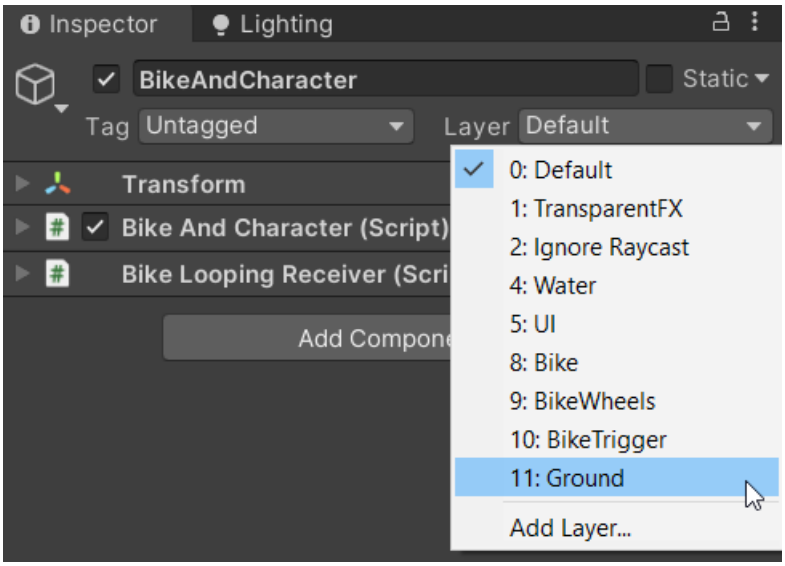
Manual

Table of contents

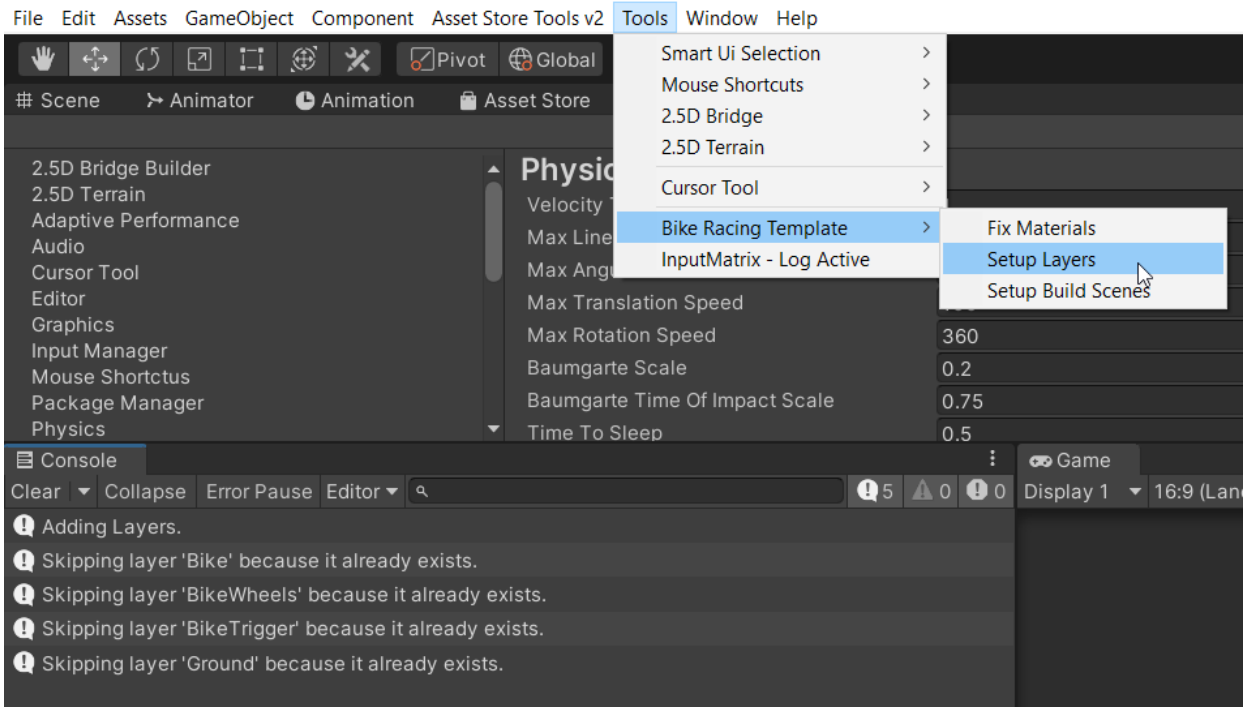
1. Preparations after import

1.1 Checking if the auto-setup worked.

Upon import the template should have added some layers named **Bike**, **BikeWheels**, **BikeTrigger** and **Ground**.



If these layers do not exist then please start the setup manually via **Tools > Bike Racing Template > Setup Layers**. You will see some logs in the console.



If that does not add the layers then please follow the instructions below (1.2 Fixing ...).

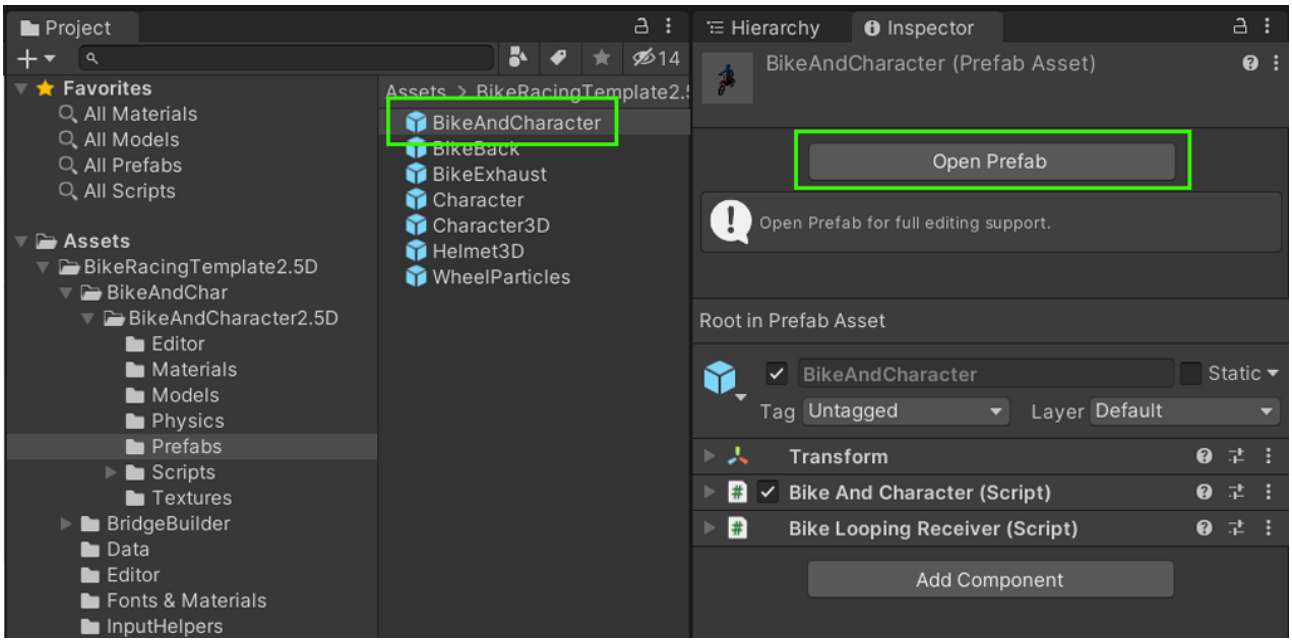
Please proceed with section 2 Understanding the project structure once the layers have been set up.

1.2 Fixing the setup if auto-setup failed

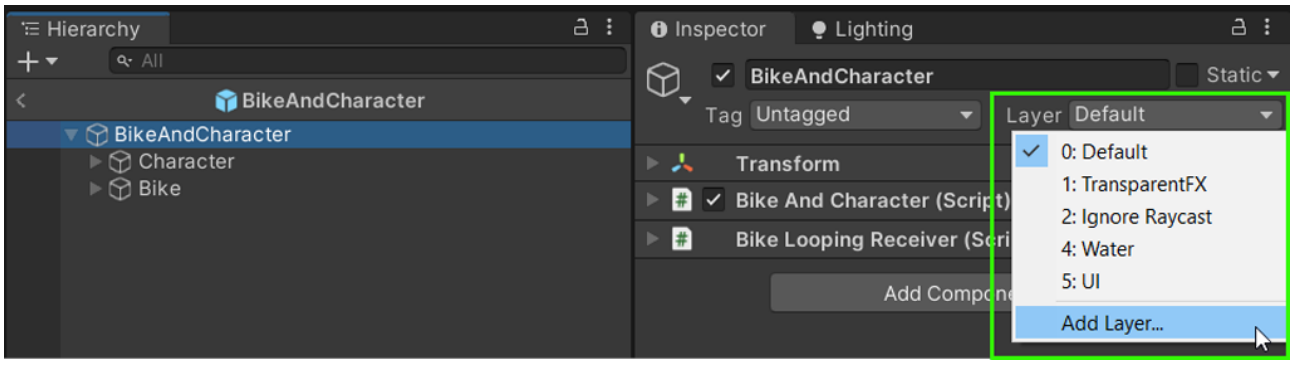
1.2.1 Manual physics layer setup (only required if auto-setup failed).

In order for the objects to properly interact with each other they have to be assigned to layers.

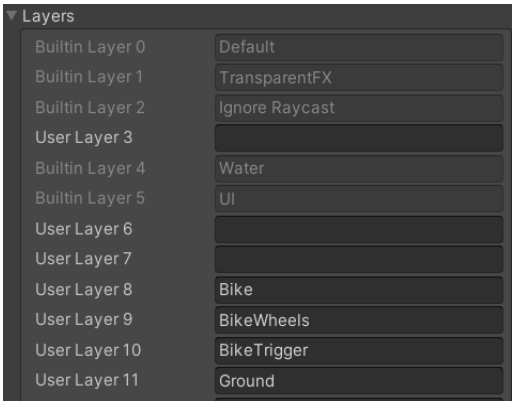
1. Open the „**BikeAndCharacter**“ prefab under **Assets/BikeRacingTemplate2.5D/BikeAndChar/BikeAndCharacter2.5D/Prefabs**.



2. In the Layers section on the right pick „**Add Layer...**“

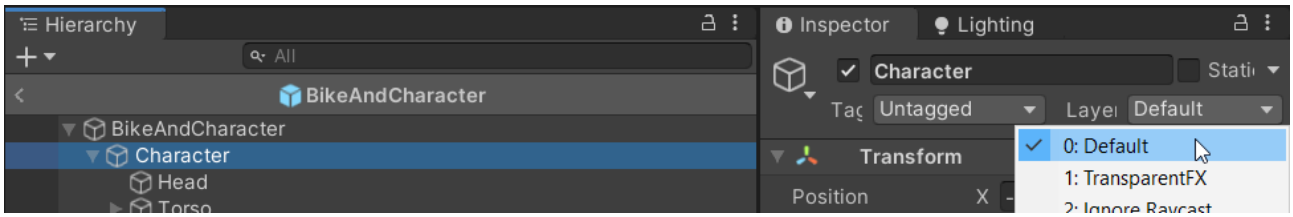


3. Then add four layers named **Bike**, **BikeWheels**, **BikeTrigger** and **Ground**.

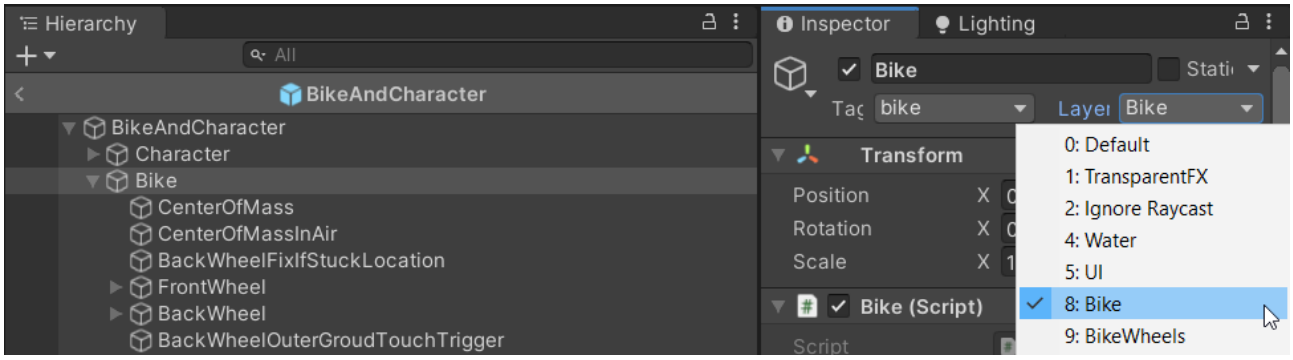


Layer 8 should be Bike
Layer 9 should be BikeWheel
Layer 10 should be BikeTrigger
Layer 11 should be Ground

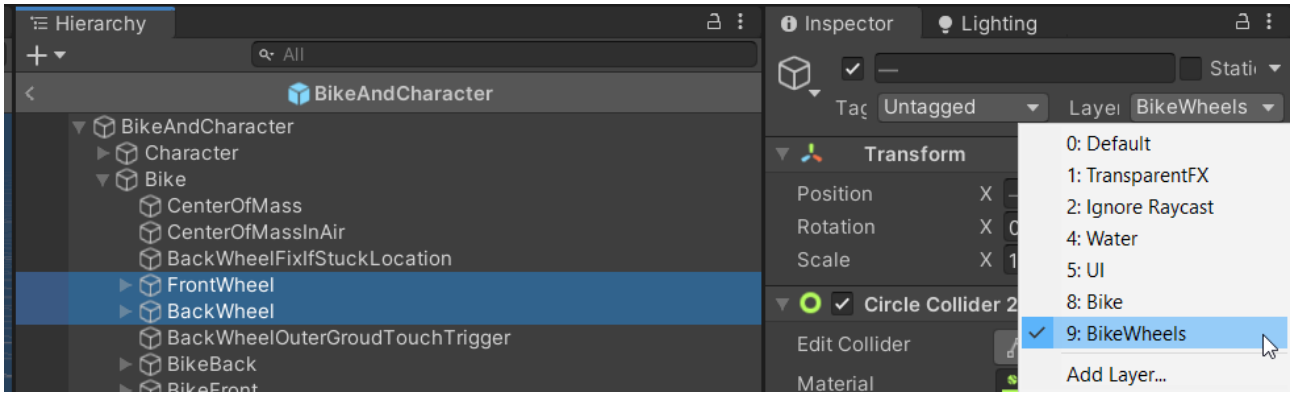
4. Select the Character and set the layer to „**Default**“



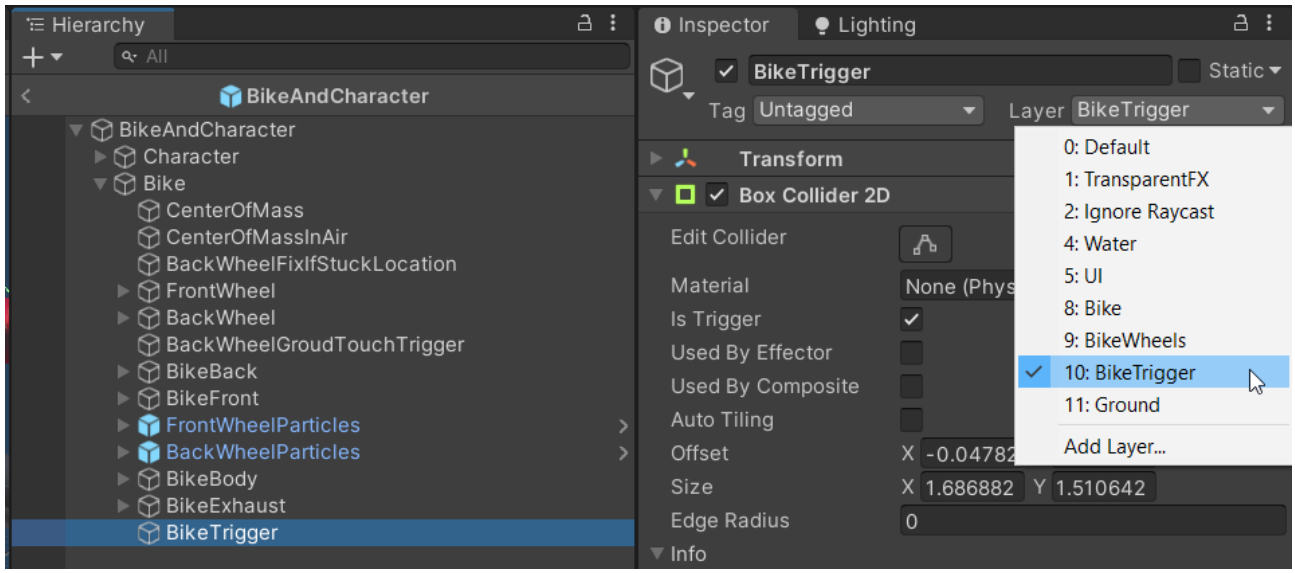
5. Assign the **Bike** layer to the bike (and the children).
If you have assigned the layers to the right numbers in step 3 then these should already be correct. If you had to assign them to differen numbers then now is the time to fix them.



6. Assign the „**BikeWheels**“ layer to the „**FrontWheel**“ and „**BackWheel**“ (and their children).

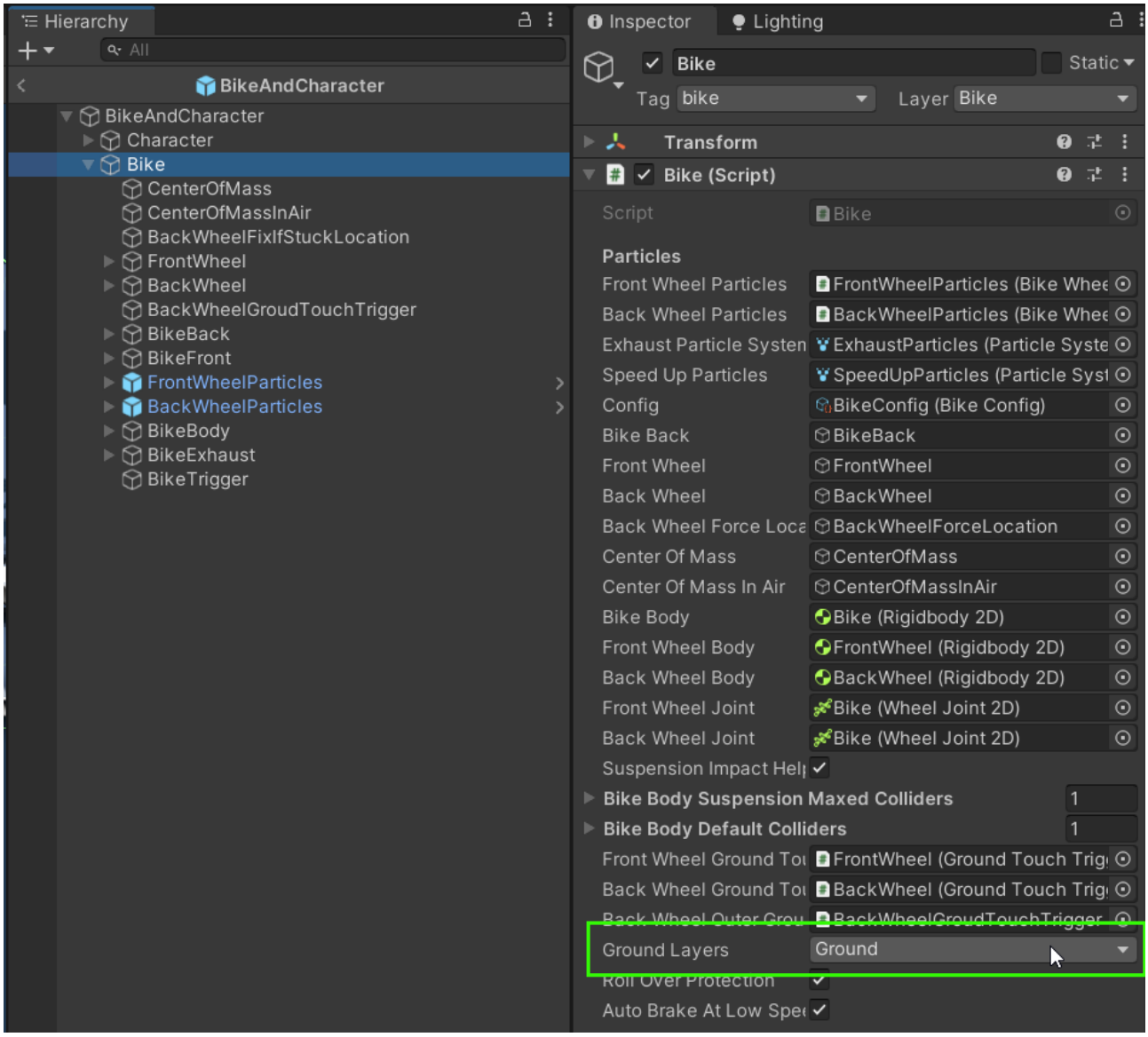


7. Assign the „**BikeTrigger**“ to the BikeTrigger object.



8. The last step within the bike is to let it know which layer will be the ground to drive on. Set the „Ground Layers“ to „Ground“.
NOTICE: the „Ground Layers“ is a multi-select box. Make sure that only „Ground“ is selected.

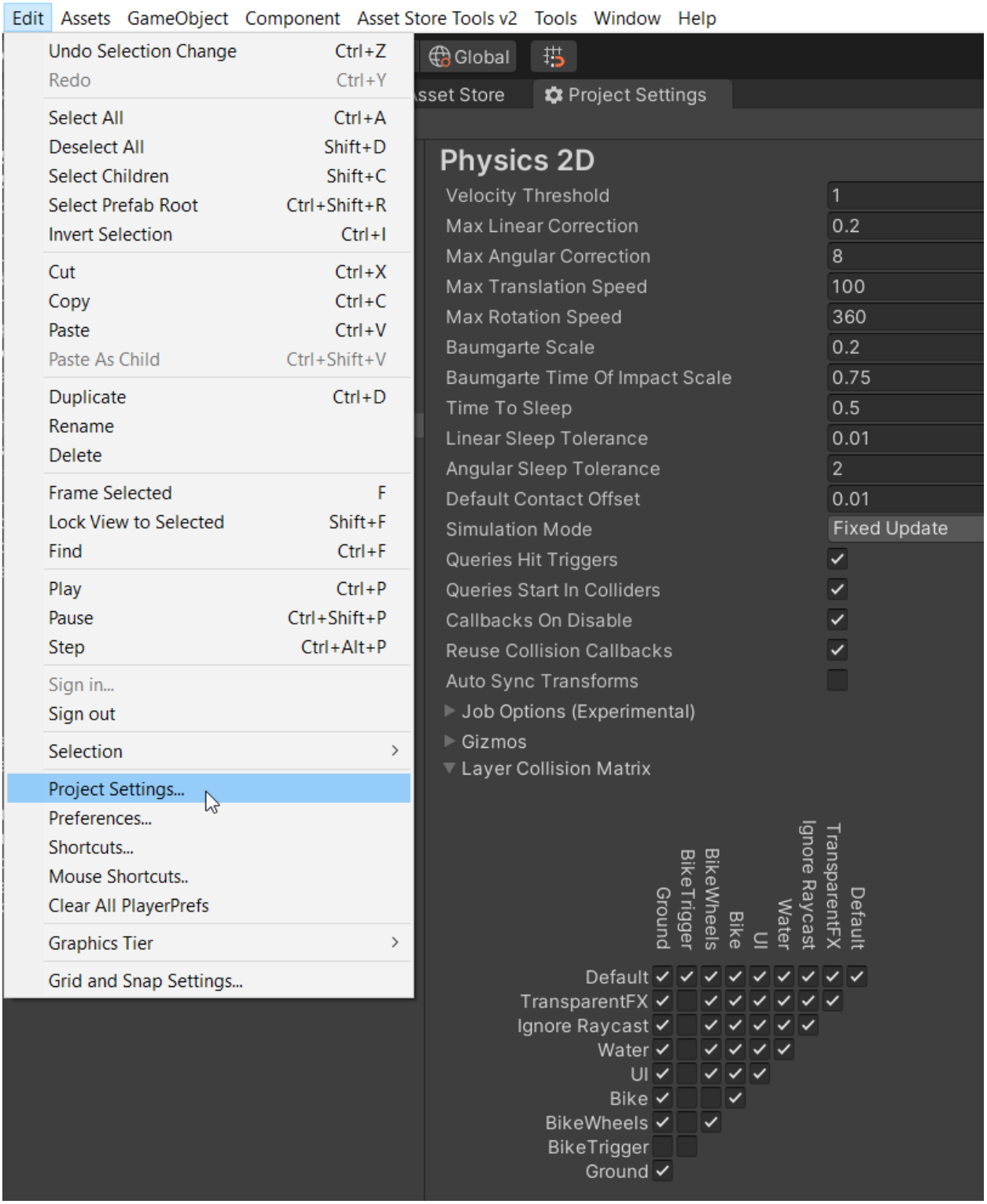
Now the setup work for the bike is done. Save it and proceed with the instructions below.



1.2.2 Configuring collision masks (only required if auto-setup failed)

Upon import the template should have configured the collision layer masks.

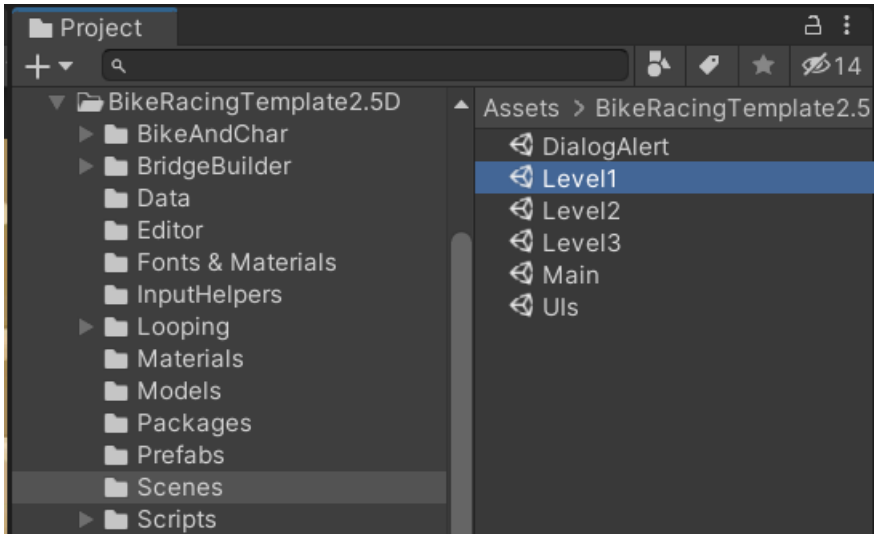
To check it go to „**Edit > Project Settings ...**“ and then compare the collision matrix as shown in the image with your current one. Make sure they match up.
You can try to restart the auto setup under „**Tools > Bike Racing Template > Setup Layers**“. You will see some logs in the console.



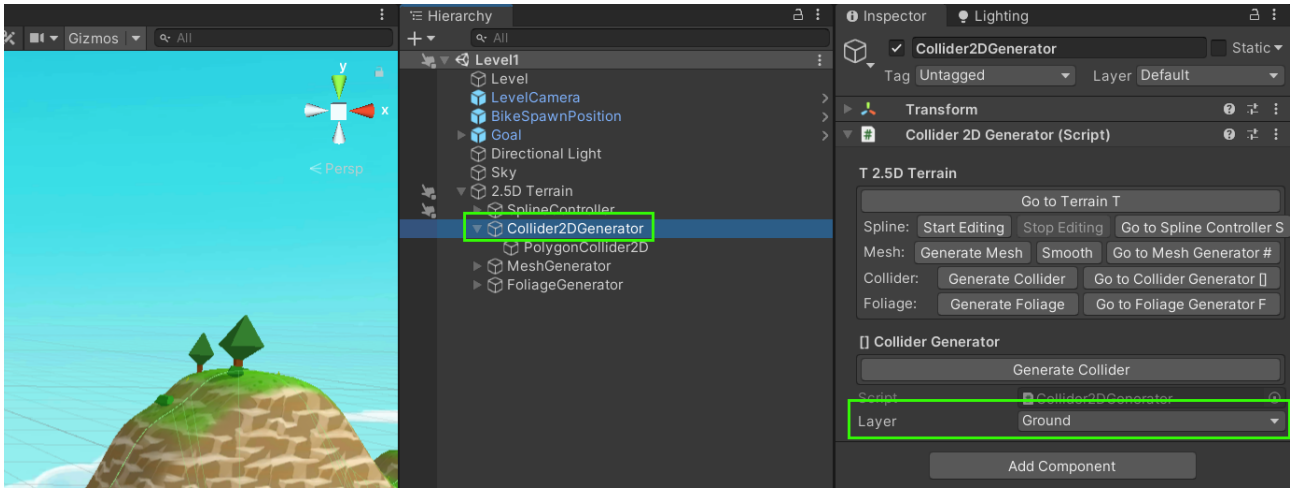
1.2.3 Assigning layers in levels (only required if auto-setup failed)

If you have set up the layers correctly in step 1.1 and 1.2 then the layers in the level scenes should already be configured correctly. Follow to instructions below to check if that's the case.

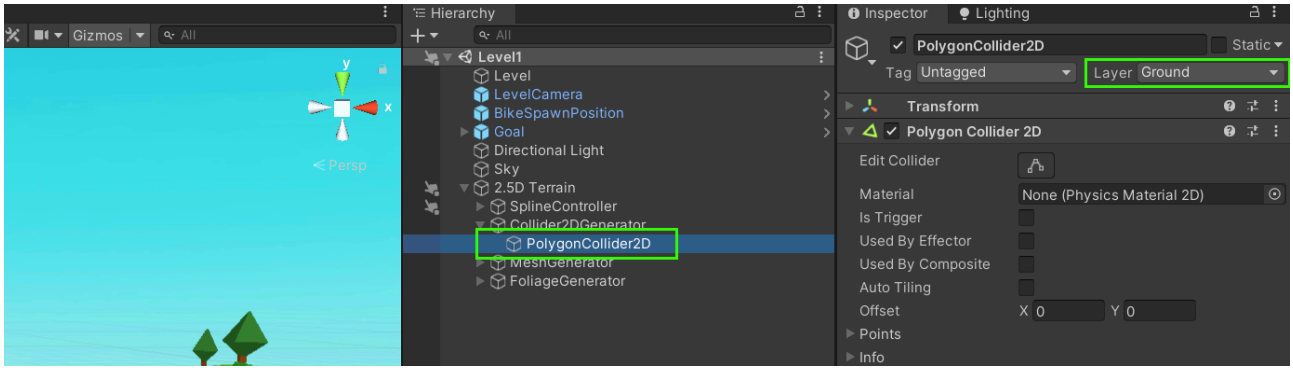
Open the Level1 scene.



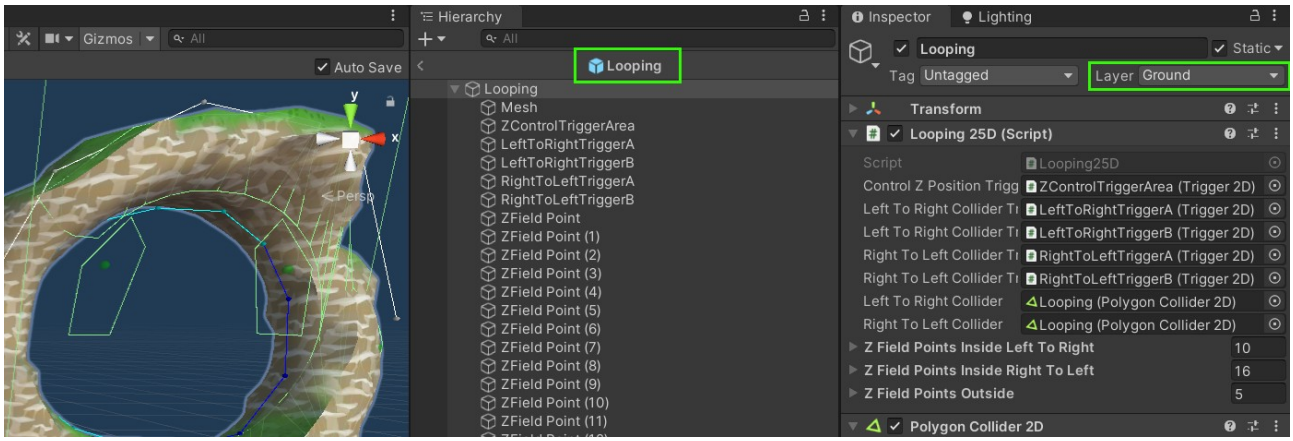
Check if the „Layer" property of the „ColliderGenerator" is set to „Ground".

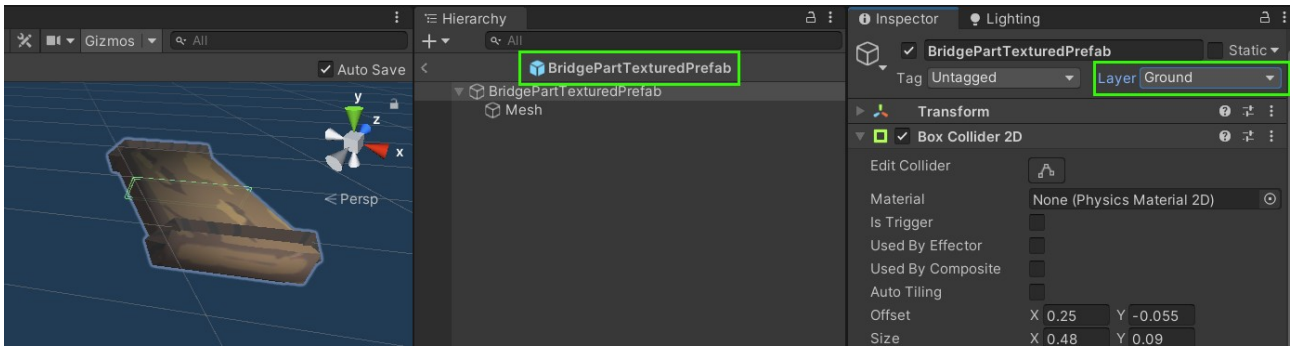


Make sure that the „layer" of the generated PolygonCollider2D is set to „Ground".



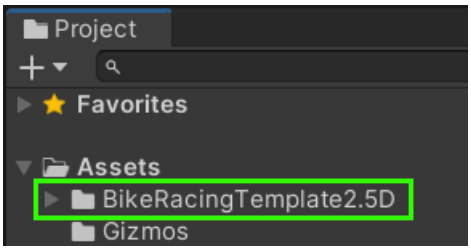
The „Looping“ prefab and the „BridgePartTexturedPrefab“ should also be on the „Ground“ layer.





2. Understanding the project structure

All the files of the template are located under the „Assets/**BikeRacingTemplate2.5D**“ directory.

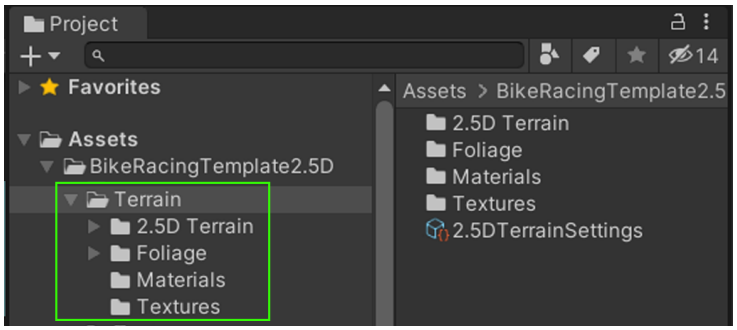


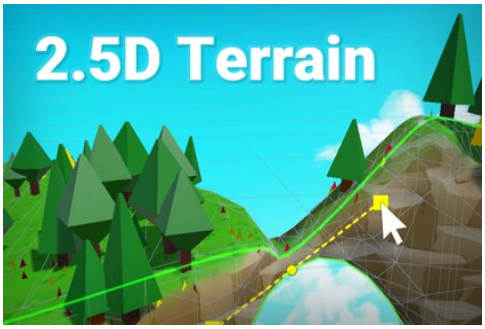
The project makes heavy use of some other Assets which are available in the Asset Store. They are included (as a copy) in this template (you don't need to purchase them). These Assets are located under separate folders within „BikeRacingTemplate2.5D“.

Please notice that these are not 100% copies of their Asset Store counterparts. They have been slightly modified to better integrate into the project.

2.5D Terrain (Asset Store, Manual)

Used for creating the ground terrain in the level scenes (see scenes Level1, Level2, Level3).

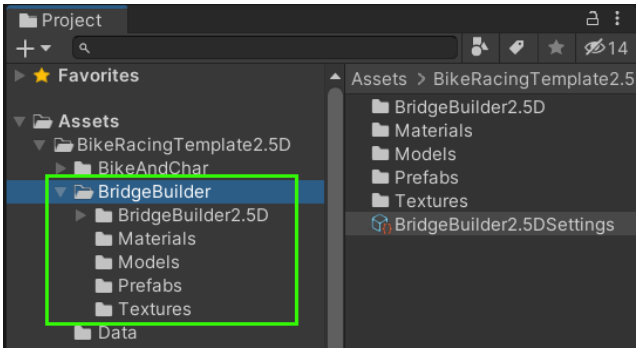




For more details on how to use the 2.5D Terrain please read the manual.

2.5D Bridge Builder (Asset Store, Manual)

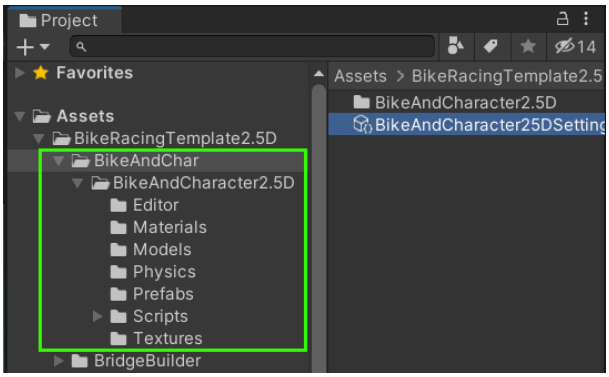
Used for creating bridges in the level scenes (see scene Level2).



For more details on how to use it please read the manual.

2.5D Bike And Character (Asset Store, Manual)

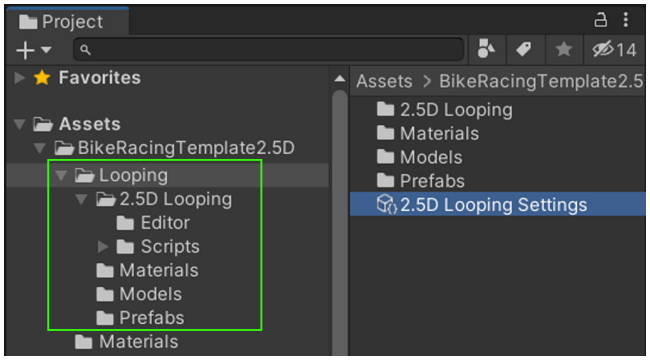
The basis for the BikeAndCharacter Prefab. It contains the bike and the driver.



For more details on how to use them please read the manual.

2.5D Looping (Asset Store, Manual)

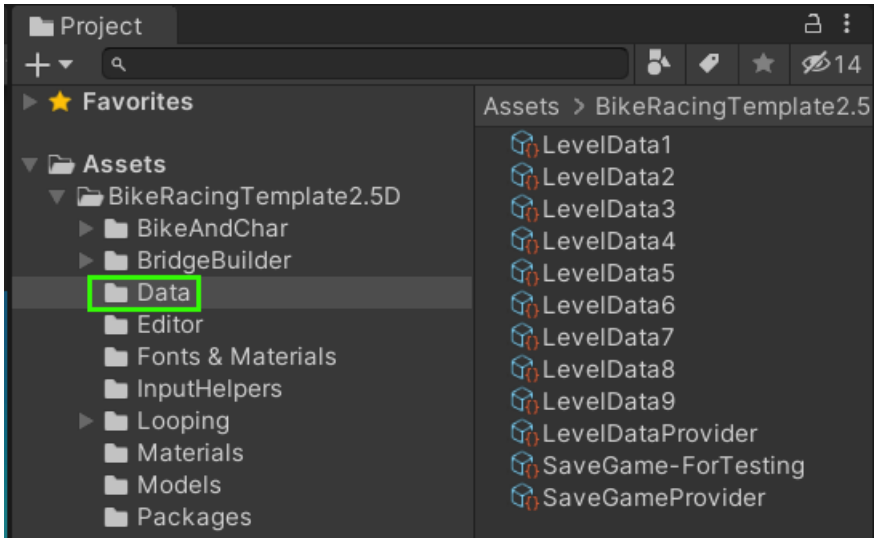
Contains the looping used in the Level3 scene.



For more details on how to use the 2.5D Looping please read the manual.

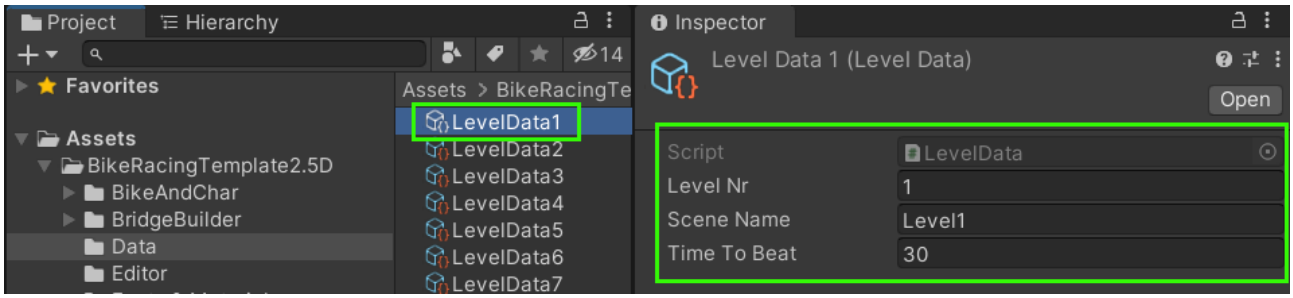
Data

The „**Data**" directory contains Scriptable Objects as data storage.

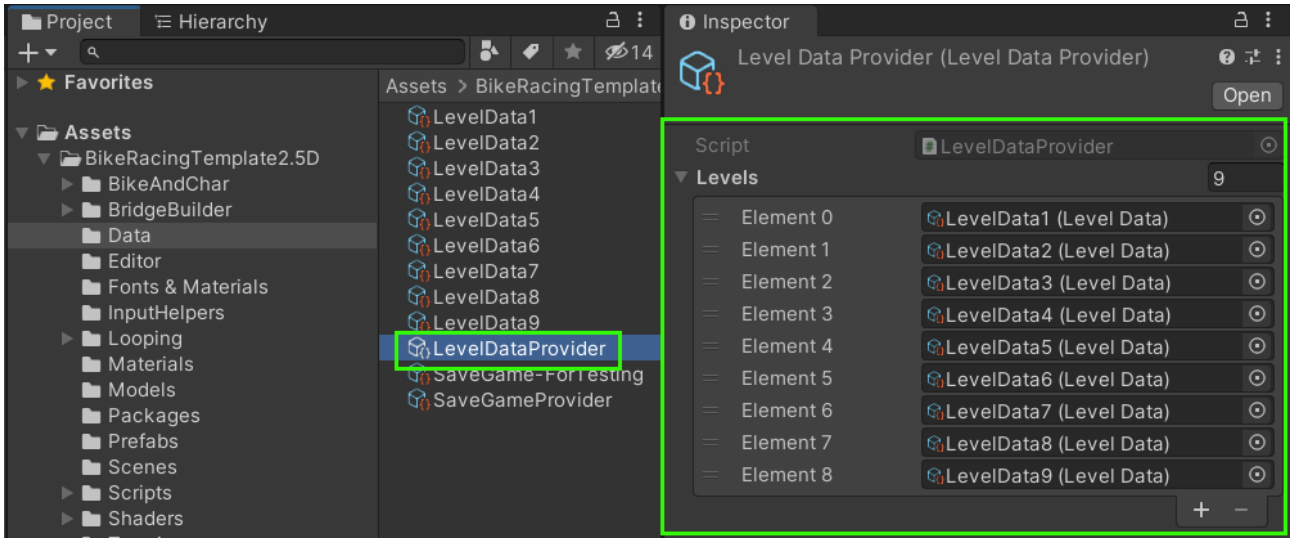


Each „**LevelData**“ object contains data for one level:

- Level Nr - The order in which they are listed and unlocked.
- Scene Name - The scene file to load for this level
- Time To Beat – The time in seconds which the player has to beat to complete the level.

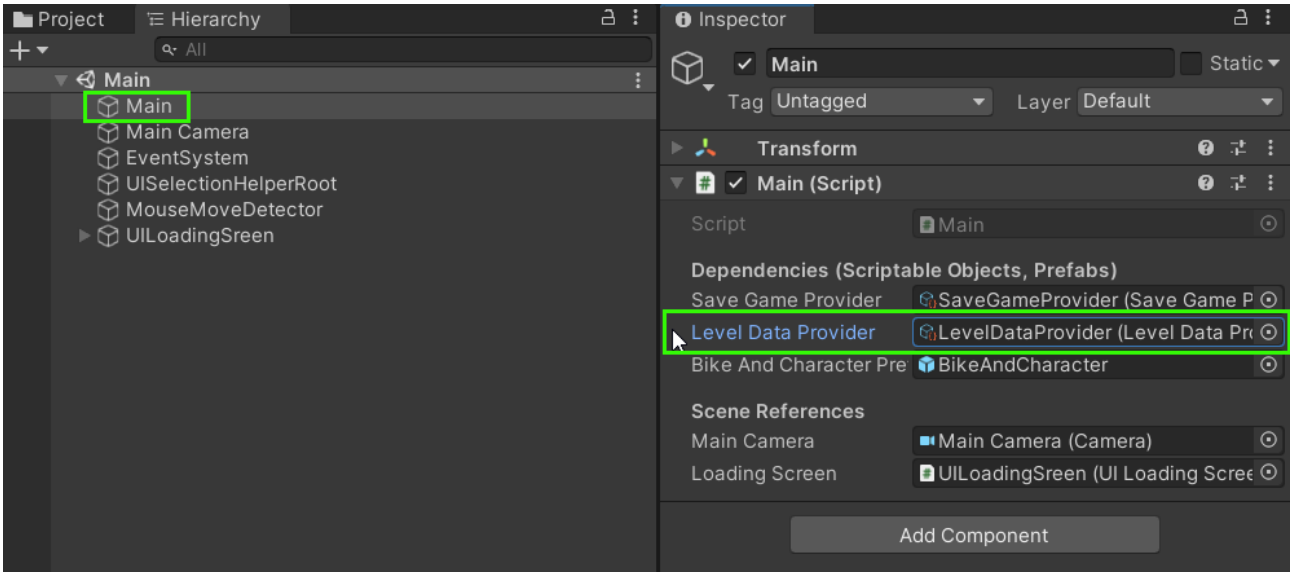


The „**LevelDataProvider**“ contains a list of all the levels. If you add new levels then don't forget to add them here too.



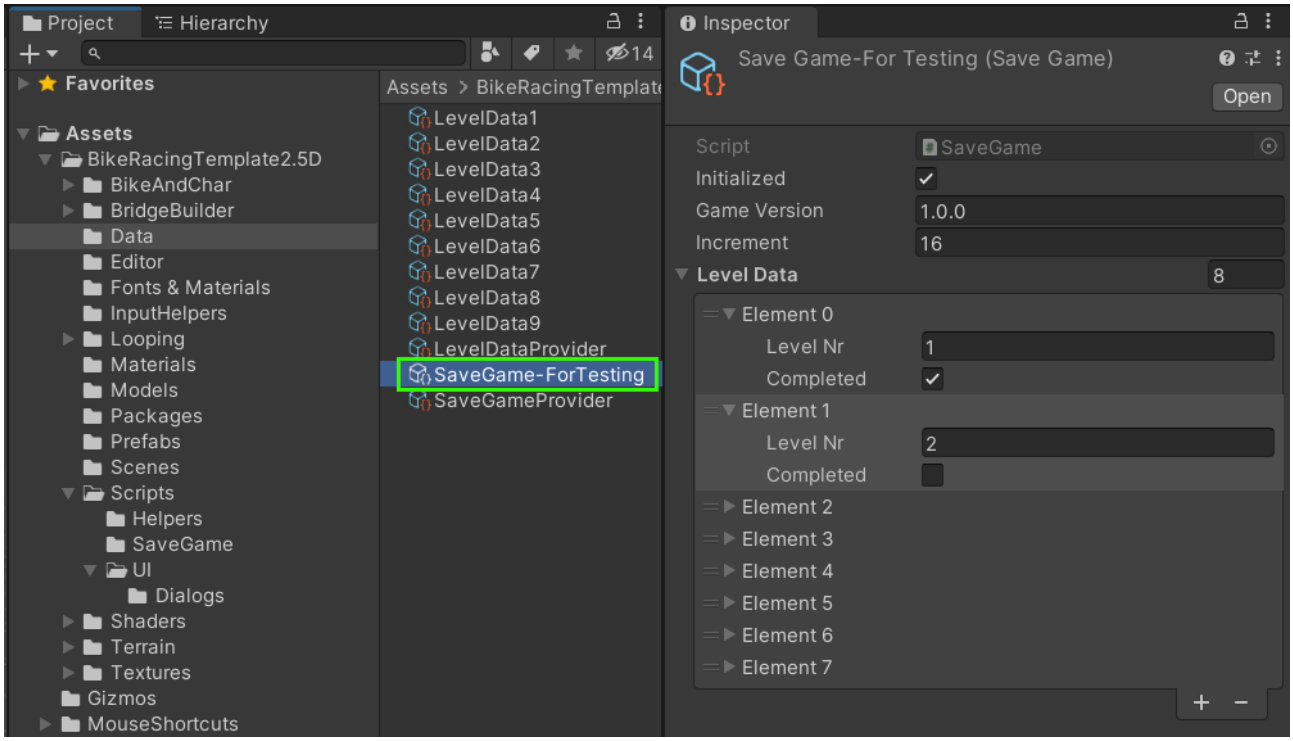
It serves as a global link to the level data (it „provides“ the level data, thus the name).
If a script needs access to the level data then it has to reference the LevelDataProvider in the Inspector. The „Main“ logic in the „Main“ scene does this for example.

Think of it as sort of a very simple dependency injection via the Inspector.

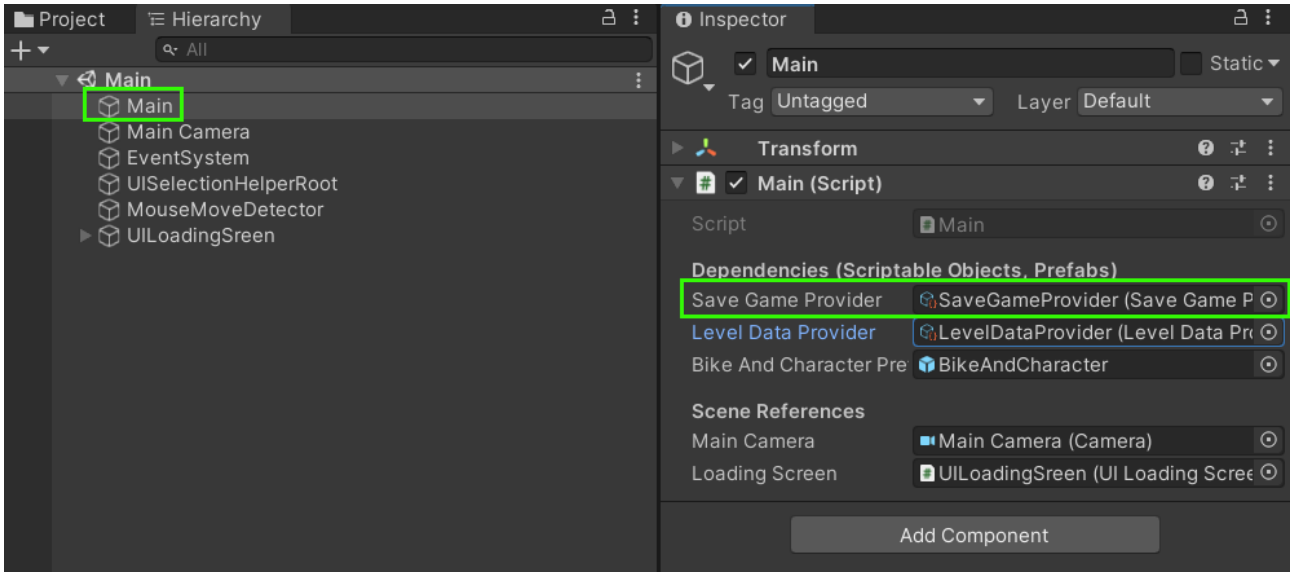


The „**SaveGame**“ object contains savegame data.

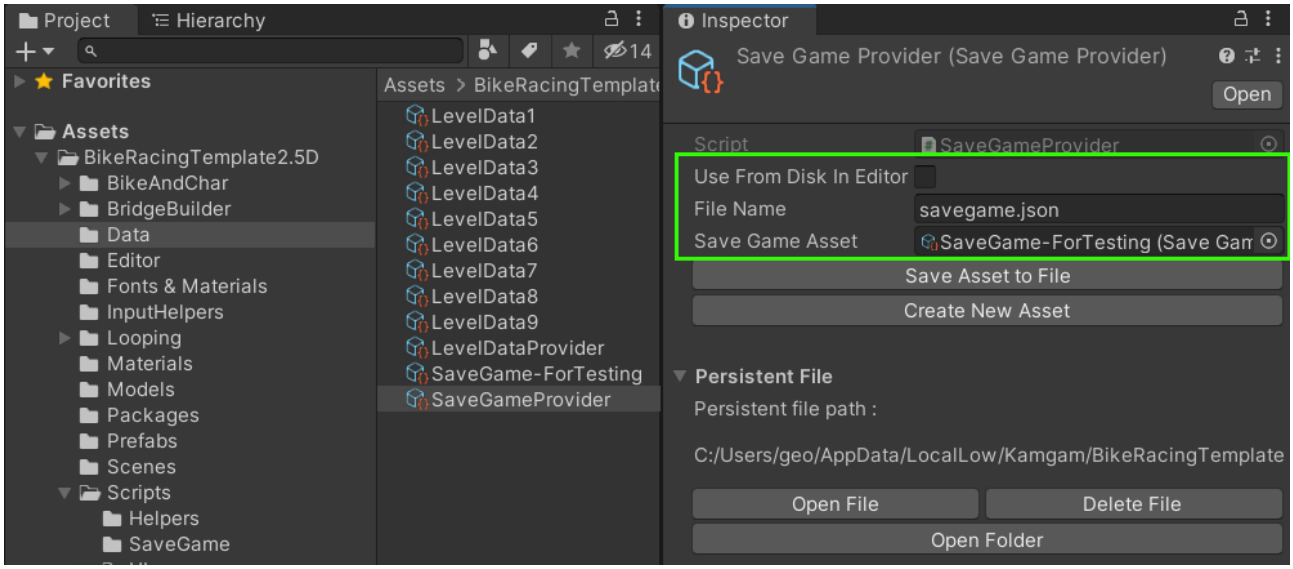
- Initialized - A flag which tells you if this savegame has been initialized (opened at least once).
- Game Version – A copy of the last game version the savegame was opened with (taken from „Main.cs“).
- Increment – An int value which goes up every time the savegame is saved. Useful for comparing the „progress“ of savegames.
- LevelData.LevelNr – The „Level Nr“ links the save game level data to a „LevelData“ object (see LevelDataProvider above)
- LevelData.Completed – Whether or not the player has beaten this level.



The **SaveGameProvider** serves as a global link to the savegame data (it „provides“ the savegame data, thus the name). If a script needs access to the savegame then it has to reference the SaveGameProvider in the Inspector. The „Main“ logic in the „Main“ scene does this for example.



You can choose whether to use a **JSON file** or an **ASSET** for data storage (see „Use From Disk In Editor“).



Depening on what you choose the savegame data exists in different states (A or B).

A) Data is stored in a JSON file:
When the game starts the JSON file will be loaded and deserialized into a runtime-only ScriptableObject (you can not see nor inspect it in the Editor).

Once SaveGameProvider.Save() is called the in-memory ScriptableObject will be serialized back to JSON and stored on disk.

B) Data is stored in an ASSET in the Editor (SaveGame-for-Testing.asset for example):
This is only possible in the Editor (not in builds). The Asset will be used directly by the game and therefore you can see any changes in the Inspector immediately (good for testing).

Once SaveGameProvider.Save() is executed a call to „AssetDatabase.SaveAssets()" will be issued to persist the changes.

- Use From Disk In Editor - You can either use a ScriptableObject Asset (see SaveGame-ForTesting) or a JSON file (see „Persistent File") as your savegame. The ScriptableObject is only available in the Editor. A build always uses the JSON file.

  By default the Editor uses the ScriptableObject linked in the „Save Game Asset" field. If you check „Use From Disk In Editor" then the Editor will also use the JSON file.

  To get quick access to the JSON file use the buttons provided in the „Persistent File" section.

- File Name – The filename of the json file (if the json file is used).

- Save Game Asset – The SaveGame ScriptableObject Asset (if the Asset is used), EDITOR ONLY!

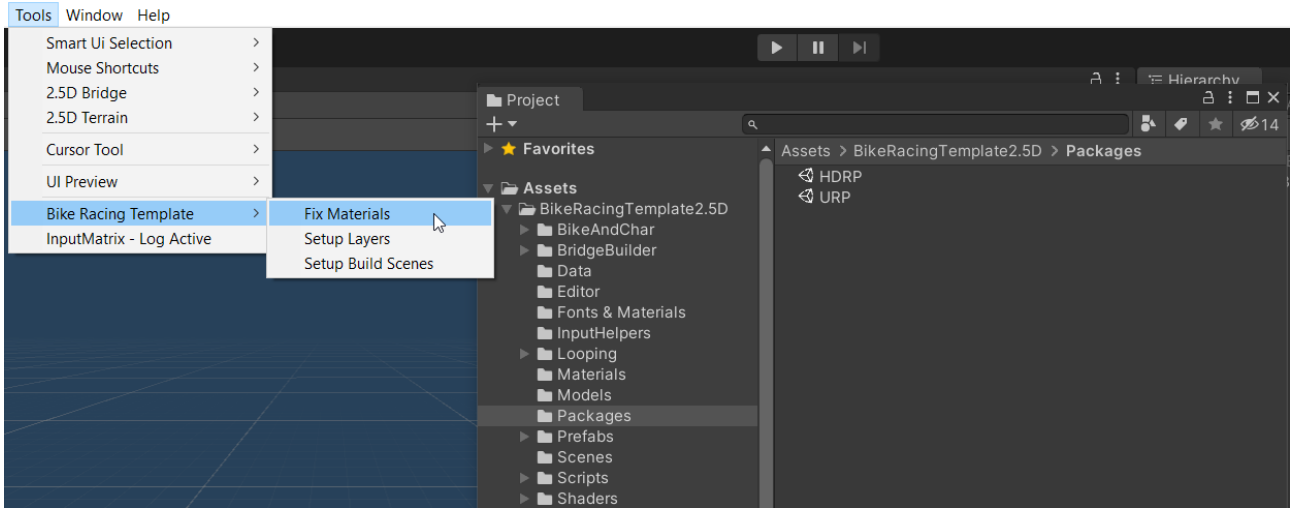
- „Save Asset to File" Button – Copies the data in the linked „Save Game Asset" to the JSON file. Useful if you want to quickly generate a json representation of the savegame.

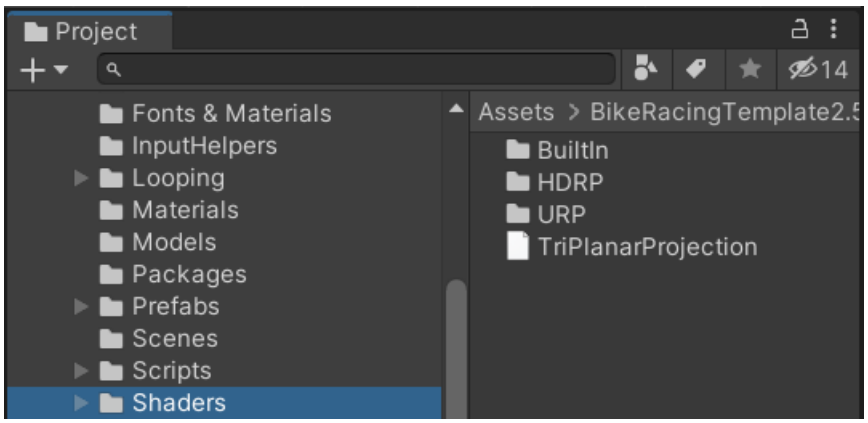
NOTICE: The SaveGame Asset is for EDITOR ONLY use. It can not be changed (saved) in builds. Builds always use the JSON file method to save data.

Packages & Shaders

The packages folder contains two .unitypackage files with materials for the new render pipelines (HDRP and URP/SRP). They will be imported by the template automatically.

If some of the materials still look broken (pink) after import then you can issue a reimport via „**Tools > Bike Racing Template > Fix Materials**".

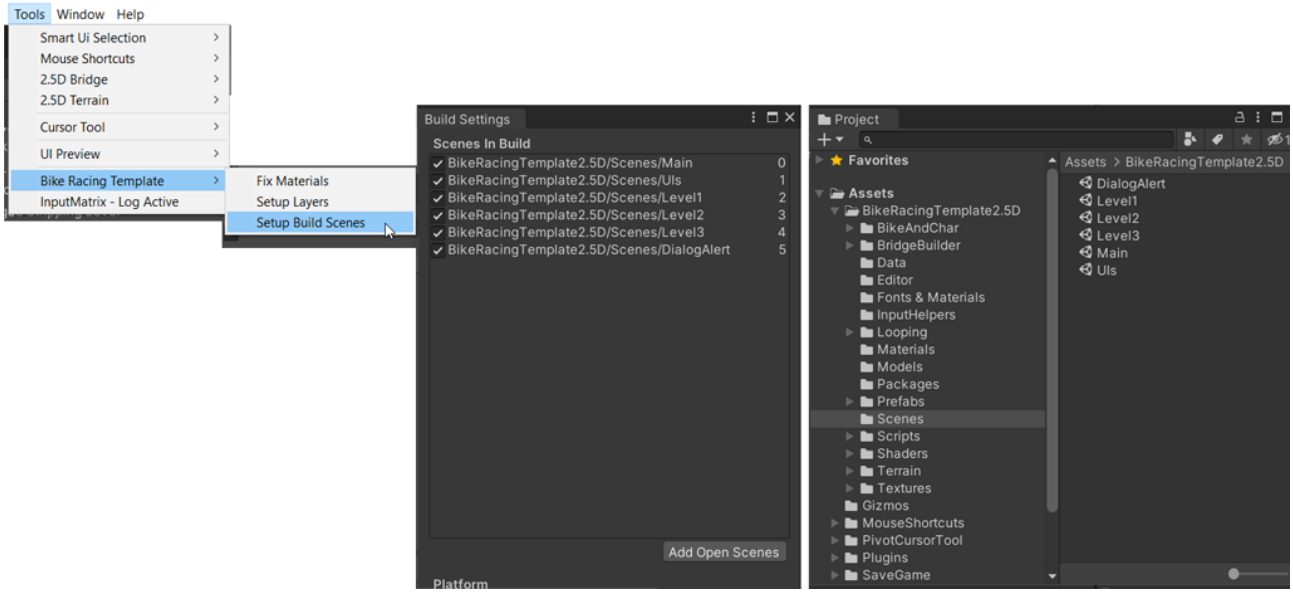




Scenes

After import the template will automatically set the „Scenes in Build". The list should look like in the image below. The Main scene is the starting point for the game and should be loaded first.

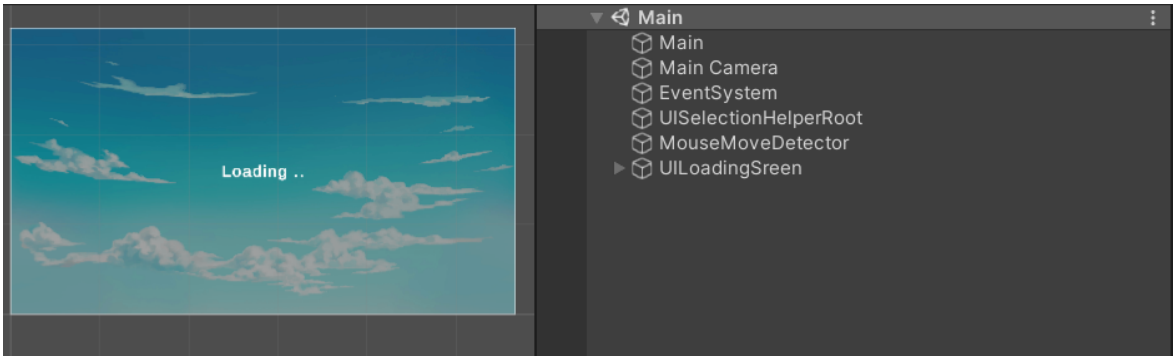
You can trigger the scene setup manually in case it did not work automatically via „**Tools > Bike Racing Template > Setup Build Scenes**".



Scenes in the build:

- **Main** - The entry point of the game. This scene has very few objects in it. It contains a loading screen, the Main logic and some Singletons.

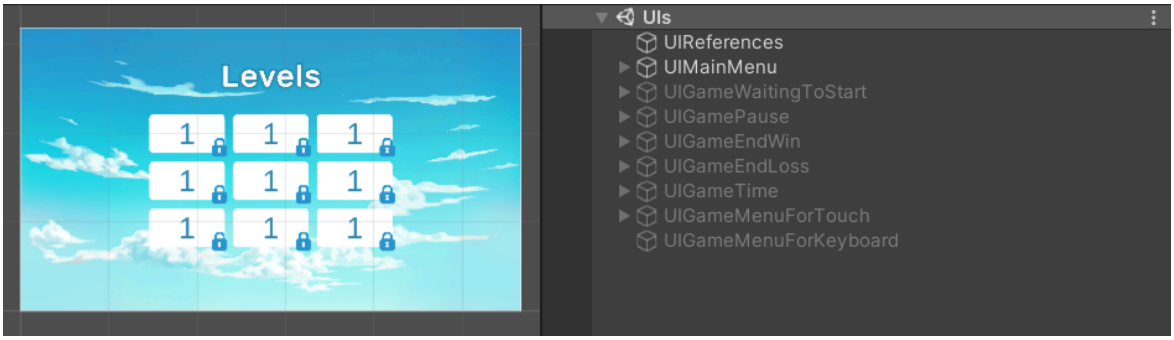
  It is the only thing that's loaded by default. All other scene are loaded additively by the Main Logic. The Main scene is very lightweight so that your game boots as fast as possible.



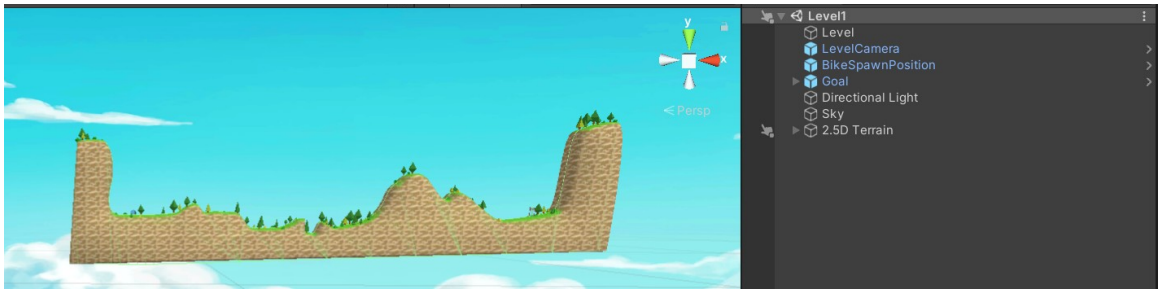
- **UIs** – This scene contains all the UserInterface objects (except for dialogs, more on that later). They are grouped in canvases. Each canvas is more or less just one screen.

All the UIs in here are loaded once and NEVER unload. These are the UIs which are used frequently, thus loading/unloading makes little sense.

UIReferences contains a list of all UIs. If you add a new UI you should add it there too.



- **Level1, Level2, Level3** – The levels contain all the parts needed for each level (surprise). For most levels that's a camera, a light source, a Level logic object, some Terrain, a starting point (BikeSpawnPosition) and and end point (Goal).



- **DialogAlert** – Some UIs are used only rarely and therefore you may not want them to be loaded all the time. The DialogAlert is an example of such a UI.

  It is a scene containing one UI which can be loaded with a static method like „UIDialogAlertAsync.Spawn()" and once it is done it unloads itself and will be gone.

  If you are not familiar with this concept then here is a nice Unite Talk from 2019 which explains it very nicely.

3. Understanding the UI

The UI is organized in canvases. Each canvas is roughly one screen. Though some are only parts of a screen, „UIGameTime" for example.

All UIs derive from the **UIBase** class. You do not have to use this system. It's just there to give you a head start. You will notice that some UIs are derived from UIBaseFade, which is a version of UIBase that fades in and out using CanvasGroups.

The UIBase does some setup for you. For example it registers click callbacks to all its children on Awake().

```
/// <summary>
/// The UIBase is an implementation of IUI which shows/hides the ui immediately.
/// <br /><br />
/// It also adds a base method for Submit/PointerUp events by calling
/// EventTriggerUtils.AddOnClickTriggersToAllChildren(...) in Awake()
/// <br /><br />
/// If you want to get informed of touch or mouse input simply override
/// OnClick(GameObject obj, BaseEventData evt) to get the events.
/// </summary>
public class UIBase : MonoBehaviour, IUI, IInputMatrixReceiver
```

It also uses the **UIInputMatrix** to determine whether or not it should reacto to input.

The UI Input Matrix

The input matrix exists to help the UI to know whether or not it should react to input.

Basically the input matrix is a stack of objects. Only the object(s) at the very top should receive input. To make sense of all the UIs floating around you can assign them a UIStack (a category).

At the start of the program the stacks will be sorted into an array so that the matrix knows who blocks whom from receiving input.

```
public static void Init(Kamgam.InputHelpers.InputMatrix<UIStack> matrix)
{
    matrix.SetMatrix(
        new UIStack?[,]
        {
            { UIStack.Debug       },
            { UIStack.Loading     },
            { UIStack.ModalDialog },
            { UIStack.Menu        },
            { UIStack.Game        }
        }
    );
}
```

Think of it as physical blocks which are stacked and hit by rain from the top (Debug) to bottom (Game). Only those which get wet will receive input.

In the example above the „Debug" stack would block all stacks below. This means if a Debug UI is displayed then no other UI will receive any input (as it probably should be).

If a UIBase is shown then it will automatically register to the InputMatrix and unregister once it is hidden again.

```
public virtual void Show()
{
    ShowImmediate();
    UIInputMatrix.Instance.Push(GetUIStack(), this, AllowParallelInput());
}

public virtual void Hide()
{
    HideImmediate();
    UIInputMatrix.Instance.Pop(GetUIStack(), this);
}
```

The InputMatrix in turn will notify the UIBase if another UI is being pushed above it. That's what the „IInputMatrixReceiver" interface is for.

```
public void OnActivatedInMatrix()
{
    isActiveInInputMatrix = true;
}

public void OnDeactivatedInMatrix()
{
    isActiveInInputMatrix = false;
}
```

This may seem like overcomplicating things because if a UI is above another then naturally you can not click the ui below (if the raycast is blocked). So why these matrix shenanigans?

Think about keyboard or controller input. The selection system in Unity has no knowledge of which UI is currently „in front" and thus any button that is active in the hierarchy could be pressed. We don't want that.

If you don't like this idea then you can of course get rid of it. Simply remove the „IInputMatrixReceiver" and the „UIInputMatrix.Instance.Push()" and „UIInputMatrix.Instance.Pop()" calls from the UIBase and you are freed from the shackles of the Matrix.