DUJAL Documentation

Dexaxi's Ultimate Jam Asset Library is an all stop shop for all things Unity development related. It aims to reduce the stress and time loss of early unity project development (Hence 'Jam' in the naming, since it's during game jams that we tend to need the most time!) by providing several systems and out of the box prefabs that can be incorporated to virtually any Unity game and serve as the basic building blocks of the finished project.

Table of Contents:

Introduction

This document aims to be “half user guide - half technical documentation” on DUJAL (Dexaxi’s Ultimate Jam Asset Library). This library aims to make the process of making any type of game easier by providing easy to use and easily expandable systems and components that are present on virtually all games, these include but are not limited to:

- A Dialogue System
- A Dungeon Generator
- Character Movement prefabs for the majority of possible character controllers
- An Audio System

The systems and components discussed below will include a small introduction, a user facing explanation, and, if applicable, a technical overview aimed at expanding or adapting the component to any particular game.

Systems

Audio System

The audio system is, as the name indicates, a system that can be used to reproduce audio. To add it to a game next steps must be performed:

- Add the AudioManager script to a GameObject in a Scene.
    - The AudioManager script is a persistent singleton, meaning it won’t be destroyed upon loading a new scene, and other instances of it will be automatically deleted from the scene.
- Create sound mixers for SFX and Music.
- Create a Sound ScriptableObject asset for each SFX or Song to be added to a Scene.
    - Assign the pertinent data to the Sound asset:
        - A name (unique)
        - If it’s a SFX
        - Adjust the pitch and volume if needed
        - The audio clip
        - If it’s loopable
- Add all pertinent assets to the ‘Sound’ list in the AudioManager in the scene.
- If needed, add pertinent songs or SFX to the ‘random music list’.

From this point on, since the AudioManager script is static, it can be accessed by:

```
AudioManager.Instance.Play("SoundName")
```

For reference, the AudioManager can also be called to play sounds by passing a Sound asset reference or by passing the audio clip associated to the Sound asset.

The AudioManager has the next functions available to it:

- float GetSoundLength(), returns length in seconds.
- float GetSoundProgress(), returns with the sound progress percent, in 0-100 format.
- bool IsPlaying(string name), returns if a particular sound is playing.
- void Play(string name), void Play(AudioClip audioClip) or void Play(Sound sound)
- void functions to Pause(string name), Stop(string name) and Resume(string name) a particular sound.
- void StopAllMusic(), which stops all Sounds that are !isSFX.
- void StopAllSFX(), which stops all isSFX sounds.
- void StopAllAudio(), which stops all sounds.
- And the same functions but for void Pause…() and void Resume…().
- There's also functions related to the Random Music feature that do all of the aforementioned, but using a random Sound asset from the Random Music List.
- AudioSource GetAudioSource(AudioClip clip), given a specific audio clip, returns the audio source associated to the Sound Asset that contains it

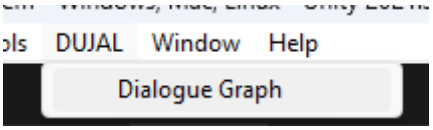
The sound system can easily be extended by adding new public functions to the SoundManager API, some ideas to improve the code as it is to better fit a specific use case would be:

- Make the Sound List be grabbed from a specific folder in the Resources folder.
- Make the AudioManager specific per level, so that only the relevant sounds need to be loaded in memory.
- If performance is key, the AudioManager should internally use a [SoundName, Sound] map to improve search times.

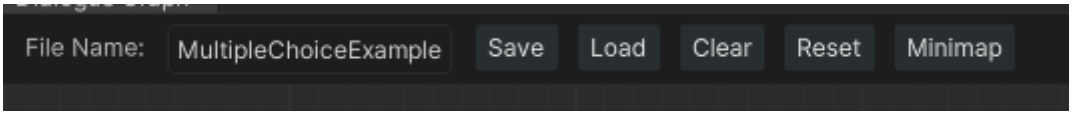
Dialogue System

The dialogue system is a series of scripts that can be used to display dialogues in games as such:

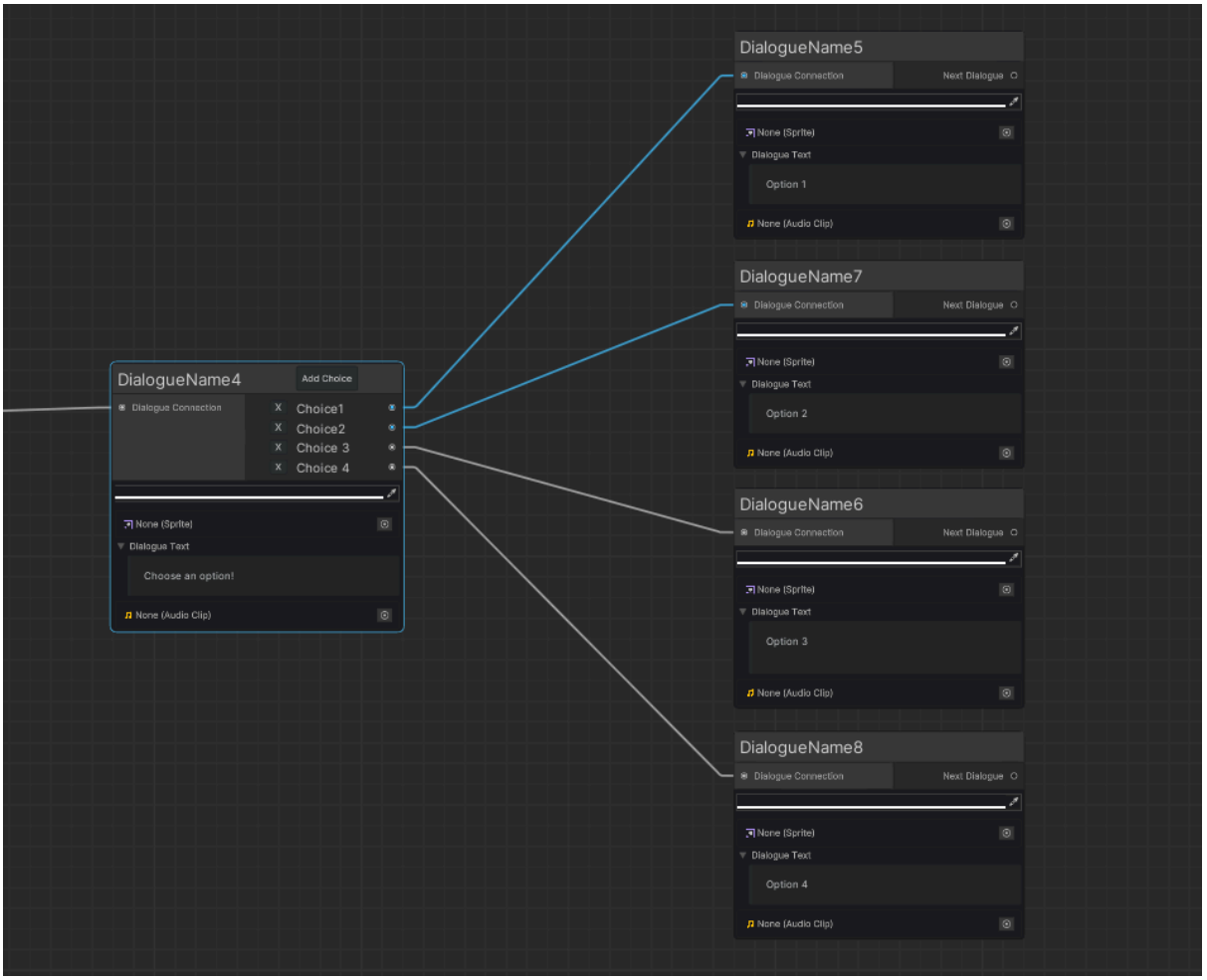
The system supports regular conversation, multiple choice branching dialogues and text animations. To open the dialogue editor, click on the dialogue graph button under DUJAL in Unity's top level navigation bar.



Once opened, the Dialogue Graph allows for saving, loading, deleting and resetting dialogue graph assets.



The graph allows the creation of single choice and multiple choice nodes and their connection. The nodes are read left to right and can be edited to change their colour, audio clip, speaker image, node name and the speaker text.Nodes can be grouped using groups, which are useful as visual aid and when filtering what dialogue to display when setting up the game.



Once a dialogue graph asset is exported (by clicking save), the user can achieve readable dialogue in game by:

- Adding a DialogueSystem component to the scene, (it is not a singleton, so several dialogues can exist at once).
- Configuring the references to display the dialogue, choose buttons for multiple choice questions, the speaker image, what text box to use and the canvas group for the dialogue box, alternatively, the user might choose to use the out of the box pre-configured prefab.
- Select a specific DialogueGraphAsset as the associate node graph, and select the starting dialogue within it.
- Call the void PlayText() function.

In order to apply text effects, the user will have to add a TextAnimationHandler component to the DialogueSystem GameObject. TextEffects can be added to a given text by using tags as such:

```
<b><wobble speed="6",amplitude="5">Fast Wobble!</></b> No wobble :(
<i><wobble speed="2",amplitude="50">Biggest Wobble.</></i>
```

As seen, aside from regular rich text clauses such as   for bold letters, the TextAnimationHandler allows for special effects using the syntax:

```
<effect modifier="modifyAmount"> text to modify </>
```

The effects currently available are:

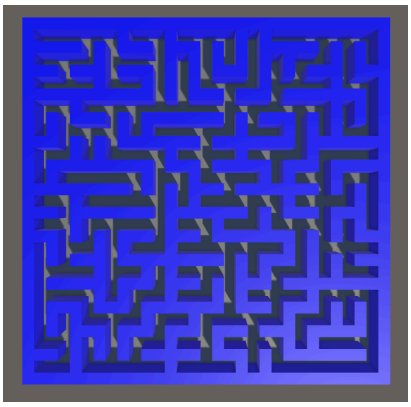
- wobble, which accepts speed and amplitude modifier tags.
- fadein, which accepts the speed tag.
- jitter, which accepts speed and amplitude tags.
- rainbow, which accepts the speed tag.

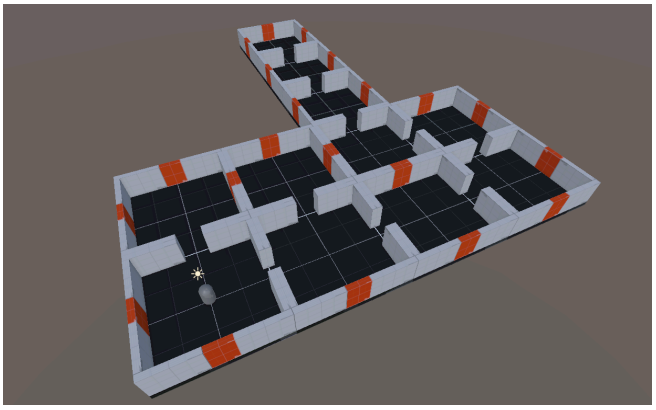
Information about the specific text for tags and modifier default values can be seen in DialogueConstants.

TextEffects is built as a framework to add new TextEffects, in order to do this, other text effects (for example, WobbleText) can be used as a template to copy, all text effects must inherit from TextEffect and must implement GetParamsFromTag(), which parses the required parameters for the new text effect; StopAnimation(), which should clear the effect dictionaries and disable the effect loop and UpdateEffect(), that implements the effect loop, meaning this is where the updating of the text mesh vertices should happen.

Dungeon Generation System

The dungeon generation system is a set of classes that can be used to create either a Binding of Isaac style dungeon, or a Dungeon Crawler style labyrinth.





To use the dungeon generator, the user will have to:

- Create a GameObject in a scene, have it include the DungeonGenerator and RoomManager scripts.
- The RoomManager requires DungeonRooms to generate RoomInstances. DungeonRooms can be created as ScriptableObjects.
    - Each DungeonRoom can be configured to have a Door on the East, West, North or South side.
    - All DungeonRoom assets are automatically loaded from the ResourcesFolder. The exact path is located in DungeonConstants.
    - The DungeonRoom asset needs to be assigned a prefab that contains the actual room visual data, any enemies or logic can be included inside the prefab, the DungeonGenerator and can be expanded to handle them with the OnRoomsInstantiated() event.
- The DungeonGenerator uses the SquaredDungeon class to generate a RoomSizexRoomSize matrix, a different class can be created to similarly generate a Labyrinth using the Labyrinth class instead.
- The SquaredDungeon can be converted into a graph structure (squaredDungeon.GetGraphDescription()) in order to apply graph algorithms to get the shortest path (graph.BFS(int source)).

The dungeon generator can be expanded to include weight in the different rooms by adding a Weight float to each DungeonRoom, then when calling:

```
return ListUtils<DungeonRoom>.GetRandomElement(validRooms);
```

Use the weighted call from ListUtils, as in:

```
GetRandomElement(List<T> list, List<double> weights)
```

To add further logic to each DungeonRoom, references to the new elements must be added to each RoomInstance (similarly to Doors). Then they can be handled through the RoomManager, which in turn is handled through the DungeonGenerator script.

Experience System

The experience system is a small set of scripts that can handle a leveling system in a game. It has the capability to pre-compute a level up map using Linear, Quadratic, Cubic and Parabolic equations. It also allows defining a minimum and maximum level.

To use the system the user will have to add the Experience script to the desired game object, the experience map can be configured through the inspector once added. Bear in mind that since the Experience script is not a singleton, there can be several level maps simultaneously, for example, if creating an RPG game with a party, each party member can have their own level and level map. Experience can be granted through:

```
public void GrantExp(float amount)
```

The game can react to OnExpGain, OnLevelUp and OnMaxLevelReached events, in order to display alerts, play sounds, trigger cutscenes, etc.

The system also allows the creation of ScriptableObjects with type ExperienceMobAsset. These assets could be representative of an enemy in a traditional RPG game, or could be a specific event in any other type of game, they can have an experience value associated to them. The ExperienceMobHandler, a static singleton class that can consume ExperienceMobAssets, will grant the associated experience when triggered. The user can also define ExperienceMultiplierEventTypes, which can modify the amount of experience a specific Mob or Event will grant. The use case could be, for example, an RPG game where the experience enemies will grant will adapt to the current user's level, in order to avoid having to create separate ExperienceMobAssets to accommodate to the 'level inflation', the user might want to define an experience multiplier in order to make all mobs have a x1.5 multiplier.

This system could be expanded, if needed, to use the Saving System (see below) to store the current level and experience amount for a specific Experience instance.

Saving System

The saving system is a set of scripts that allows the storage of dictionaries, lists and alphanumeric variables as .json files in a player's device. To use the system the following steps must be met:

- Add a SaveDataHandler static singleton to a Scene.
- Add the desired variables to the SaveSlot class

```
    public class SaveSlot
    {
        //Add here AS PUBLIC VARIABLES all the data that you want to
save. Delete the Test Dictionary and Int if necessary.
```

```
        public int TestSavedInt;
        public SerializableDictionary<int, string> TestDictionary;
        public List<string> TestList;
        public SaveSlot()
        {
            //Default constructor for Save Data (assign default values
at the start of the game FOR ALL THE VARIABLES YOU WANT TO SAVE).
            TestSavedInt = 0;
            TestDictionary = new SerializableDictionary<int, string>();
            TestList = new List<string>();
        }
    }
```

- To access or modify the saved data the user will have to create new functions in SaveDataHandler in order to Get and Set the specific variable that the game needs to store. Once modified, the SaveData is automatically able to be converted into a json and then decrypted back.

Scene Loading System

The asynchronous loading system allows automatic asynchronous additive scene loading and automatically displays a Loading Screen in the meantime. To use it simply add the static singleton SceneLoader and call:

```
SceneLoader.Instance.LoadScene(SceneIndex, delay);
```

The SceneIndex class is an extender enum that holds static references to all of its possible values, and can cast itself to int or string implicitly. In order to add new scenes to the SceneLoader, simply add the new scenes to the build order in unity, and add the new type to the SceneIndex, bear in mind that the SceneIndex counts with ExampleScenes that can (and must) be removed in order to implement the system in a real game.

Components

Draw 2D Collider Component

The DrawPolygonCollider2D and DrawBoxCollider2D scripts can be added to any 2D sprites with that type of collider and said collider will be displayed even in a packaged build during play. In order to use it just add the component to a sprite with the collider and set the needed LineRenderer color and prefab.

Floater Component

The floater component can be added to any GameObject with a transform and it will be forced to float. The floating motion can be configured to have an offset, speed or amplitude. The floating object can also be set to rotate in a specific angle to a specific speed. The floating object can be stopped and resumed at will.

Health Component

The health component adds a health amount to a GameObject. The health can be increased or decreased, and a public event will be triggered when the health is updated. The following events are available:

- OnHealthUpdated
- OnDamageDealt
- OnHeal
- OnFullHeal
- OnDeath

Interaction Component

The Interaction component is made up of two separate scripts, the Interactor and the Interactable. The Interactor GameObject is the one that moves, when the Interactor entity enters the Interaction Range of an Interactable GameObject, the player will be able to press a specific button or key to trigger the Interact event. An Interactable can be configured so that it is enabled or disabled, it can be configured to only be togglable once, and has a set interacting range, said range is visible through a gizmo in editor mode.

LaunchRigidbody Component

The LaunchRigidBody component is a static stateless class that has two functions to launch a specific rigidbody in the desired direction with the required force. This is an example of one of the LaunchRigidbody functions:

```
        public static void LaunchRigidBody3D(Rigidbody rigidbody,
Vector3 launch, Vector3 power)
```

LookAtCamera Component

The LookAtCamera Component will force any GameObject that it is added to to perpetually look at the main camera. This is useful for 2D Sprites when used in 3D spaces, for example when displaying players' usernames above their characters' heads.

Screen Shake Component

The ScreenShake component allows to produce the screen shake effect in both regular and cinemachine cameras. This component consists of three different static singletons, ScreenShakeFromAnimationCurve2D and ScreenShakeFromAnimationCurve3D and the AnimationCurveSelector. In order to be able to trigger the screen shake effect, the user will have to follow the next steps:

- Add either the 3D or 2D screen shake to the scene.
- Add the AnimationCurveSelector script to the scene.
- Create the desired amount of AnimationCurveAsset ScriptableObjects to the project.
- Add the curves to AnimationCurveType.
- Call ShakeScreen:

```
// Use Main Camera
ScreenShakeFromAnimationCurve3D.Instance.ShakeScreen();
// Or use Cinemachine camera
ScreenShakeFromAnimationCurve3D.Instance.ShakeScreenCinemachine(cinemach
ineCamera, curveType);
```

Snap To Grid Component

The SnapToGrid Component lets the user define a grid size on the x, y and z axes and forces any GameObject that has the component attached to it to adhere to the defined grid specifications. This makes it very easy to build grid based games and levels. Bear in mind that the code is locked so that it is only used while in editor to save computing power, but

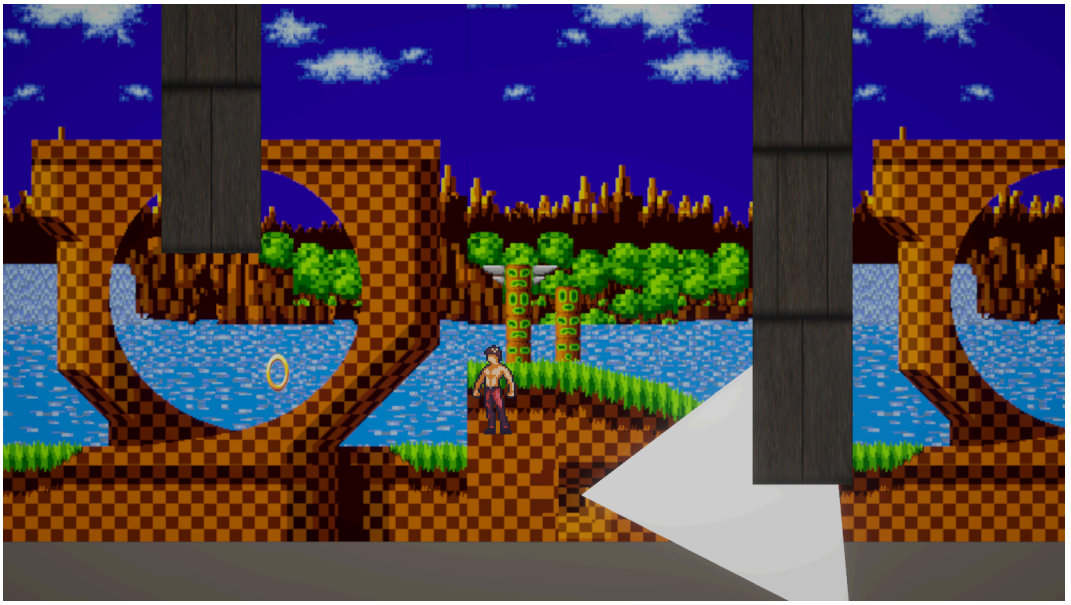
this restriction can be removed by removing the preprocessor directives in the SnapToGrid class in order to make objects adhere to the grid during gameplay.

Character Controllers

Character controllers all use a specific input map and are available and pre-configured through their each specific prefab. All controllers are available in both physics based and regular modes, the physics based ones add force to the rigidbody in the GameObject in order to move the character in different directions, while the regular ones move the GameObject discreetly by adding steps to the desired axes. The forces applied to both types of controllers (speed, jump force…) can all be customized through the inspector. Controls for all controller types can be checked in the MovementInput input asset.

2D Side Scroll

This controller resembles a 2D side scroll platformer like Super Mario Bros or Terraria, it can be added to the scene either by adding the 2DScrollCharacterController or PhysicsBased2DScrollCharacterController. In this case, the physics based controller also allows for wall jumping, while the discrete controller only allows for simple platforming.

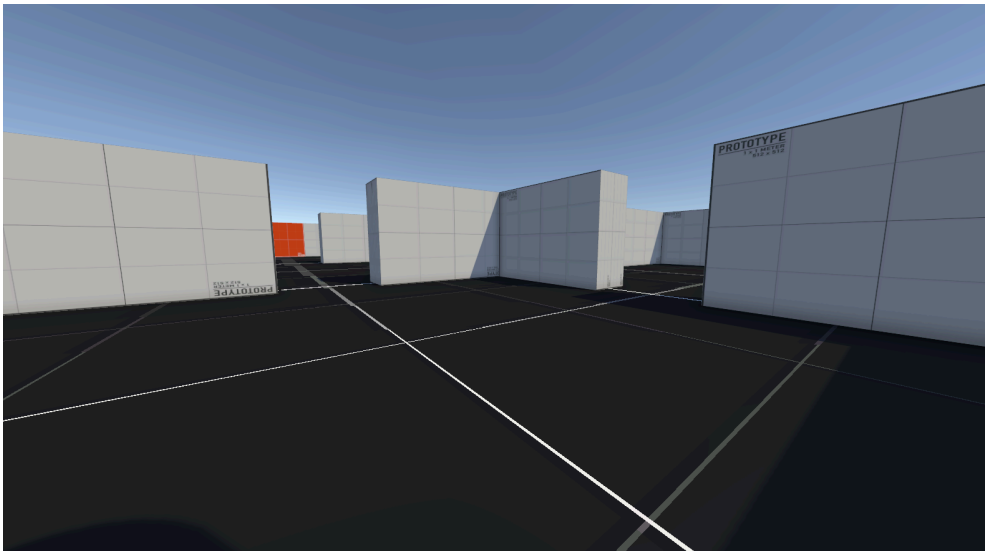


2D Top Down

This controller is based on top down 2D games like Enter the Gungeon or The Legend of Zelda, it can be added to a scene by dragging the TopDownCharacterController or PhysicsBasedTopDownCharacterController prefabs to it. In this case both controllers have the same base functionality, it is just the implementation that differs. The component also includes a ‘pointer’ prefab that points to the mouse position that can function as a gun in games like Enter the Gungeon or AK-XOLOTL.

3D First Person

The 3D First person controller is based on games like What Remains of Edith Finch or Call of Duty, and can be added to a scene by dragging the Simple3DFPSCharacterController or PhysicsBased3DFPSCharacterController prefabs to it. The physics based component allows for running, crouching, dashing, wallrunning and walljumping, whilst the discrete controller can only move, jump and crouch.

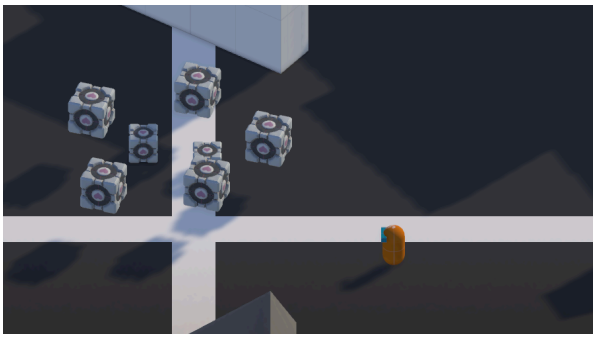


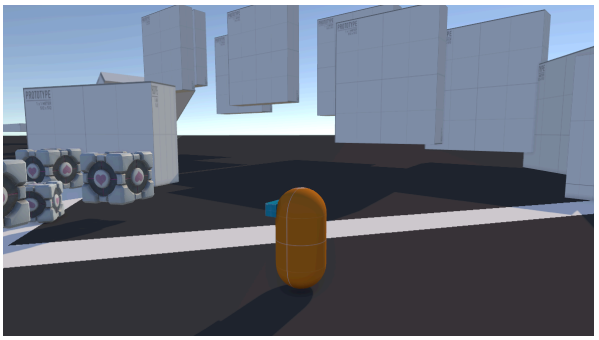
3D Third Person

The 3D third person controller comes with several different camera presets:

- Normal third person - Based on games like Grand Theft Auto V
- Shoulder third person - Based on games like God of War (2018)
- Fixed Y axis 3D top down - Based on retro games like the original Kingdom Hearts for the PS2
- Fixed XY axis 3D top down - Based on games like Tunic
- Free 3D top down - Similar to the ones above, but allows for some vertical movement.
- Isometric 3D top down - Based on games like Project Zomboid

And can be added to a scene by adding Simple3DThirdPersonCharacterController or PhysicsBased3DThirdPersonCharacterController. The only difference between them is that the physics controller can slide after crouching in slopes, and the discrete one can't.





3D Scroll

This controller is based on games like Metroid Dread or Sonic Superstars, the controller can be added to a scene by adding the Simple3DSideScrollCharacterController or PhysicsBased3DSideScrollCharacterController prefabs to it. The only difference between them is that the physics controller can slide after crouching in slopes, and the discrete one can't.