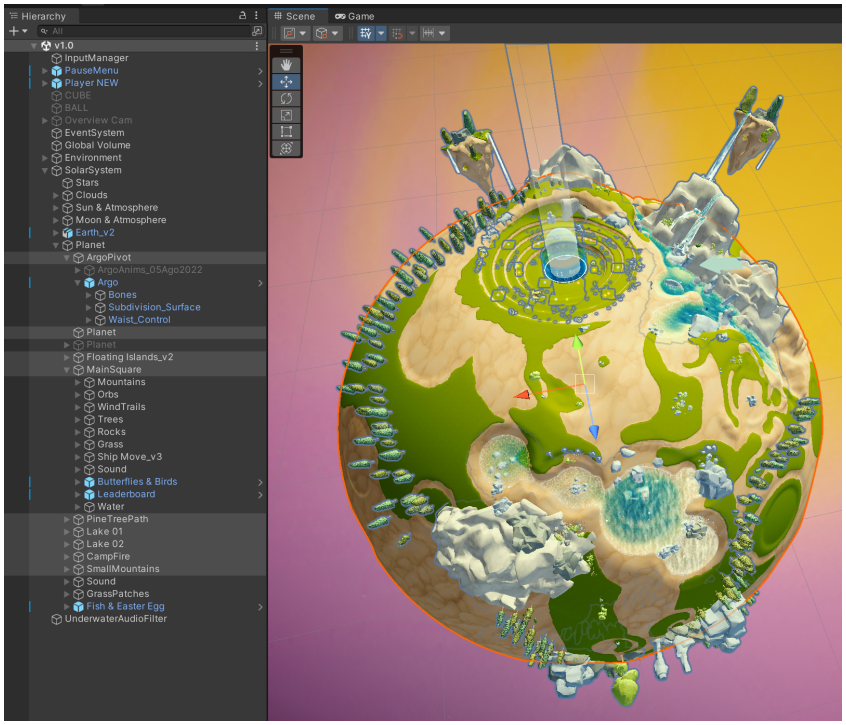
The SolarSystem root Game Object assembles all of the "physical" game objects that compose the 3D environment of the Argo Metaverse. It is called a Solar System because due to the round shape of its main object and game level (Planet), as well as the existence of other "physical" bodies (Sun & Atmosphere, Earth_v2, Moon & Atmosphere, Stars, Clouds), the whole assembly resembles an actual astronomical system.
These 6 main objects mentioned above figure as the primary children objects of the root (SolarSystem). The most important of them are Sun & Atmosphere and Planet,

which together define the very base of the Argo World physical environment.

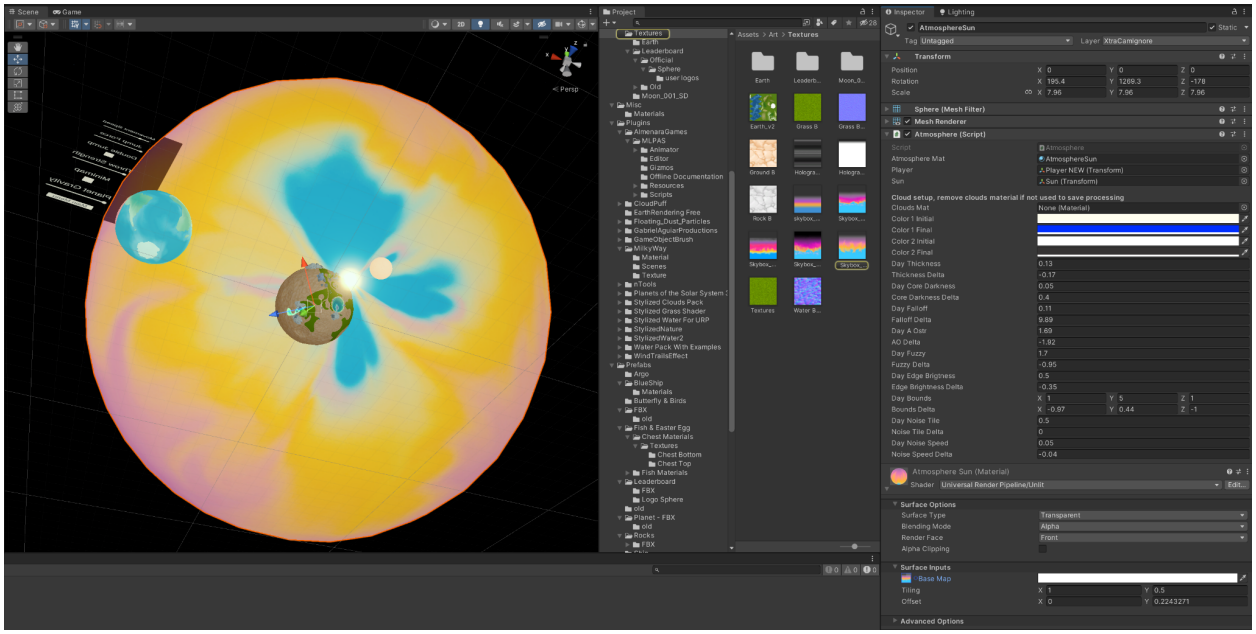


SolarSystem hierarchy structure (the 6 main objects highlighted)

4.7.1. Planet



4.7.2. Sun & Atmosphere



4.8. Cinematics

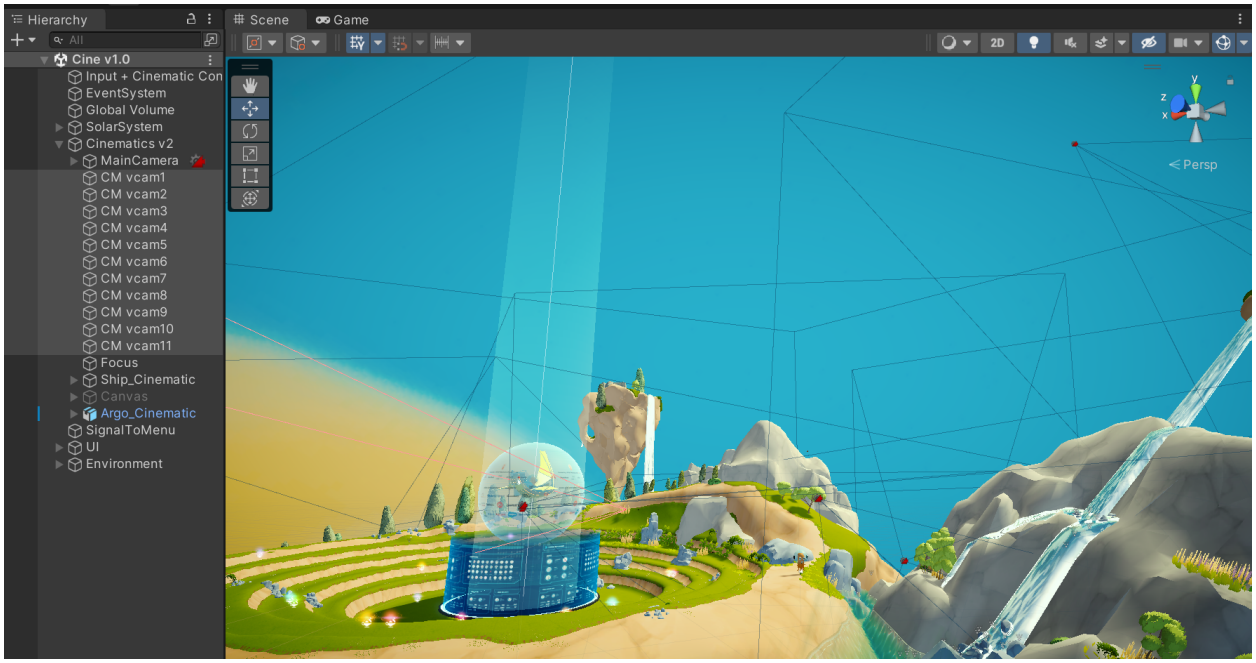
The Cinematics object gathers the necessary cameras, virtual cameras and other cinematic objects in order to allow for the creation of cinematic scenes that can be compiled in builds. In the case of the Argo Metaverse project, it is only the **Cine** Unity scene which has this object as it is the only scene in which cinematics occur. All of these cinematic objects only work together though because they are created and/or connected using Cinemachine.

Cinemachine is a suite of modules for operating the Unity camera. Cinemachine solves the complex mathematics and logic of tracking targets, composing, blending, and cutting between shots. (see Cinemachine Documentation for more information and complete manual)

4.8.1. Virtual Cameras

Cinemachine does not create new cameras. Instead, it directs a single Unity camera (Main Camera) for multiple shots. You compose these shots with Virtual Cameras —which move and rotate the Main camera and control its settings— being separate GameObjects from the Main Camera, and behave independently. (check the documentation referenced above to clarify on how to use Virtual Cameras)

On the Argo project 11 virtual cameras are used. They can be used for one or more (or even for none) of the shots added and edited in the Timeline panel (see 4.8.3. Timeline and Animation).



4.8.2. Cinemachine Brain

The Cinemachine Brain is a Script component in the Unity camera itself. Cinemachine Brain monitors all active Virtual Cameras in the Scene. It chooses the next Virtual Camera to control the Unity camera. It also controls the cut or blend from the current Virtual Camera to the next.

4.8.3. Timeline and Animation

The Timeline panel (Window > Sequencing > Timeline) is used to activate, deactivate, and blend between Virtual Cameras and combine Cinemachine with other animated GameObjects and assets to interactively construct the cutscenes.

Note below how the main Cinematics Game Object is selected, making the Timeline panel show all the cinematic objects that are modified during the cutscenes. However, scrolling through the Timeline, will highlight objects being animated at that point: in this case, the CM vcam 8, and Ship_Cinematic.

Note that in Game view it is Virtual Camera 8 which is enabled, as established in the red MainCamera track shown above. And that it not only captures in its view the Ship_Cinematic object, but both are animated by their own animation clips/tracks, in such a way that they move alongside each other when the Game view is played.