Extension Tools For Unity

Online documentation:
https://www.extensiontoolsforunity.com/

Table of contents:

On this page

Intro

Philosophy and Goal

The goal of this Asset is to **improve the development workflow** when developing by implementing some handy features and extensions to Unity.

However none of these features are meant to be a one-stop solution to every situation. As an example, let's take the attributes added in Extension Tools for Unity. I added both the GroupItem and Button attribute, a solution like Odin Inspector offers hundreds of attributes with countless of options. Many of them however you will rarely or never use.

When adding a feature to ExtensionTools I look at these three points.

- **Is it worth it?** (Will people use this feature enough to warrant implementation? Or will this just feel like another feature that will rarely get used and makes the Asset feel bloated?)
- **Easy to use** (The feature should be as easy to use as possible and follow the default Unity Development Workflow, no bizarre and complex setups to get something working)
- **Simplicity is key** (Instead of writing a feature with countless of complex options and features, write a simple one that does exactly what it is meant to do)

I appreciate any feedback and suggestions for future releases!

Getting Started

Install Extension Tools for Unity from the **Asset Store**.

Once you installed the asset you are ready to start improving your workflow! Make sure you implement the correct namespaces when using the asset.

```
using ExtensionTools;
```

In the next chapters you will learn all about how the editor has been improved and how you can take advantage of the new tools and features.

On this page

Moving and transforming

One of the tools which can be used to improve your workflow are **Animations**. Animations are a way to move and transform GameObjects in an easy way. To be able to use these Animations and tween GameObjects you have to start by including the ExtensionTools namespace on top of your file.

```
using ExtensionsTools;
```

Moving and transforming a GameObject

Using animations to move GameObjects are very easy. By default transforms already have extensions to do some basic animations such as a **simple move, scale, and rotate**.

The easing type is an optional parameter which changes the way the animation will be interpolated. Some examples of some of the easing types can be found **here**

```
transform.MoveTowards(targetPosition, time,
ExtensionTools.Animations.EasingType.SmoothInOut);
transform.RotateTowards(targetPosition, time,
ExtensionTools.Animations.EasingType.Linear);
transform.ScaleTowards(targetScale, time,
ExtensionTools.Animations.EasingType.ElasticInOut);
```

Additionally you can use a **parabolic movement** to simulate for example a grenade throw or a jump:

```
Vector3 upDirectionandHeight=Vector3.up*5f
transform.MoveParabolicTowards(targetPosition, upDirectionandHeight, time);
```

Playing Custom Animations

Instead of using the built-in extensions, you can also use the **PlayAnimation** Method to play an Animation on your transform such as the **Shake Animation** in which the first two arguments are the starting position and starting rotation.

```
using ExtensionsTools.Animations.Presets;

...

transform.PlayAnimation(new
EShakeAnimation(transform.rotation,transform.position,rotationShakeAmount,positionShakeA
```

Stopping Animations

To cancel these animations all you have to do is call:

```
transform.CancelAllAnimations();
```

In the next chapter you will see how you can create your own custom animations to play.

Creating custom animations

Creating custom animations to tween GameObjects is very easy to do. You start by creating a new class which inherits from EAnimation.

```
using UnityEngine;
using ExtensionTools.Animations;
public class ECustomAnimation: EAnimation
{
        public ECustomAnimation()
        {
            m_OverrideRotation = false;
            m_OverrideScale = false;
            m_OverridePosition = false;
        }

        public override Vector3 GetPosition(float animPercentage)
        {
            return Vector3.zero;
        }

        public override Quaternion GetRotation(float animPercentage)
        {
            return Quaternion.identity;
        }

        public override Vector3 GetScale(float animPercentage)
        {
            return Vector3.zero;
        }
}
```

In the constructor you decide what this animation will affect. In this example we will create an animation which moves from point A to B in a waving pattern, so we only want the position to be changed.

```
m_OverridePosition = true;
```

To be able to know where this animation starts and ends we need to pass these values through the constructor, additionally we also want to know the height of the waves.

```
Vector3 StartPoint;
Vector3 EndPoint;
float WaveHeight;

public ECustomAnimation(Vector3 start,Vector3 end,float waveheight)
{
        StartPoint = start;
        EndPoint = end;
        WaveHeight = waveheight;

        m_OverridePosition = true;
        m_OverrideRotation = false;
        m_OverrideScale = false;
}
```

Now we can use these values in our **GetPosition** method to return the animated position.

```
public override Vector3 GetPosition(float animPercentage)
{
        float WaveScale = 10f; //This could also be a variable

        Vector3 interPolatedPosition = Vector3.LerpUnclamped(StartPoint, EndPoint,
animPercentage);
        interPolatedPosition += Vector3.up * WaveHeight * Mathf.Sin(animPercentage *
WaveScale); //We return the sine wave and add it to the interpolated position;

        return interPolatedPosition;
}
```

Our animation is finished! Now we can play the animation like this

```
transform.PlayAnimation(new ECustomAnimation(transform.position, targetPosition,
waveHeight),time);
```

Using easings

Easings allow us to change the way we interpolate in our animation. Adding easing to our previous animation is very easy to do. First we add an easingType as a parameter in our constructor.

```
    EasingType EasingType;

    public ECustomAnimation(Vector3 start,Vector3 end,float waveHeight,EasingType
easingType=EasingType.SmoothOut)
    {
        StartPoint = start;
        EndPoint = end;
        WaveHeight = waveHeight;
        EasingType = easingType;

        m_OverridePosition = true;
        m_OverrideRotation = false;
        m_OverrideScale = false;
    }
```

Now we can go back to our **GetPosition** method and change the animPercentage like this

```
public override Vector3 GetPosition(float animPercentage)
{
        float WaveScale = 10f; //This could also be a variable

        animPercentage = Easings.Evaluate(animPercentage, EasingType); //We apply
easing

        Vector3 interPolatedPosition = Vector3.LerpUnclamped(StartPoint, EndPoint,
animPercentage);
        interPolatedPosition += Vector3.up * WaveHeight * Mathf.Sin(animPercentage *
WaveScale); //We return the sine wave and add it to the interpolated position;

        return interPolatedPosition;
}
```

Debug Camera

The goal of the Debug Camera is to be used as a **flying camera to quickly look and fly around** during playmode without this interfering with the game.

The **recommended way to activate the Debug Camera is by just switching the InputMode** because it allows you to check the inputmode to see whether we are currently debugging and prevent the character from moving when controlling the debug camera.

Change to the Debug mode can be done using the Console or by using

```
InputMode.SetInputModeDebug();
```

Additionally if you choose not to use the InputMode approach you can call these functions

```
using ExtensionTools.Camera;

...

DebugCamera.INSTANCE.Activate();
DebugCamera.INSTANCE.Deactivate();
```

Extensions

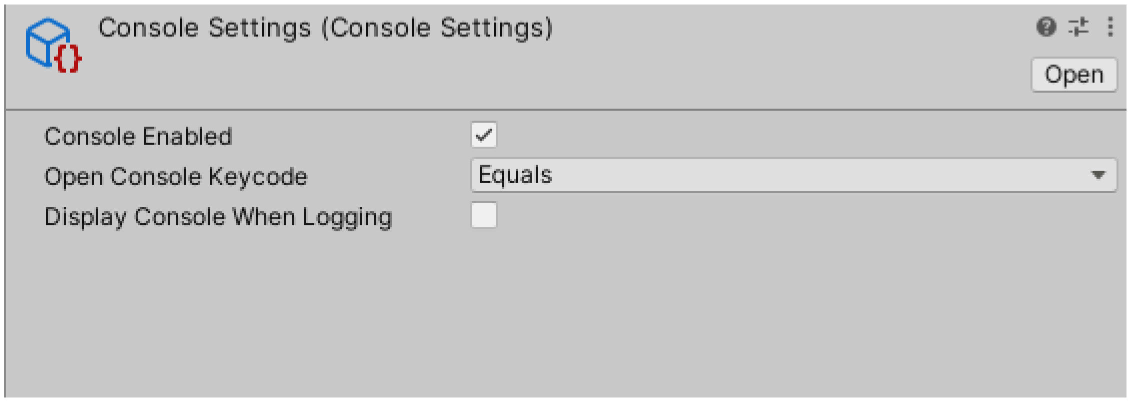
The Camera class has a few extensions

```
using ExtensionTools.Camera;

...

camera.ScreenShake(shakeTime,shakeAmount);
camera.InterpolateFoV(targetFoV,time);
camera.IsEditorSceneCamera();
camera.IsMainCamera();
```

Opening Console and Settings

To open the console we first have to make sure it's enabled. To do that go to Window>Extensions Tools>Console Settings.

Now you will see some options in the inspector.



Console Enabled ✓
Open Console Keycode: Equals
Display Console When Logging ☐

Console Enabled

Simply enables or disables the Console.

Open Console Keycode

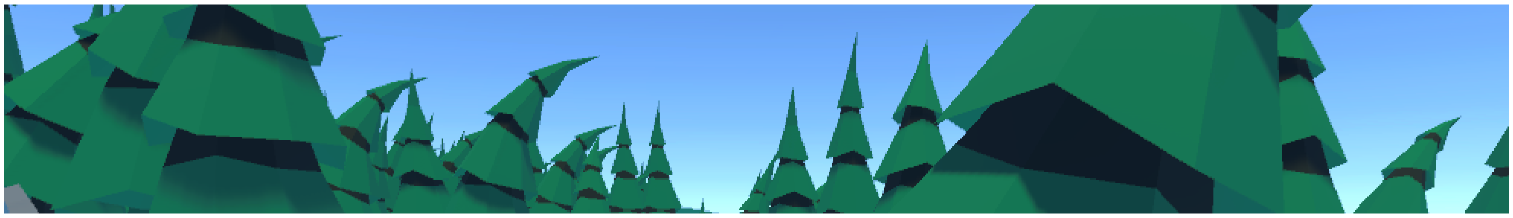
The keycode used to open and close the Console.

Display Console When Logging

Should the console display a dissapearing log when logging even when the Console is not open.

Opening the Console

Once you enable the Console and press the assigned Key while in Playmode the console should open. From here you can use all the available commands (use **help** to get a list).



Custom commands

Creating custom commands is very easy to do. All you have to do is create a new class which inherits from Command.

```
using ExtensionTools.Console.Commands;
using ExtensionTools.Console.Commands.Parameters;
using ExtensionTools.Console.Suggestions;
public class SetTagCommand : Command
{
    public override string Execute(object[] parameters)
    {
        throw new System.NotImplementedException();
    }

    public override string GetCommand()
    {
        //returns the actual command
        return "settag";
    }

    public override string GetHelpString()
    {
        //returns the string displayed when using the command incorrectly
        return "Set the tag of a GameObject. >settag \"gameObjectName\" \"tag\"";
    }
}
```

Now we have to make sure you can pass parameters to this command, to do this we create the constructor and assign the parameter array to the base constructor.

We assign a ParameterGameObject for the GameObject and ParameterString for the Tag. As you can see we also added a GameObject Suggestion Window to the GameObject Parameter. This will display a popup with all GameObjects in the game to choose from.

IMPORTANT! When using parameters with white space, You should always put the parameter between quotes (for example: >settag Main Camera Untagged should be: >settag "Main Camera" Untagged)

```
public class SetTagCommand : Command
{
    public SetTagCommand():base(new Parameter[] { new ParameterGameObject(new
GameObjectSuggestionWindow()), new ParameterString() })
    {

    }
...
}
```

We are now ready to actually execute the command, so in the **Execute** Method we write:

```
    public override string Execute(object[] parameters)
    {
        GameObject go = parameters[0] as GameObject;
        string tag = (string)parameters[1];

        go.tag = tag;

        return "Set tag for " + go.name + " to " + tag;
    }
```

Custom parameters

Sometimes when creating custom commands you might want to use a custom parameter as well.

By default these are the parameters already supported:

- ParameterBool
- ParameterEnum
- ParameterFloat
- ParameterGameObject
- ParameterInt
- ParameterScene
- ParameterString

With these you can do pretty much everything, let's say we want to change the layer of a GameObject. We could easily just use the ParameterString however this would also allow for non-existing layers to be parsed.

So let's create our own ParameterLayer instead.

```
using ExtensionTools.Console.Commands.Parameters;
public class ParameterLayer : Parameter
{
    public override bool TryParse(string parameter, out object parsedObject)
    {
        int Layer = LayerMask.NameToLayer(parameter.Replace("\"",""));
        parsedObject = Layer;

        if (Layer > -1) //Check if it exists, Parsing successful
        {
            return true;
        }
        return false;//Parsing not successful
    }
}
```

Make sure you remove the quotes from strings which might have white space!

As you can see not much is needed to create a custom Parameter. However we can also add a ConsoleSuggestionWindow which popups and gives us a list of all the layers to choose from!

Custom ConsoleSuggestionWindow

Adding a SuggestionWindow for the Layers is simply done by creating a new class and inheriting from ConsoleSuggestionWindow.

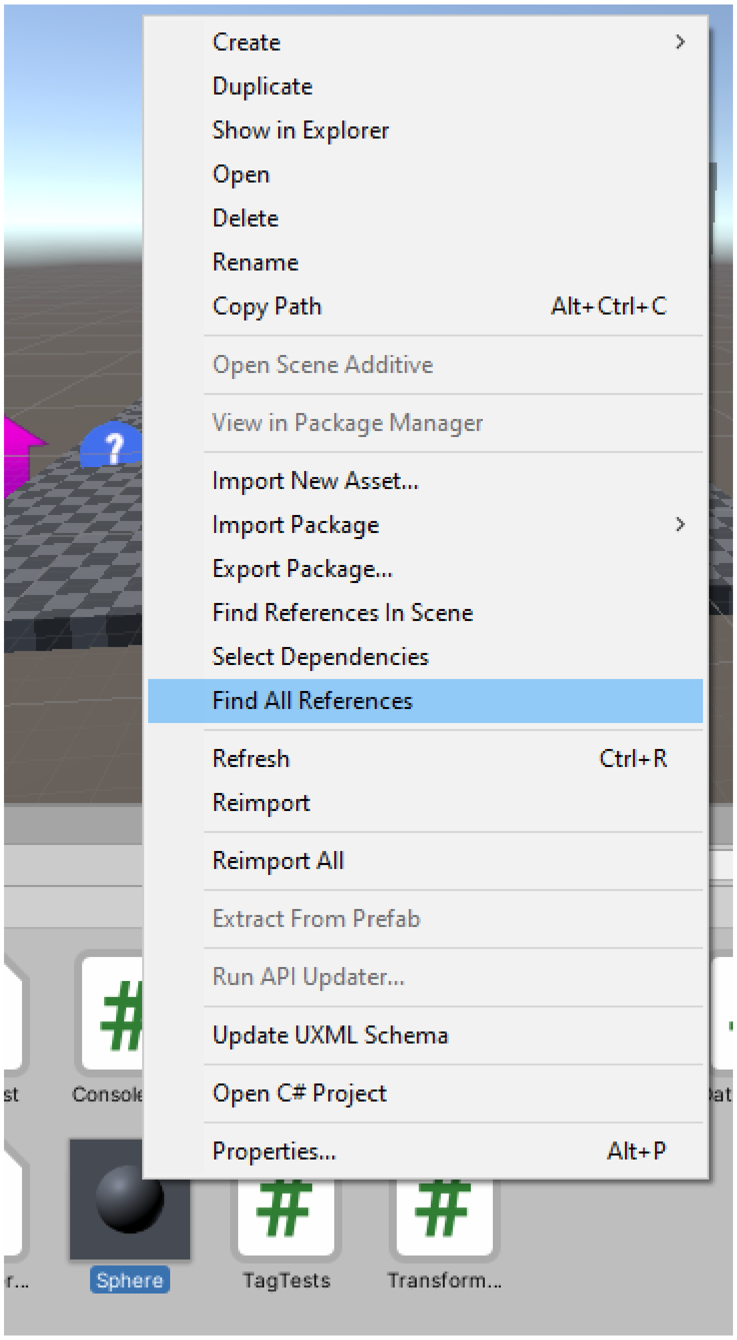
```
using System.Collections.Generic;
using UnityEngine;
using ExtensionTools.Console.Suggestions;
public class LayerConsoleSuggestionWindow : ConsoleSuggestionWindow
{
    List<string> m_Suggestions = new List<string>();
    public override void InitializeWindow()
    {
        m_Suggestions.Clear();
        //Add all layers
        for (int i = 0; i <= 31; i++) //unity supports 31 layers
        {
            var layerName = LayerMask.LayerToName(i);
            if (layerName.Length > 0) //Get length of the layername
                m_Suggestions.Add("\""+layerName+"\"");
        }
    }

    protected override List<string> GetSuggestions(string currentString)
    {
        return m_Suggestions;
    }
}
```

Now we can go back and add this suggestion window to the constructor of our ParameterLayer.

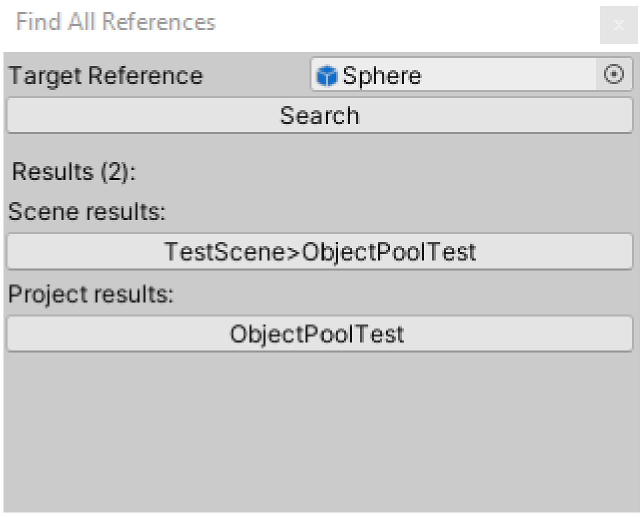
```
public class ParameterLayer : Parameter
{
    public ParameterLayer() : base(new LayerConsoleSuggestionWindow())
    {
    }
...
}
```

Search for references to asset

An incredibly handy feature to keep your project clean and to keep track of references is the new option to search for references of a selected asset. All you have to do is right click the asset in your project and click on **Find All References**.



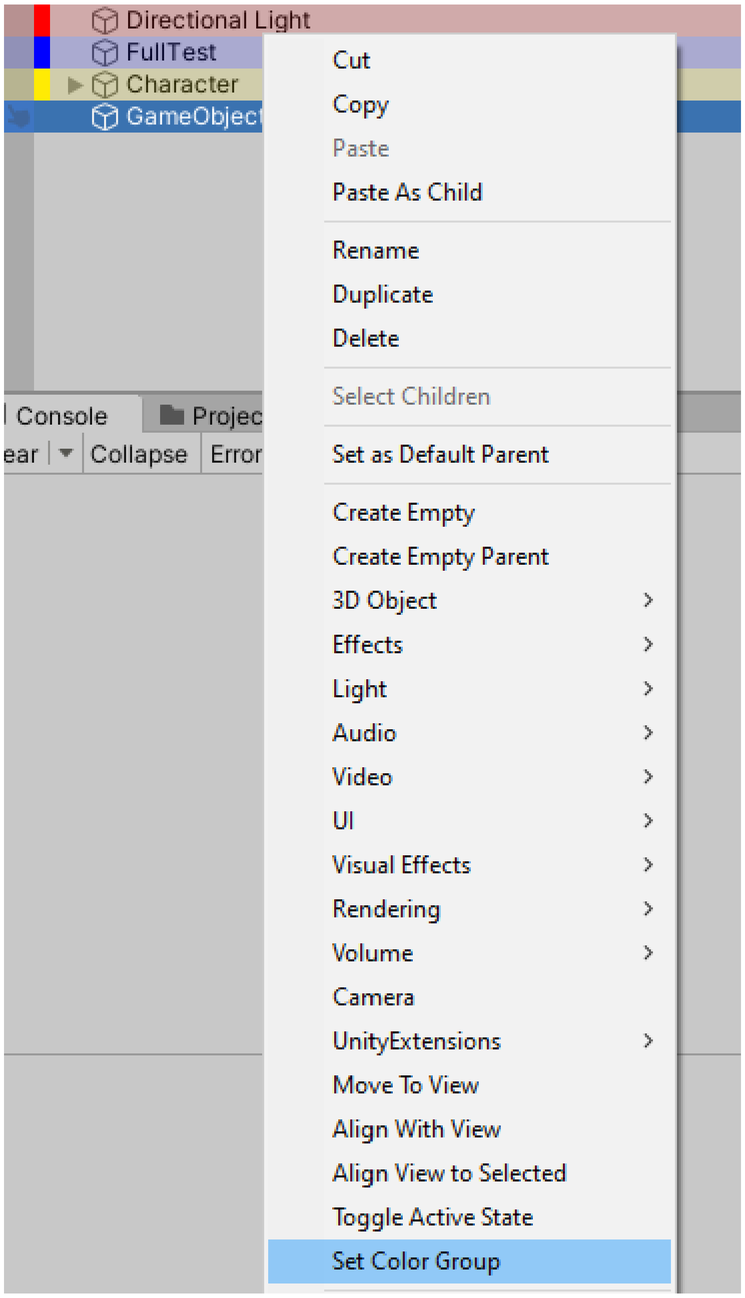
This starts looking for references and will display this window with all found results. Sorted by results in scenes and results in the project itself such as prefabs, scripts, etc.


Find All References
Target Reference
Sphere
Search
Results (2):
Scene results:
TestScene>ObjectPoolTest
Project results:
ObjectPoolTest

Hierarchy and Color groups

The Hierarchy has been improved as well, supporting grouping GameObjects by Color and a handy Enable/Disable button.

To change the color of a GameObject all you have to do is right click the GameObject and select **Set Color Group**



Button and Group Inspector Attributes

Two new attributes have been added to make the inspector easier to use. Namely the **GroupItem** and **Button** attributes.

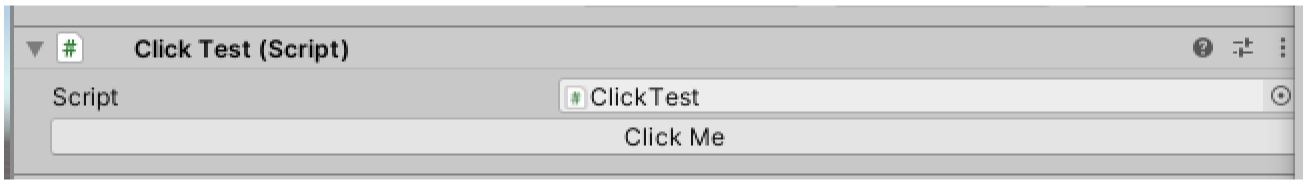
Using these works exactly the same as any other attributes.

Button

To use the button attribute, simple write **[Button]** above any function.

```
using ExtensionsTools;

...

[Button]
void ClickMe() {
    Debug.Log("I've been clicked!");
}
```

And the result in the inspector:



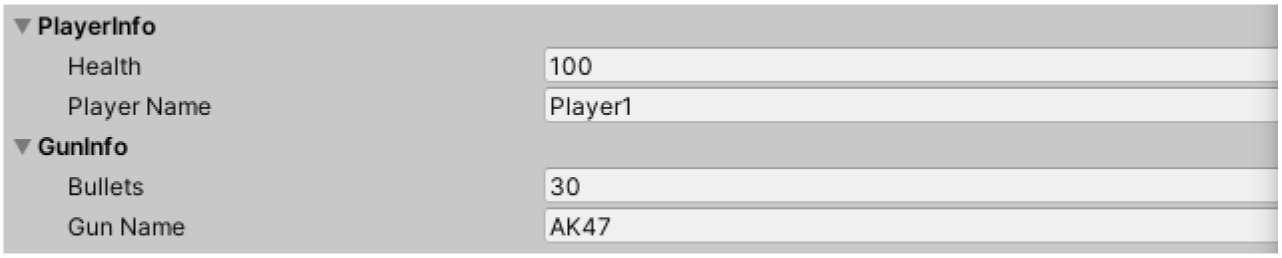
Group

To use the groupitem attribute simple write **[GroupItem(groupName)]** before an **SerializeField**

```
using ExtensionsTools;

...
```

```
[GroupItem("PlayerInfo")][SerializeField] float m_Health;
[GroupItem("PlayerInfo")][SerializeField] string m_PlayerName;

[GroupItem("GunInfo")][SerializeField] int m_Bullets;
[GroupItem("GunInfo")][SerializeField] string m_GunName;
```

And the result in the inspector:



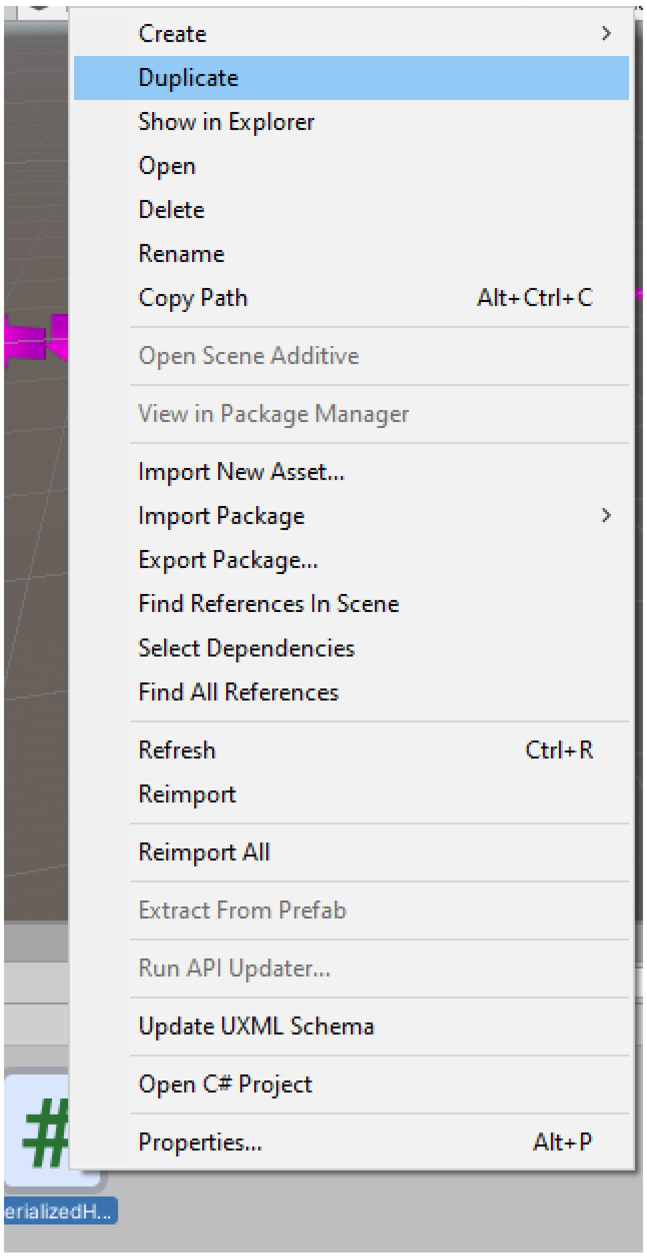
```
▼ PlayerInfo
    Health          100
    Player Name     Player1
▼ GunInfo
    Bullets         30
    Gun Name        AK47
```

On this page

Other

Duplicate

The editor now also has a duplicate button when right clicking an Asset. Allowing you to duplicate your selected asset



GameObject Events

Extension Tools for Unity makes it easier than ever to listen to events. Adding a callback to an event is done using the event listener like this:

```
using ExtensionsTools;

...

gameObject.GetEventListener().OnEnabled += OnEnableCallback;
```

Here is a list of all possible events:

Event
OnNameChange
OnTagChange
OnLayerMaskChange
OnIsStaticChange
OnDisabled
OnEnabled
OnDestroyed
OnAddComponent
OnDestroyComponent
OnReparent
OnTransform

Event
OnMove
OnRotate
OnScale
OnCollisionEntered
OnCollisionStayed
OnCollisionExited
OnTriggerEntered
OnTriggerStayed
OnTriggerExited
OnCollision2DEntered
OnCollision2DStayed
OnCollision2DExited
OnTrigger2DEntered
OnTrigger2DStayed
OnTrigger2DExited

Game Events

Extension Tools for Unity also allows you to listen to Game Events without having to create a new monobehaviour.

```
using ExtensionsTools;
using ExtensionsTools.Events;
...

GameEventListener.INSTANCE.OnApplicationFocused += OnFocusCallback;
```

Here is a list of all possible events:

Event
OnApplicationFocused
OnApplicationQuitted
OnApplicationPaused
OnGUIRender

On this page

Game Data

Saving and reading data between scenes or play sessions is easily achieved using the GameData class.

Saving Data

You only need one method to save all of your data.

```
using ExtensionTools.Data;

...

GameData.SetData("position", Vector2.one);
GameData.SetData("name", "test string");
GameData.SetData("color", Color.green);
GameData.SetData("player", new Player());
...
```

You can use the **SetData** method with the first variable being a variable name and the second being the value to save any primitive type or even custom structs or classes. However when saving a custom class or struct **make sure these are marked serializable**!

Reading Data

Similar to saving data, you only need one method to read data.

```
using ExtensionTools.Data;

...

Vector3 value;
if (GameData.TryGetData("position", out value))
{
    //Value is found
}
else
    //Value not found
...
```

Saving And Loading Data to/from Disk

Saving and loading data to/from the disk is required if you want to store save files for usage between sessions.

Thankfully this can easily be done.

```
using ExtensionTools.Data;

...

//Save the current data to the disk under a default saveFile in the default path
GameData.SaveToDisk();

//Try to load data from the disk under a default saveFile in the default Path
GameData.TryLoadFromDisk();
```

By default the data will be saved and read from the **Application.persistentDataPath** with **default** being the filename. This can however be changed.

```
GameData.SaveToDisk("profile1","C:/Saves/");
GameData.TryLoadFromDisk("profile1","C:/Saves/");
```

Deleting data

Sometimes you might want to remove some data. To do this simply call the **RemoveData** method.

```
GameData.RemoveData("position");
```

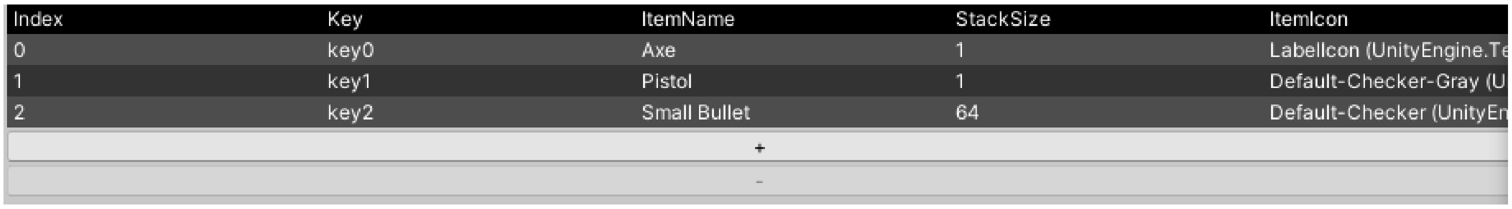
To delete a whole savefile, you can use the **DeleteSaveFileFromDisk** method. With the two optional parameters being once again the name of the savefile and the path.

```
GameData.DeleteSaveFileFromDisk("profile1","C:/Saves/");
```

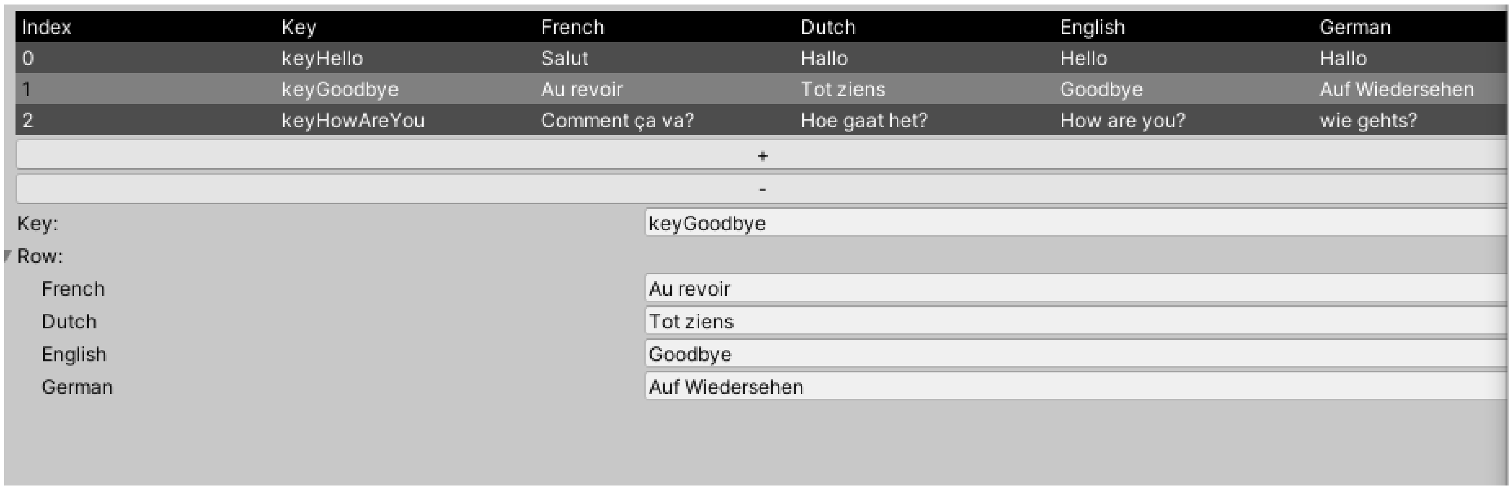
On this page

Data Tables

Data Tables are an easy way to represent data in a structured table. It can be used for all kinds of data types such as localization, items, players,... You can think of it as spreadsheet built into Unity.



Index	Key	ItemName	StackSize	ItemIcon
0	key0	Axe	1	LabelIcon (UnityEngine.Te
1	key1	Pistol	1	Default-Checker-Gray (U
2	key2	Small Bullet	64	Default-Checker (UnityEn



Index	Key	French	Dutch	English	German
0	keyHello	Salut	Hallo	Hello	Hallo
1	keyGoodbye	Au revoir	Tot ziens	Goodbye	Auf Wiedersehen
2	keyHowAreYou	Comment ça va?	Hoe gaat het?	How are you?	wie gehts?

Key: keyGoodbye
Row:
French: Au revoir
Dutch: Tot ziens
English: Goodbye
German: Auf Wiedersehen

Creating a Data Table

Before we create a Data Table, we first have to create the type that will be stored in the Data Table. To do that, we right click in our **project->Create->ExtensionsTools->DataTable->Data Table Type**

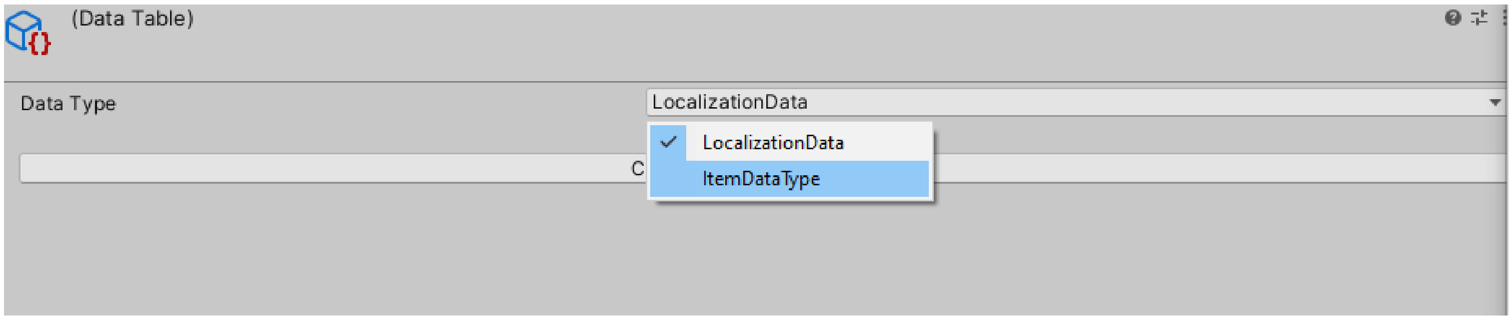
Now a new script will be created, Let's rename it to ItemDataType. When we open the script we will see something like this.

```
using ExtensionTools.Data;
public class DataTableType1: DataTable.DataStruct
{
    public string ExampleName;
    public int ExampleValue;
}
```

This is just an example, we will rename the class and adjust the class like so:

```
using ExtensionTools.Data;
public class ItemDataType: DataTable.DataStruct
{
    public string ItemName;
    public int StackSize;
    public Texture ItemIcon;
}
```

We are now ready to actually create our Data Table. To do that, we right click once again in our **project->Create->ExtensionsTools->DataTable->DataTable**. And we select the type we just created.



We have now succesfully created our DataTable and we can start adding data to it, using the + button and clicking on the row we want to edit.

Accessing rows

There are different ways to access the rows in the DataTable. Let's start with using the keys we can assign in the DataTable.

```
using ExtensionTools.Data;

...

[SerializeField]
DataTable m_ItemDataTable;

void Start(){
```

```
          ItemDataType itemData;
          if (m_ItemDataTable.TryGetRow<ItemDataType>("key0", out itemData))
          {
              //Found row
          }
          else
              //No row found with key keyName
  }
```

In our example using keys doesn't really make sense. We rather access these values using the index. This can be done like this.

```
ItemDataType itemData= m_ItemDataTable.GetRowFromIndex<ItemDataType>(0);
```

On this page

Mono Singletons

Mono Singletons are monobehaviours of which only one can be present at the same time. They can easily be referenced using the INSTANCE property. They are ideal for manager classes such as a Game Manager, Audio Manager, etc... When there is no instance present in the scene, one will be created when you try to use the singleton. In most cases you don't want to add the script to a GameObject in your scene.

Creating a Singleton

To create a singleton, all you have to do is inherit from Monosingleton.

```
using ExtensionTools.Singleton;
public class MyCustomSingleton : MonoSingleton<MyCustomSingleton>
{
    public void LogTest()
    {
        Debug.Log("I am a singleton");
    }
}
```

Now we have a singleton and we can easily call the LogTest method from anywhere without having to keep a reference.

```
MyCustomSingleton.INSTANCE.LogTest();
```

Making the Singleton persistent across scenes

Sometimes singletons shouldn't be created foreach scene. We might want to have one persistent reference to the same singleton across all scenes. For example an Audio Manager, it doesn't make sense to have one for every scene since audio will be handled the same in every scene.

To keep singletons persistent across scenes we just have to set the DontDestroyOnLoad boolean to true in the constructor.

```
using ExtensionTools.Singleton;
public class MyCustomSingleton : MonoSingleton<MyCustomSingleton>
{
    public MyCustomSingleton() : base(true)
    {
    }

    public void LogTest()
    {
        Debug.Log("I am a singleton");
    }
}
```

On this page

Object Pooling

If you've been working in Game Development for a while you might be familiar with the concept of object pooling. **Object pooling lets you reuse objects** so you don't have to allocate new memory everytime you want to spawn an Object.

In the ExtensionTools there's a **simple solution for pooling GameObjects**. This is ideal for situations where you have to spawn a lot of GameObjects from the same Prefab, such as shooting a gun and spawning bullets.

Creating the Object Pool

Creating the Object Pool is very easy to do. Let us take the example of shooting a gun and spawning bullets.

```
using ExtensionTools.ObjectPooling;
public class Gun : MonoBehaviour
{
    [SerializeField]
    GameObject m_BulletPrefab;

    [SerializeField]
    int m_MaximumBullets= 10;

    UnityObjectPool m_ObjectPool;
    void Start() {
        m_ObjectPool = new UnityObjectPool(m_BulletPrefab, m_MaximumBullets, false);
    }
}
```

When we take a look at the declaration of the UnityObjectPool we can see what's happening.

```
public UnityObjectPool(GameObject prefab,int initialpoolSize=10,bool autoExpand=false,
LogLevel logLevel=LogLevel.Warning)
```

First we pass the BulletPrefab this is the Prefab which will be instantiated and reused. Now we set the initial pool size this is the amount of bullets that we can have concurrently.

And finally we set Auto expand to false, Setting this to true would allow the pool to grow above the initial pool size when needed. Since we disabled this the oldest GameObject will be used instead when we reached the Maximum Pool Size.

Now we can spawn the bullet.

Important! Make sure to reset all the necessary values since we are reusing GameObjects and not spawning new ones

```
void SpawnBullet() {
   GameObject go = m_ObjectPool.SpawnObject(transform.position, Quaternion.identity,
this.transform);
   go.GetComponent<Rigidbody>().velocity = (transform.forward * 10);
}
```

Releasing Pooled Objects

In most cases we also want to **Release** our Objects. This will mark them as inactive and will tell our ObjectPool that they are ready to be reused. This is especially important when enabling **autoExpand** because the Object Pool will keep creating new Objects when no Objects are marked to be reused.

Releasing an Object is done by getting the **UnityPooledObject component** on the GameObject and calling **Release()**;

Let's say we want our bullets to be released when they hit an obstacle. We can easily implement this using the EventListener

```
void SpawnBullet() {
   GameObject go = m_ObjectPool.SpawnObject(transform.position, Quaternion.identity,
this.transform);
   go.GetComponent<Rigidbody>().velocity = (transform.forward * 10);

   go.GetEventListener().OnCollisionEntered = null; //We clear all bindings first
since we reuse Objects
```

```
    go.GetEventListener().OnCollisionEntered += (Collision col) => {
go.GetComponent<UnityPooledObject>().Release(); };
}
```

On this page

GizmosExtended

There has been expanded on the default Unity Gizmos, making it easier to draw complex gizmos such as Arrows, Textures, Icons, Text, Circles... The usage is the same as the regular Unity Gizmos except with the GizmosExtended class instead of the Gizmos class.

```
using ExtensionTools.Gizmos;
public class GizmosTest : MonoBehaviour
{
    private void OnDrawGizmos()
    {
            GizmosExtended.DrawText(transform.position, "This is a Gizmo",
Color.white);
            GizmosExtended.DrawArrow(transform.position, transform.position +
Vector3.up * 5);
    }
}
```

This is a list of all new Gizmos

Method
DrawCircle
DrawArrow
DrawPath
DrawTexture
DrawIcon
DrawText

Gizmos in URP

When using the Universal Renderpipeline you might come across a warning tellign you that you didn't add the Gizmo Feature to the Render Pipeline.

In this case you want to navigate to your Default Renderer Asset in your project and add the Gizmos Feature.

Character Movement

ExtensionTools For Unity has a built in First Person Character Movement. This is meant for prototyping or for some FPS games. This character respects the current InputMode state.

The easiest way to quickly create a First Person Character is by right clicking in the **Hierarchy->ExtensionTools->Character**.

Adjusting the input buttons and axis

By default the character movement will move fine with Mouse and Keyboard as long as you kept the default Unity Axes and Buttons in the Project Settings. However sometimes you might want to change these and add support for Joysticks etc...

To do this, just click **Window->ExtensionTools->Gameplay Settings**

On this page

Flying Movement

There also is an built-in Flying Movement Component. This is mainly used to scout the scene during development phase and behaves similar to the Scene Camera. This component is also used for the Debug Camera.

Creating a Flying Camera is done by right clicking in the Hierarchy->ExtensionTools->Flying Camera.

Adjusting the input buttons and axis

This is done exactly the same as with the Character Movement, since they both use the same Axes and Buttons. And can be read more about here

On this page

Input Mode

The Input Mode class is a handy way to switch between several modes of input. For example, when the game is paused or we are in an UI inventory we want the **cursor to be visible and the cursor's lockstate to be confined**. In this case we would set the **InputMode to UI**. When unpausing or when closing our inventory we once again put our **InputMode to Game** and the cursor will be **hidden and locked**.

This also allows us to check our InputMode before handling input. We don't want our character to be able to walk around when we are managing our inventory.

```
using UnityEngine;
using ExtensionTools;
public class Inventory : MonoBehaviour
{
    void OpenInventory()
    {
        //Enable cursor and unlock
        InputMode.SetInputModeUI();
    }

    void CloseInventory() {
        //Hide cursor and lock
        InputMode.SetInputModeGame();
    }
}
```

In some cases you might want your cursor to be visible in the Game Mode as well, for example when creating a card game. To do this you have two options.

Option one is to pass a GameModeSettings class everytime you change the inputmode back to Game Mode.

```
InputMode.SetInputModeGame(new
InputMode.GameModeSettings(CursorLockMode.Confined,true));
```

Option two is to override the default GameModeSettings somewhere in the start of the game. Now you don't have to pass GameModeSettings everytime you change the Inputmode back to Game.

```
InputMode.SetDefaultGameModeSettings(new
InputMode.GameModeSettings(CursorLockMode.Confined, true));
```

Checking for InputMode

When writing our character movement or input in game we don't want it to be called when we are in the UI Game Mode. It doesn't make sense to be able to move our character when we are in a pause menu or in our inventory. So we just write a simple if statement before our movement code.

```
if (InputMode.currentInputmode == InputMode.Input.Game)
{
      //movement code
}
```

The Debug Mode

The Debug Mode is a handy mode during development of your game but should only be used for development purposes. By default switching to this Inputmode will enable the Debug Camera and allow you to move around in your scene without disturbing the flow of the game. Think of it as a sort of God Mode for development purposes.

The recommended way to switch to the Debug Mode is by using the Console

On this page

Collections

Collections such as lists and arrays have gotten more extensions to improve your workflow as well.

```
using ExtensionsTools;

...

string[] array = new string[] { "this", "is", "a", "test" };
int index = 3;
if (array.IsIndexValid(index))
    return array[index];
return ""
```

Regular Collections

Here is a list of all extensions for Lists or arrays

Method	Description
Shuffle	Shuffles this collection
IsArrayIdentical	Check if an array is identical to another array
Union	Creates an union of this collection and another collection
Intersect	Creates an intersection of this collection and another collection
Difference	Creates a difference of this collection and another collection
AddUnique	Add a value to the collection if it doesn't exist in this collection yet
GetRandom	Gets a random value from this collection

Method	Description
RemoveElements	Remove all elements equal to this value
IsIndexValid	Checks if this index is valid in the collection and returns true if so
Resize	Resize this collection

GameObject/Vector/Transform Collections

Method	Description
SortByDistanceFromTarget	Sorts a collection by distance fr a target
GetClosestToTarget	Returns the closest GameObject/Vector/Transform target in this collection
GetFurthestFromTarget	Returns the furthest GameObject/Vector/Transform from a target in this collection
GetGameObjectsInRange/GetTransformsInRange/GetPositionsInRange	Returns the GameObjects/Transforms/Posit in range

GameObject Collections

Game Object Method	Description
GetAverage	Get the average position of this collection

Serializable Dictionary and Hashsets

A struggle with the regular .NET dictionaries and hashsets is that these cannot be serialized and thus can't be displayed in the editor or saved to files.

The SerializableDictionary and SerializableHashset changes this.

```
using ExtensionTools.Collections;

...
[SerializeField]
SerializableHashSet<string> m_HashSet;
[SerializeField]
SerializableDictionary<string, Vector2> m_Dictionary;
```

And the editor result:

Ordered Dictionary and Hashsets

Sometimes we need Dictionaries or Hashsets which also guarantee their order.

For this purpose you can use:

- OrderedDictionary
- OrderedSet

On this page

GameObjects

GameObjects have gotten a lot of handy extension methods which can be used to improve your workflow. When using ExtensionsTools for Unity you will notice that GameObjects have gotten more methods to use. Usage is very simple:

```
using ExtensionTools;

...

var rigidbody=gameObject.GetOrAddComponent<Rigidbody>();
```

Tags

Often during development you want a GameObject to be able to have multiple tags, this is now possible. Using ExtensionTools for Unity GameObjects support having multiple tags. This is done using the tag component or the tag extensions methods.

All extensions

Method	Description
DestroyImmediate	The same as GameObject.DestroyImmediate(obj)
Destroy	The same as GameObject.Destroy(obj)
SafeDestroy	Destroy this GameObject/Component/Asset immediately or not based on Application.IsPlaying
GetOrAddComponent	Gets the component if it exists, if not it gets created and added

Method	Description
GetEventListener	Gets the EventListener (see Event Listener
StartCoroutine	Starts a coroutine on this GameObject (similar to Monobehaviour.StartCoroutine but you don't need a separate monobehaviour)
StopCoroutine	Stops a coroutine started by StartCoroutine on this GameObject
StopAllCoroutines	Stops all coroutines started by StartCoroutine on this GameObject
SetIcon	Sets an Icon for the GameObject in the editor
RemoveTag	Removes a tag
AddTag	Adds a tag
HasTag	Check if this GameObject contains this tag

GameObjectExtended

There is also an additional class for static functions namely **GameObjectExtended**. Currently it only supports one function.

```
using ExtensionTools;

...

var objects=GameObjectExtended.GetAllGameObjectsInScene();
```

Method	Description
GetAllGameObjectsInScene	Retreives all GameObjects in the scene

On this page

Colors

Similar to GameObjects Colors also have gotten some new extension methods.

```
using ExtensionsTools;

...

Color Salmonred=Colors.Salmon;
Salmonred.ChangeSaturation(0.2f)
```

All extensions

Method	Description
Copy	Copies the values of this Color to another Color
ChangeSaturation	Saturates or Desaturates this color
ChangeBrightness	Brightens or Darkens this color
ShiftHue	Shifts the hue of this color
GetLuminance	Returns the luminance of this color

New Colors

The static Colors class has a lot of additional colors to use.

Color

Color
LavenderBlush
Snow
DarkGray
LightGray
DarkRed
Salmon
Orange
OrangeRed
Gold
Maroon
Chocolate
Wheat
Olive
DarkGreen
ForestGreen
Teal
DeepSkyBlue
LightBlue

Color
Purple
Fushia
Pink
HotPink

Transforms

Similar to GameObjects Transforms also have gotten some new extension methods.

```
using ExtensionTools;

...

transform.FindDeepChild("ChildName");
```

Method	Description
DestroyAllChildren	Destroys all the children of this Transform
UnparentAllChildren	Unparents all the children of this Transform
ReparentAllChildren	Gives all the children a new parent
SetChildrenLayermask	Sets the layermask of all (deep)children
FindDeepChild	Similar to transform.Find but also includes deepchildren
MoveTowards	Simply move this transform to a target within a certain time frame
MoveParabolicTowards	Simply move this transform parabolic to a target within a certain time frame (Useful for grenade throws and jumps ...)
PlayAnimation	Play a custom animation on this transform
CancelAllAnimations	Cancel all animations on this transform
RotateTowards	Simply rotate this transform towards a target
ScaleTowards	Simply scale this transform towards a target

Structs

ExtensionTools for Unity also supports some new structs.

- Line2D
- Line3D
- Transformation