Petteri Paju

IMGUI EXTENSIONS IN UNITY3D

Bachelor's Thesis
Degree Programme in Information Technology / Game Programming

2017

Kaakkois-Suomen ammattikorkeakoulu

Tekijä/Tekijät Petteri Paju	**Tutkinto** Insinööri (AMK)	**Aika** Joulukuu 2017
Opinnäytetyön nimi IMGUI Extensions In Unity3D		69 sivua 0 liitesivua
Toimeksiantaja Kaakkois-Suomen Ammattikorkeakoulu / GameLab		
Ohjaaja Lehtori Niina Mässeli		

Tiivistelmä

Tämän opinnäytetyön tarkoituksena on tutkia ja havainnollistaa kuinka Unity 3d pelimoottorin käyttöliittymää voidaan laajentaa. Editori laajennukset voivat lisätä Unityn editoriin uusia työkaluja ja toimintoja. Editori laajennukset voivat hyödyntää pelinkehitystö, parantamalla työnkulkua, vähentämällä virheitä ja automatisoimalla yleisiä tehtäviä. Editori laajennukset rakennetaan Unityn sisään joko visuaalisina työkaluina tai automatisoituina komentoina, jotka voidaan aktivoida napeilla.

Opinnäytetyö sisältää sekä teoriaa, että käytäntöä, siitä kuinka Unityn editori toimii. Käyttöliittymä esittelee teorian Unityn käyttöliittymästä, sekä yleiset elementit joita käytetään editori laajennusten toteuttamiseen. Koska tästä aiheesta ei ole useita aikaisempia opinnäytetöitä, tavoitteena on tarjota selkeitä esimerkkejä, jotka demonstroivat editori laajennusten toiminnollisuutta. Opinnäytetyön lopussa esitellään käytännönosuus, jonka aiheena oli luoda esimerkki peli, jonka kehityksen tueksi luotiin erilaisia editori laajennuksia. Päätavoitteena oli esitellä eri tapoja, kuinka editori laajennuksia voidaan käyttää.

Vaikka aiheeseen liittyy joitakin toimintoja ja aihealueita, jotka jäivät työn ulkopuolelle, työ onnistuneesti käsitteli tärkeimmät aihealueet, jotka liittyvät editori laajennuksiin. Yleisellä tasolla työ kävi läpi perusasiat, muutamia yleisiä käytäntöjä sekä edistyneempiä menetelmiä. Käytännönosuus demonstroi mainiosti mahdollisuuksia, joita editorilaajennukset tarjoavat.

Asiasanat

ohjelmointi, unity, ui, imgui, käyttöliittymä, c#

South-Eastern Finland University of Applied Sciences

Author (authors)	**Degree**	**Time**
Petteri Paju	Bachelor of Engineering	December 2017
Thesis Title IMGUI Extensions In Unity3d		69 pages 0 pages of appendices
Commissioned by Kaakkois-Suomen Ammattikorkeakoulu / GameLab		
Supervisor Niina Mässeli, Senior		

Abstract

The goal of this thesis was to study and demonstrate how the Unity3d game engines user interface can be extended. Editor extensions can add new tools and functionalities to Unity editor. Editor extensions can be very beneficial to game development, by improving the workflow, adding safety, and automating common tasks. Editor extensions are built inside Unity as either visual tools or automated tasks triggered by menu clicks.

The thesis contains both theory and practice about Unity editor works. The thesis discusses the theory of Unity editor system as well as the most commons elements that are used to build custom tools. As there are not many previous studies about the topic, the thesis aims to provide illustrative examples to demonstrate how editor extensions work. The thesis concludes with a case study, where a sample game was made along with different kind of editor extensions. The main goal of the case is to showcase how editor extensions can be used in different ways.

While there were some features and topics that were cut out from the thesis, it successfully covered most crucial information necessary for developing editor extensions. The overall thesis was able to cover the basics, some good to know practices as well as some advanced topics. The case study serves as a serviceable demonstration of possibilities provided by editor extensions.

Keywords

programming, unity, ui, imgui, user-interface, c#

CONTENTS

TERMS AND ABBREVIATIONS

Control	An element in UI. Control can be an element like button, label, or textbox.
GUI / UI	User-interface, made of both visual and interactive components.
Inspector	A window that allows viewing and modifying properties of UnityEngine.Objects.
Serialization	A progress where data is translated from one format to another for transfer and storage purposes.
ScriptableObject	A special script class in Unity. Instances of this class can be saved to a file. Inherits from UnityEngine.Object
UnityEngine.Object	A base class for gameObjects and components in Unity

1 INTRODUCTION

The main purpose of this thesis is to research and illustrate how appearance and functionality of Unity game-engine can be extended. Editor extensions are an asset in game development, regardless of the scale of the project. Editor extension can improve readability of a UI and optimize a workflow, potentially giving a major boost in the development speed. Instead of allowing the project to be limited by default Unity tools, by extending editor one can create whole new ways to develop a game.

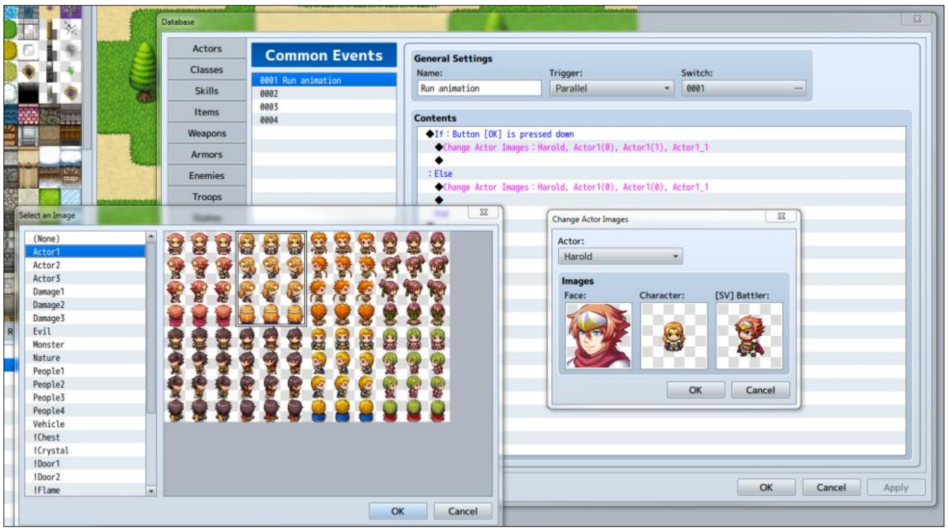
The thesis begins by explaining in detail what editor extensions are and why they should be used while illustrating some potential pratfalls that they bring. After the concept of editor extensions has been explained, the thesis moves to examine Unity Editor and discusses some basic concepts critical to understanding how the system works. Next topics move from theory to practice and start to examine different elements used to create editor extensions, such as containers and UI-elements. Before talking about the case, the goal is to cover the most common editor elements, their uses and how they are implemented.

The case study demonstrates how different tools are used in practice. The topic of the case study was to create editor tools for a 2D-turn-based strategy game. The case aims to illustrate how different kind of editor tools are implemented and how they change the way a game can be developed.

2 EDITOR EXTENSIONS

Unity is a very extensive game engine and can be used to develop any kind of game, from text adventures to open-world 3D-games. Thanks to this extensiveness Unity has become one of the most used game engines in the industry. This fact also exposes one of its flaws, namely that Unity lacks many game genre-specific tools. Unity's tools are generic by design so that they can be used in different kind of projects. Different types of games, however, require all kinds of different tools and features, that Unity simply does not provide. This, of course, makes sense, since filling the engine with too many different tools would make any game engine very convoluted and alienate beginners.

In contrast to a multi-purpose game engine, like Unity, there exist many game engines, that are aimed towards a specific type of game genre. Because these engines are made with a specific type of game in mind, they can include more tools that are necessary for the specific game type. For example, RPG Maker game engine can be used to create old-school 2D-roleplaying games. RPG Maker includes many visual tools like character creation, event-generation, and map-drawing tools, some of these tools can be seen in Figure 1. In the same way that Unity has tools, that can be used in any kind of game, RPG Maker has tools that can be used in any kind of RPG-game. The distinction is that with RPG Maker it is much easier to create RPG games because a developer has all necessary tools from the get-go. This becomes even clearer with engines that are developed specifically for a single game in mind. Developing a similar game with default Unity tools would take much longer and be less intuitive.



General Settings
Name: Run animation
Trigger: Parallel
Switch: 0001

Contents

```
◆If : Button [OK] is pressed down
  ◆Change Actor Images : Harold, Actor1(0), Actor1(1), Actor1_1
  ◆
 : Else
  ◆Change Actor Images : Harold, Actor1(0), Actor1(0), Actor1_1
  ◆
```

Change Actor Images
Actor: Harold
Images
Face:
Character:
[SV] Battler:

Figure 1. RPG Maker (Enterbrain Inc. 2017)

This brings us to Unity editor extensions. Editor extensions are custom tools that provide additional functionality to Unity Editor. These tools alter how default Editor tools work or implement completely new functionalities. Just like tools in RPG Maker, these custom tools can make the game development process more intuitive and reduce the development time. (Tadres 2015, 17.)

There are different kinds of editor tools. Some tools are simple minor UI-changes or macros that automate some basic action. Other tools serve as the backbone for the entire game development, whole new visual tools that completely change the way the game is developed. Some tools are made specifically for certain projects or game studio, others are sold as commercial products in Unity Asset Store. (Tadres 2015, 17.)

2.1 Advantages

As mentioned above, editor extensions are very advantageous to the project. A basis example is action automation or macros. Macros are used to automate sequences of manual inputs into a single button click. Macros can make commonly repeated actions faster and reduce the risk of mistakes that could happen if action was done repeatedly by hand. (Tadres 2015, 24.)

Some extensions can alter the appearance of default Editor UI. Unity allows developers to add custom tooltips and alter how some objects appear in the editor. Hand tailoring UI makes it easier for new members of the team to understand the project. Tooltips are especially useful for newcomers. Unity Editor can be extended to inform a user when game object is missing a reference or a necessary component, or even automatically fix the issue. By making tools safer and easier to use, even non-programmer members of the development team can develop the game in Unity. (Smith & Queiroz 2015, 507-508.)

Some more complex tools are visual interfaces for actions that would normally need coding to be accomplished. A great example is many visual scripting tools sold in Unity Asset Store. These tools are like Blueprint in Unreal Engine, seen in Figure 2. Visual scripting tools allow users to create behavior trees, that would normally need user-written scripts. In short, editor extension can be used to reduce the amount of code that needs to be written for the game. (Miles 2016, 60.)

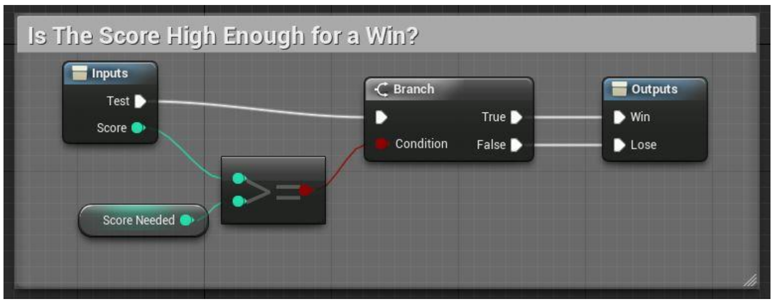


Figure 2. Unreal Engine Blueprint (Epic Games 2017)

The examples above express some core advantages for editor extensions: speed, safety, and usability. In essence, editor extensions make game development more intuitive, reducing the overall development time by optimizing the workflow. (Smith & Queiroz 2015, 530.)

2.2 Disadvantages

In some ways, editor extensions can also work against the project. Developing of complex tools takes time, that could be better used developing the actual game. When creating tools, it is important to evaluate whatever the time invested in creating a tool is worth the time the tool would save. Whatever tools should be made depends on many factors such as the scale of the project, complexity of the actual tool and how often it would be used. If the project is small, investing a lot of time in editor tools might, in the end, prolong the development time. Longer and more complex the actual project is, more beneficial the tools will be to the project. (Blow 2004.)

Another problem with editor extensions is that they may introduce new functionality to Unity that developers must learn to use. While some more popular editor extensions sold in Unity Asset store are well documented and have a lot of tutorials online, this might not be the case for extensions that are built in-house. When introducing new tools to the project, it is important that they are well documented so that even new members of the team can learn to use them.

Extensions can also introduce some new vulnerabilities to the project. Because Unity is constantly releasing new versions, there is a danger that custom made editor tools become incompatible with the newest version, making it harder to migrate game to the newer version. Editor tools are also subject to changes in the project itself. Major changes in the base game require equal changes to editor tools, this, in turn, increases the development time.

Like any program, editor extensions are vulnerable to bugs, which can cause corruption and data loss. This is especially true for tools that modify data trees and files. Poorly made tools can fail to save changes or even accidentally delete data. This risk can be reduced with backups, failchecks and frequent testing.

2.3 Example extensions

There are many 3rd party made editor extensions available in Unity Asset Store, some free and others for sale. Extensions range from simple debug tools to extensive game toolkits. In past, many tools have become so popular that they have been officially integrated into Unity. Because these tools are by themselves a commercial product, they are highly finalized. It is important to note that for tools built for in-house usage visual appearance is not the first priority. As long as tools works and are at least fairly usable, investing additional time for the outer appearance of tools might be a waste of resources.

2.4 PlayMaker

One of the most popular editor extensions is a visual scripting tool Playmaker made by Hutong Games (Figure 3). PlayMaker is a node-based programming tool, which allows the user to create script-like behavior with visual tools. Conceptually Playmaker is similar to Blueprints in Unreal Engine. Playmaker visualizes function calls and logic with nodes and wires connecting them. This allows even less programming oriented developers to create logic for games. Due to visual nature of the tool, it is easy to understand how the “code” works. Playmaker made behavior has a smaller risk of syntax and logic errors than regular code, due to its visual nature. (Miles 2016, 60-61.)

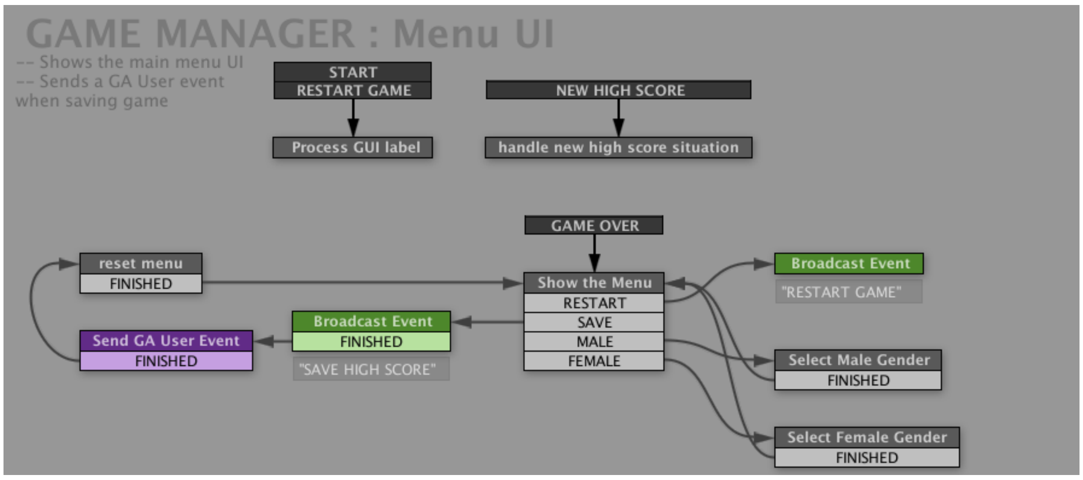


Figure 3. Visual scripting in Playmaker (Hutong Games 2017)

Other Playmaker features include Network support, editor localization and visual debugging. PlayMaker is widely used to a point where Unity has released official video tutorials for the PlayMaker. Other developers have also released their own add-ons to the Playmaker further expanding its range of usage. PlayMaker is still a 3rd party tool and is sold in Asset Store for 65$. (Hutong Games 2017.)

2.5 Anima2D

Anima2D (Figure 4) is a 2D-animation tool developed by Mandarina Games. Anima2D completely changes the workflow of how 2D objects are animated in Unity. Anima2D's main feature is bones, a skeleton that warps the 2D-sprite it is attached to. Bones are a common tool in both 2D and 3D animation, they allow intuitive animation of a character because one can move specific part of the character's body without moving other. (Mandarina Games 2015.)

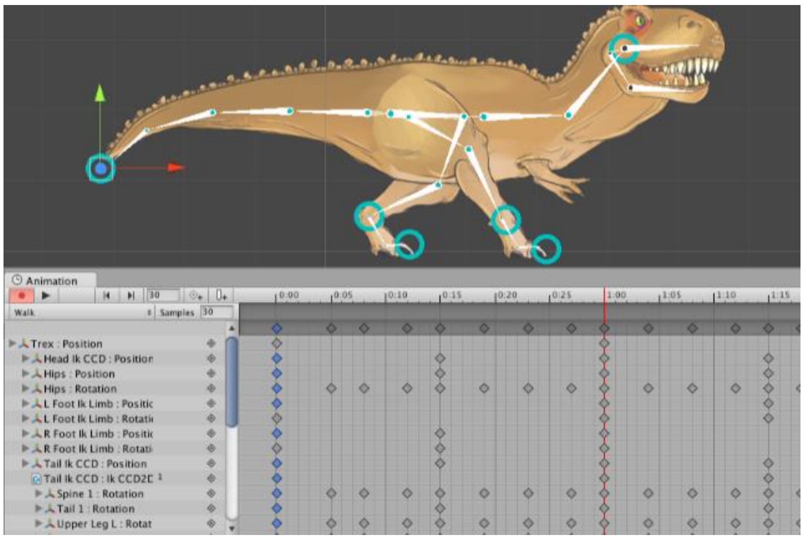


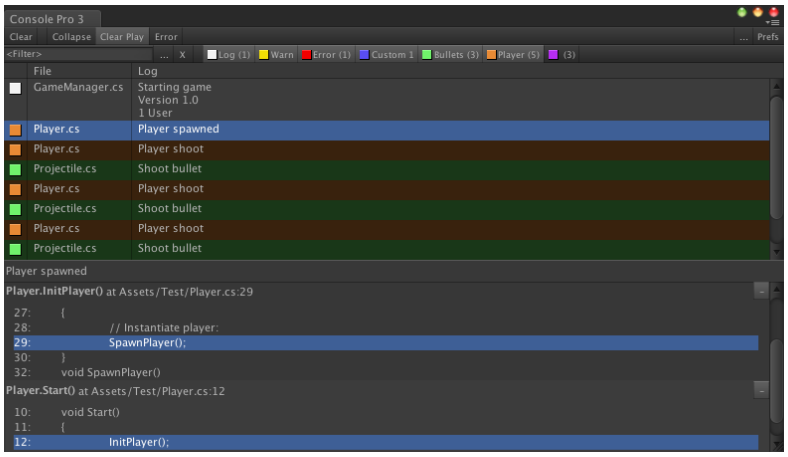
Figure 4. Anima 2D (Mandarina Games 2015).

As editor extension Anima2D is a bit different from Playmaker, which essentially adds convenience to programming progress, but generates results that could be achieved by regular programming. Anima2D adds functionality not possible with normal Unity animation tools. Anima2D is integrated to Unity interface and work together with normal animation window (seen in Figure 4).

Anima2D became free in 2017, thanks to a deal between Unity Technologies and Mandarina Games. Before that, the add-on was on sale for 60$. According to Anima2D's co-founder Sergi Valls, who has now joined the Unity Technologies, the end goal of the partnership is to completely integrate Anime2D the tools to Unity as opposed to keep them as a separate plugin. (CG Channel Inc 2016.)

2.6 Editor Console Pro

Editor console Pro (Figure 5) is meant to replace Unity's own Debug-console (Figure 6). Editor Console Pro serves the same basic function as the normal Console. The purpose of a console window is to display errors and messages generated by Unity. While the normal Console handles this job well, it is very basic and often displays a lot of useless information, making the window look crowded. Editor Console Pro is more organized and easier to read. Editor Console Pro even display the actual code, which causes the error instead of just giving the script row number of the error or warning.

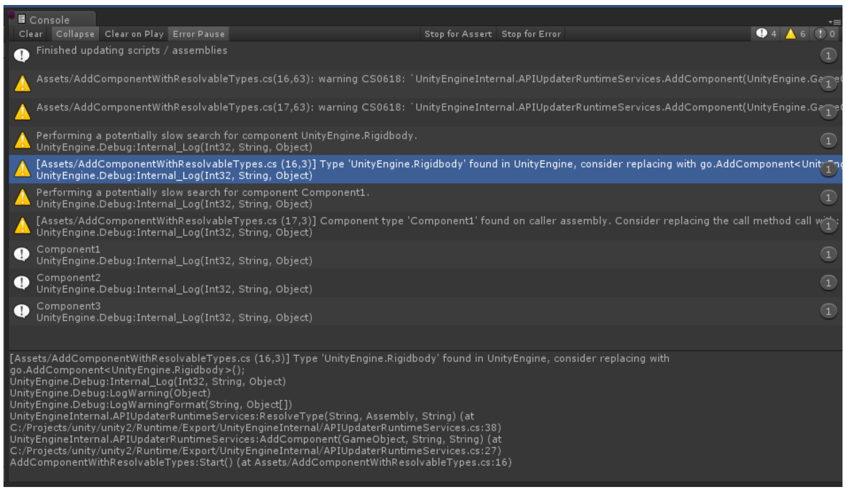


	File	Log
⬜	GameManager.cs	Starting game Version 1.0 1 User
🟧	Player.cs	Player spawned
🟧	Player.cs	Player shoot
🟩	Projectile.cs	Shoot bullet
🟧	Player.cs	Player shoot
🟩	Projectile.cs	Shoot bullet
🟧	Player.cs	Player shoot
🟩	Projectile.cs	Shoot bullet

```
Player spawned
Player.InitPlayer() at Assets/Test/Player.cs:29
27:     {
28:             // Instantiate player:
29:             SpawnPlayer();
30:     }
32:     void SpawnPlayer()
Player.Start() at Assets/Test/Player.cs:12
10:     void Start()
11:     {
12:             InitPlayer();
```

Figure 5. Editor-Console Pro (FlyingWorm 2017)

The biggest selling point for the Editor Console Pro is its additional features, that make debugging process easier. The developer can search Console window for specific entries, track changes made to variables, debug standalone versions of the game and permanently hide unnecessary messages. (The Knights of Unity 2016). All these functions are meant to make Console window more usable and informative. Unlike other tools mentioned in this chapter, Console Window pro is a utility tool, not directly used to develop the game, but nevertheless a great example of how editor extensions can be used in variety of different ways.



```
Finished updating scripts / assemblies                                                                1
Assets/AddComponentWithResolvableTypes.cs(16,63): warning CS0618: `UnityEngineInternal.APIUpdaterRuntimeServices.AddComponent(UnityEngine.Ga   1
Assets/AddComponentWithResolvableTypes.cs(17,63): warning CS0618: `UnityEngineInternal.APIUpdaterRuntimeServices.AddComponent(UnityEngine.Ga   1
Performing a potentially slow search for component UnityEngine.Rigidbody.                             1
UnityEngine.Debug:Internal_Log(Int32, String, Object)
[Assets/AddComponentWithResolvableTypes.cs (16,3)] Type 'UnityEngine.Rigidbody' found in UnityEngine, consider replacing with go.AddComponent<Unit   1
UnityEngine.Debug:Internal_Log(Int32, String, Object)
Performing a potentially slow search for component Component1.                                        1
UnityEngine.Debug:Internal_Log(Int32, String, Object)
[Assets/AddComponentWithResolvableTypes.cs (17,3)] Component type 'Component1' found on caller assembly. Consider replacing the call method call w   1
UnityEngine.Debug:Internal_Log(Int32, String, Object)
Component1                                                                                            1
UnityEngine.Debug:Internal_Log(Int32, String, Object)
Component2                                                                                            1
UnityEngine.Debug:Internal_Log(Int32, String, Object)
Component3                                                                                            1
UnityEngine.Debug:Internal_Log(Int32, String, Object)

[Assets/AddComponentWithResolvableTypes.cs (16,3)] Type 'UnityEngine.Rigidbody' found in UnityEngine, consider replacing with
go.AddComponent<UnityEngine.Rigidbody>();
UnityEngine.Debug:Internal_Log(Int32, String, Object)
UnityEngine.Debug:LogWarning(Object)
UnityEngine.Debug:LogWarningFormat(String, Object[])
UnityEngineInternal.APIUpdaterRuntimeServices:ResolveType(String, Assembly, String) (at
C:/Projects/unity/unity2/Runtime/Export/UnityEngineInternal/APIUpdaterRuntimeServices.cs:38)
UnityEngineInternal.APIUpdaterRuntimeServices:AddComponent(GameObject, String, String) (at
C:/Projects/unity/unity2/Runtime/Export/UnityEngineInternal/APIUpdaterRuntimeServices.cs:27)
AddComponentWithResolvableTypes:Start() (at Assets/AddComponentWithResolvableTypes.cs:16)
```

Figure 6. Unity Debug Console (Unity Technologies 2015)

3 IMGUI-SYSTEM

IMGUI (Immediate Mode GUI) is the name of the system used by Unity to create UI for its editor. IMGUI used to be the primary system for in-game UI until it was replaced by the new Unity UI-system in 2014. The Unity Developer Blog describes IMGUI and Unity UI as Immediate mode GUI and Retained mode GUI respectively. (Unity Technologies 2015.)

Understanding the difference between two systems is key to understanding how IMGUI works. As Retained GUI-system, the current Unity UI retains information about the elements that are being drawn on the screen. This means that elements drawn to UI are created only once in code, and any changes to the said element are done by changing values of that element. In contrast, IMGUI-elements are recreated every time the UI is re-drawn using values provided by the code. So, to change the content of the button, one must change the values that code gives to the button before it is recreated. IMGUI is a system that constantly redraws itself, unlike Unity UI, that only needs to be redrawn when an element is changed. (Unity Technologies 2015.)

3.1 Events

Drawing IMGUI-UI happens inside a single draw-loop, usually called OnGUI or something similar. To handle events like button presses IMGUI takes advantage of Unity event-system which allows UI to behave differently depending on the current event. The basic flow of drawing IMGUI is shown in Figure 7. (Unity Technologies 2015.)

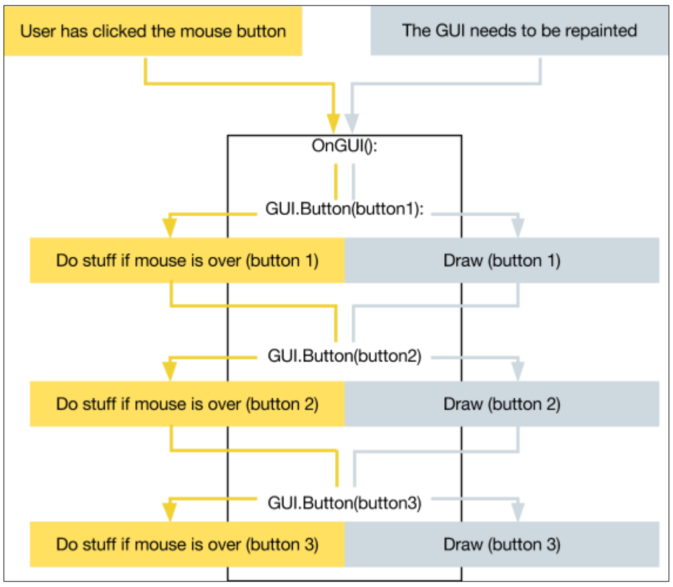

User has clicked the mouse button
The GUI needs to be repainted
OnGUI():
GUI.Button(button1):
Do stuff if mouse is over (button 1)
Draw (button 1)
GUI.Button(button2)
Do stuff if mouse is over (button 2)
Draw (button 2)
GUI.Button(button3)
Do stuff if mouse is over (button 3)
Draw (button 3)

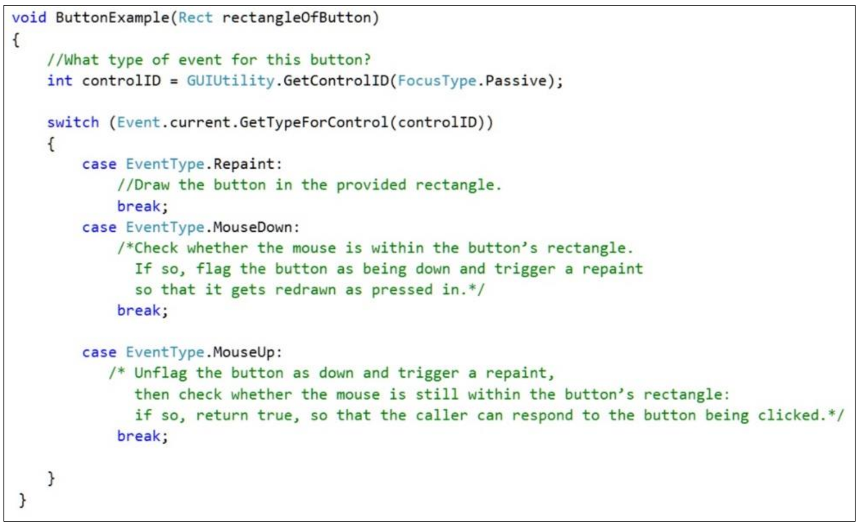
Figure 7. IMGUI flow-tree (Unity Technologies 2015)

As can be seen in Figure 7, what happens inside a GUI.Button-method depends on the current event, some of which depends on user's action. The most commonly used events are explained in Table 1.

Table 1. Event Types

Name of EventType	Set condition
EventType.MouseDown	Set when the user has just pressed a mouse button down.
EventType.MouseUp	Set when the user has just released a mouse button.
EventType.KeyDown	Set when the user has just pressed a key.
EventType.KeyUp	Set when the user has just released a key.
EventType.DragUpdated	Set when Drag & drop operation updated.
EventType.DragPerform	Set when Drag & drop operation performed.
EventType.Repaint	Set when IMGUI needs to redraw the screen.

Controls such as buttons and input fields are called inside OnGUI-method. Inside control methods, exists a switch-case statement that determines how each control acts during different events. A simplified version of the content of button-method can be seen in Figure 8. (Unity Technologies 2015.)



```
void ButtonExample(Rect rectangleOfButton)
{
    //What type of event for this button?
    int controlID = GUIUtility.GetControlID(FocusType.Passive);

    switch (Event.current.GetTypeForControl(controlID))
    {
        case EventType.Repaint:
            //Draw the button in the provided rectangle.
            break;
        case EventType.MouseDown:
            /*Check whether the mouse is within the button’s rectangle.
              If so, flag the button as being down and trigger a repaint
              so that it gets redrawn as pressed in.*/
            break;

        case EventType.MouseUp:
           /* Unflag the button as down and trigger a repaint,
              then check whether the mouse is still within the button’s rectangle:
              if so, return true, so that the caller can respond to the button being clicked.*/
            break;

    }
 }
```

Figure 8. IMGUI button example

3.2 Editor Assembly

Editor scripting is not too different from normal Unity scripting, it uses the same programming languages and can do anything that normal Unity script can. There are however some things that must be taken into account when writing editor scripts.

First, all editor scripts belong to a different assembly as other Unity code. A normal game code is a part of CSharp-assembly, while editor scrips are a part of CSharp-Editor-assembly. Editor-assembly will not be included in built version of the game, for this reason, scripts inside CSharp-assembly should never refer to a script inside Editor-Assembly, doing so will cause an error during the build process (Tadres 2015, 28). Editor-assembly scripts, however, can safely refer CSharp-Assembly scripts.

To include a script in Editor-assembly, the script file must be stored in a folder named “Editor” or its subfolders. There can be multiple Editor-folders. Scripts included in Editor-Assembly can use UnityEditor-namespace, which includes API necessary for editor extensions. Besides additional functionality provided by Unity Editor API, editor scrips function like regular scripts. (Tadres 2015, 28-29.)

3.3 Serialization

Serialization means a process where data is translated from one format to another for transfer and storage purposes. The opposite is called deserialization, in which serialized data is translated back to its original format (Kogent Learning Solutions 2009, 514). Unity uses serialization for many things, such as garbage collection, object instantiation, and prefab management. Unlike most of the scripts in Unity, which are written in C#, Unity serialization is written in C++ (Unity Technologies 2012a). Unity serialization system is very crucial for editor extensions, which need to modify and save data.

Unity serialization affects any serializable class that is referred by UnityEngine.Object-objects. UnityEngine.Object is a base class for most built-in objects in Unity. Most commonly serialization is called when a user enters or exits its play-mode in the Editor (Figure 9). When the user presses the play button to test the game, Unity reloads all of its mono assemblies, which destroys all user-defined data in UnityEngine.Objects. To preserve the data, Unity serializes every UnityEngine.Object in C#-side and saves the serialized data to the C++-side of Unity, where the data will be unaffected by C#-assembly reload. After serialization, all data on C#-side is destroyed and later re-created with the data stored in C++-side. For the user, this process is normally unnoticeable, as objects before and after serialization seem identical. (Unity Technologies 2012a.)

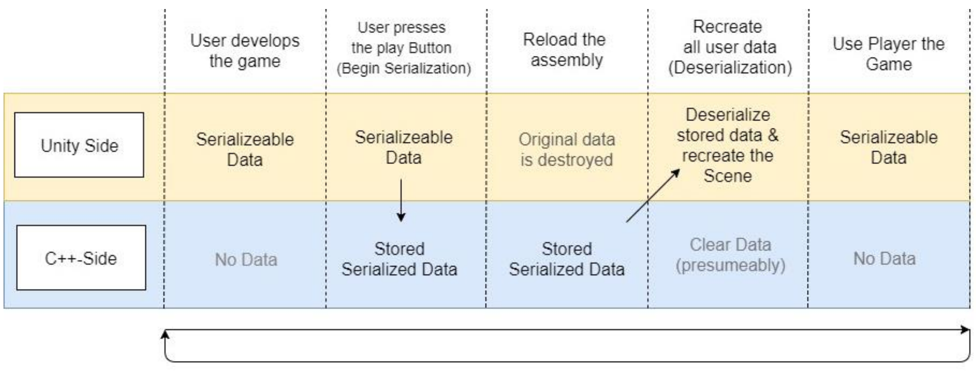
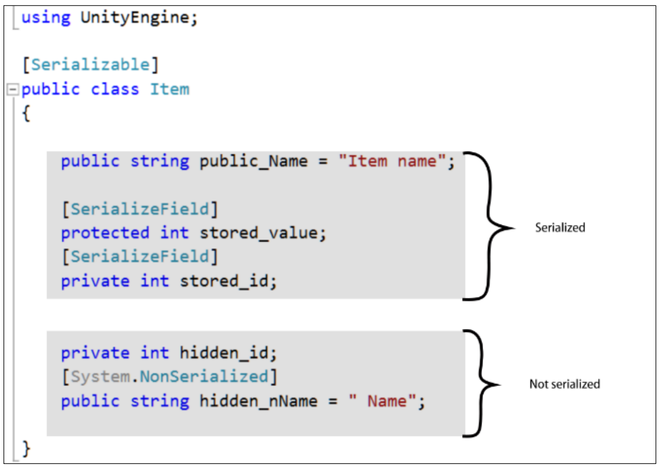


Figure 9. Serialization data flow

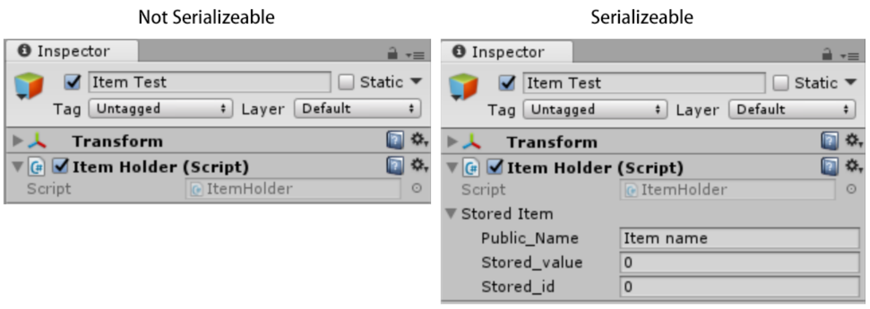
Unity can serialize any object that inherits from UnityEngine.Object as well as any serializable class inside that Object. Serialization works for all common primitive data types such as integers, strings, and arrays. Any non-abstract custom class can be serialized as demonstrated in Figure 10. To make a custom class serializable, the class must be preceded with Serializable-attribute. When a custom class is marked as serializable, all public serializable fields of the class are serialized by default (Unity Technologies 2012a). To serialize private or protected fields developer must add Serializefield attribute above the field (Tadres 2015, 76). Public fields can also be marked with a NonSerialized attribute, this means that serialization will ignore these fields during serialization and fields are reverted to back to default values (Wagner 2010,159). NonSerialized attribute is used with fields that contain temporary data that doesn't need to be saved.



```
using UnityEngine;

[Serializable]
public class Item
{

    public string public_Name = "Item name";

    [SerializeField]
    protected int stored_value;
    [SerializeField]
    private int stored_id;


    private int hidden_id;
    [System.NonSerialized]
    public string hidden_nName = " Name";
}
```

Figure 10. Serialized class

For editor extensions, marking class serializable is important. Because all public instances of a serialized class are also exposed in Inspector-window as shown in Figure 11. Now all changes to instances of the class will also survive the serialization process. Serializable fields can also be hidden from Inspector with HideInInspector-attribute. (Tadres 2015, 76.)



Not Serializeable:

Item Test
Tag Untagged Layer Default
Item Holder (Script)
Script ItemHolder

Serializeable:

Item Test
Tag Untagged Layer Default
Item Holder (Script)
Script ItemHolder
Stored Item
Public_Name Item name
Stored_value 0
Stored_id 0

Figure 11. Serialized class in Inspector

There are several limitations that need to be taken into account when dealing with the Unity serialization. For example, Unity serialization cannot serialize objects marked as static, constant or read-only (Unity Technologies 2014). One of the most prominent limitations in Unity serialization system in terms of editor extensions is the lack of support for polymorphism for custom classes. This problem is best described in the example in Figure 12 below.

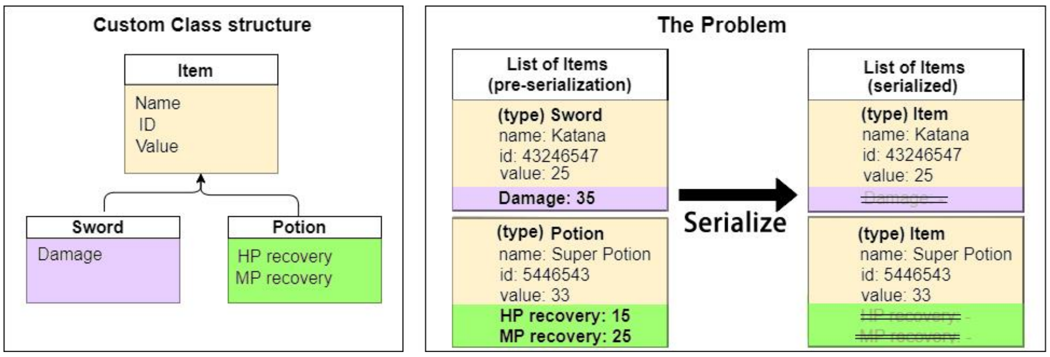


Figure 12. Polymorphism problem

In this example, we have a simple polymorphic class structure with classes Sword and Potion, both derived from parent class Item. If one wants to create an editor tool, that creates items and stores them in a chest, for example, one naturally wants to store all items as a single list of Items. This is where Unity serialization problem comes in. When the items in the list are serialized, they are serialized as Items, ignoring any data that belongs to its child classes Sword and Potion. Deserialized objects are naturally recreated as Items, but now without the data which belong to their original classes, Sword or Potion. The data is simply gone and unrecoverable and all that remains is a generic list of Items. Only classes not affected by this issue are classes derived from UnityEngine.Object (Unity Technologies 2014).

Lack of polymorphism support poses a huge problem for extensions that create and modify data. There is no way to make the exact solution in Figure 12 to work with Unity serialization, but there are several ways to work around the problem. However, each “solution” to the problem brings with it a whole bunch of new problems. This chapter explores three possible solutions.

The first solution is obvious: To not use polymorphism at all. Instead of using a class structure with inheritance, it is worth considering condensing the whole class structure to a single superclass. As any programmer can attest to, this is generally a bad idea. For very simple class structure this can be a valid solution, but even for a slightly complicated class structure, this solution is hardly even worth considering. Compressing all behavior of class structure to single class makes the code more complicated and closes doors on many possibilities that polymorphism gives to a program.

The second solution (Figure 13) is to make sure that when an object is stored, it is stored as its respective type. The reason for the whole problem in Figure 12 is that Sword and Potion were saved and serialized as Items. If objects are saved in separate lists dedicated to the specific type, the objects are serialized correctly. These separate lists are later combined when serialization is no longer an issue. This workaround possesses the disadvantage of the need to manage a potentially huge number of lists for each serialized type as well as saving and combining the lists. Like previous solutions, this solution is viable for a simple class structure but becomes harder to manage the bigger the class structure is.

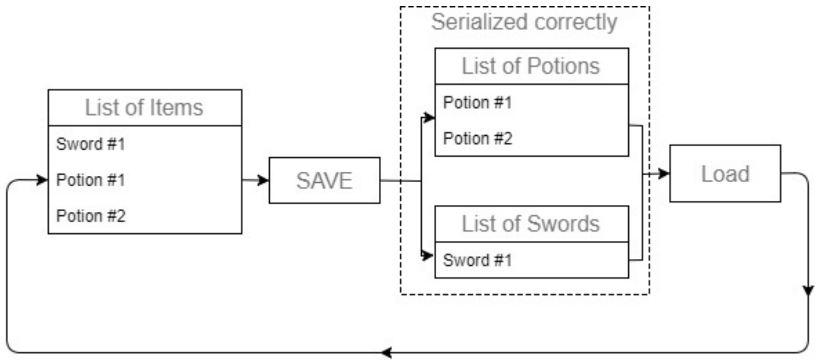


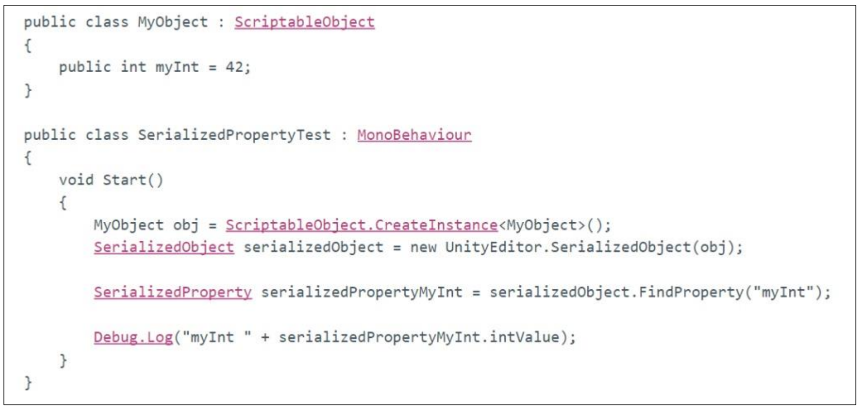
Figure 13. A list Serialization

The final solution discussed here is making class structure derive from ScriptableObject-class. ScriptableObject is a class derived from UnityEngine.Object class and is not therefore affected by the problem. ScriptableObject is a special class that functions like any script, but instances of the script must be saved on the disk as files (Tadres 2015, 189-190). All data derived from Scriptable objects are serialized as expected. The downside to this solution is that for every single item created, it is necessary to create a new ScritableObject-file. Not only does this increase the number of files in the project, but also makes it necessary for editor extension to manage all created files. But with proper file management, this solution can be very beneficial. One last issue with this solution is that values inside ScritableObjects cannot be exposed inside MonoBehaviour-inspector. To accomplish this one must create a custom property drawer for the ScriptableObject-class. This process is explained in chapter 6.2.2 (Unity Technologies 2014.)

As discussed in this chapter, understanding Unity serialization and its limitations is very important when creating complex editor extensions. Unfortunately, there is no perfect solution to get around all limitations that Unity serialization provides, but by understanding how the system works one can come up with many solutions to the problem. What solution is the best for each situation depends on many factors such as data complexity, performance needs and how easy it is to implement. Is creating editor extension worth any downside it causes? This question is vital when developing an extension.

3.4 SerializedObject and SerializedProperty

Default Unity Editor tools rely heavily on serialization. Instead of creating a separate editor for every component and object individually, Unity is able to generate generic editors by using SerializedObjects and SerializedProperties. SerializedObjects are serialized representations of any UnityEngine.Object. SerializedProperties in turn, are serialized representation of properties inside SerializedObjects. SerializedObjects are used by Editor-class to generate all default inspectors for user-made components (Unity Technologies 2017a). Accessing SerializedProperties is very different from normal C#-scripting. To refer to property one must use method called FindProperty, which finds properties based on their name in the code, asking for non-existent property results in error (Tadres 2015, 105). As demonstrated in Figure 14, this process is very different from traditional C#-programming. The example prints the value of myInt to the Unity Debug console.



```
public class MyObject : ScriptableObject
{
    public int myInt = 42;
}

public class SerializedPropertyTest : MonoBehaviour
{
    void Start()
    {
        MyObject obj = ScriptableObject.CreateInstance<MyObject>();
        SerializedObject serializedObject = new UnityEditor.SerializedObject(obj);

        SerializedProperty serializedPropertyMyInt = serializedObject.FindProperty("myInt");

        Debug.Log("myInt " + serializedPropertyMyInt.intValue);
    }
}
```

Figure 14. SerializedObject editor script

The interesting part about these serialized-classes is that they are completely generic (Tadres 2015, 105). Modifying SerializedProperties is done with generic accessor control, called PropertyField. PropertyFields automatically generate an appropriate controller for the variable; strings get textboxes, booleans get checkboxes etc. Even though SerializedProperties are generic, they do hold information about what kind of variable each property originally was. By using this information stored in a propertyType variable, Unity determines what type of accessor control should be used (Unity Technologies 2017b). For example, when propertyType is ProperyType.Boolean, Unity uses the boolean accessor control: a checkbox. This process is repeated to all properties inside serializedObject as well as every serializable property inside serializable custom classes. (Smith & Queiroz 2015, 514.)

Despite being very different from normal C#-code, there are many advantages to using SerializedObjects to modify object instead of modifying the object directly. SerializedObject have inbuilt functionalities such as Undo, multi-object editing, and prefab-management. It is important to note that all this functionality can be manually implemented to tools, that do not use SerializedObject. (Tadres 2015, 104-105.)

While Unity does recommend using SerializedObjects whenever possible, there are legit reasons not to use the feature. Disadvantages with serializedObjects are caused by their generic nature. Because SerializedProperties only contain data, but not methods of the class instance, it can be challenging to implement some class specific behavior for custom tools. Especially when making tools that manage complex data trees and use custom classes and methods. In some cases, modifying object directly and creating Undo-functionality manually is a better option. (Meier 2014.)

4 EDITOR-WINDOWS

This chapter discusses different ways editor scripts are called and where editor tools can be drawn. While simple macros can be called using a hotkey or menu-click, most editor extensions require some sort of container on the screen, where editor tools are drawn. These containers include different menus and windows. Some of these containers are whole new windows created via code, but it is also possible to insert custom tools inside existing Unity UI-elements such as menus and Inspector window.

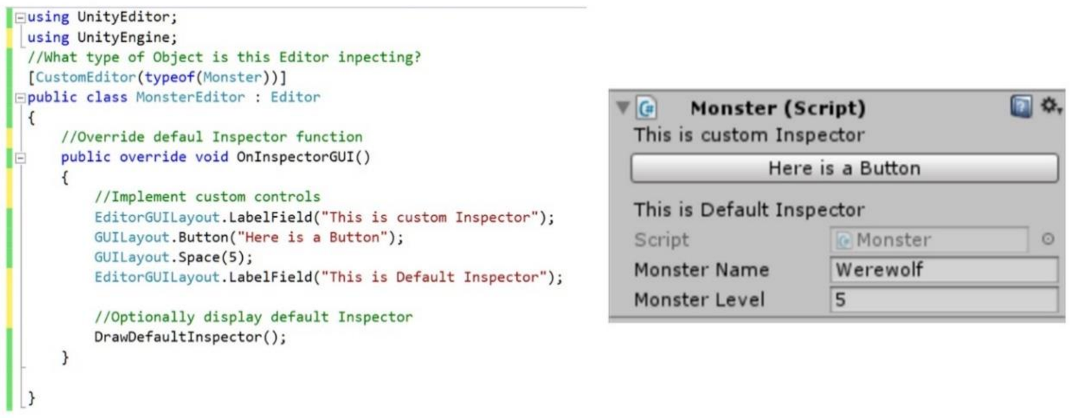
4.1 Inspector-Window

Inspector window is used to display and modify the information of selected UnityEngine.Object, in the context of Inspector windows this means either Component or MonoBehaviour script. With Inspector window, the user can and modify properties of the selected object (Pierce 2012, 29-30). It is important to note that in UnityEditor each UnityEngine.Object can have a custom Editor-class object associated with it. When Object is displayed in Inspector window, Unity calls its respective Editor class, which tells how Object should be drawn in Inspector. Normally Editors for user made classes are automatically generated, but this default behavior can be overridden by creating a new Editor derived-class which affect all objects of the specific class. (Unity Technologies 2017a.)

The default Inspector for MonoBehaviour only allows displaying and modifying public or serializable fields within the scripts (Cogut 2015, 143). Default Inspector provides simple controls to modify data; Input fields for string-data, a Colour picker for Colour-data and so on. But functionality provided by default controls is fairly limited. What if a developer wants an integer data to be within certain range or have a button that reverts some data back to default values? For more complex behavior, the developer can either modify default Inspector's functionality with PropertyAttributes or override existing inspector for the MonoBehaviour with Custom Editors. All custom editor behavior can also be temporarily disabled by enabling Debug-mode in Inspector window.

4.1.1 Custom Editor

Every object derived from UnityEngine.Object-class can have its own Editor-class, which override how the Object is displayed in the editor as well as affect the Scene View. By overriding the OnInspectorGUI-method of Editor-class, it is possible to draw completely customized UI, specifically for the class. These self-made UIs can have functionalities that are far beyond what default Inspector can accomplish. (Tadres 2015, 107,) To create Custom Editor-class, one must first create a new script under Editor-folder. The new script must implement both UnityEditor and UnityEngine -namespaces, derive from Editor-class and be preceded with CustomEditor attribute before it is declared. CustomEditor attribute tells Unity which objects Inspector the Editor-class overrides (Smith & Queiroz 2015, 514). Figure 15 shows a bare-boned sample of editor script for MonoBehaviour-class Monster. As can be seen it is also possible to draw default inspector, by using DrawDefaultInspector-method (Tadres 2015, 82).



```
using UnityEditor;
using UnityEngine;
//What type of Object is this Editor inpecting?
[CustomEditor(typeof(Monster))]
public class MonsterEditor : Editor
{
    //Override defaul Inspector function
    public override void OnInspectorGUI()
    {
        //Implement custom controls
        EditorGUILayout.LabelField("This is custom Inspector");
        GUILayout.Button("Here is a Button");
        GUILayout.Space(5);
        EditorGUILayout.LabelField("This is Default Inspector");

        //Optionally display default Inspector
        DrawDefaultInspector();
    }

}
```

Monster (Script)
This is custom Inspector
Here is a Button
This is Default Inspector

Script	Monster
Monster Name	Werewolf
Monster Level	5

Figure 15. Custom Inspector

Editor-class can access properties of the inspected object in two ways: by referring the object itself or by accessing the serialized representation of the object. Editor-class automatically serializes the inspected object and stores it to private serializedObject-variable, which is SerializedObject-type. Direct reference to the inspected object is stored in target-variable, which is an Object-type. If multiple objects of the same type are chosen, each object is stored inside the targets-array. To apply multi-object editing, the editor class must be marked with CanEditMultipleObjects attribute. (Unity Technologies 2017a). As discussed in chapter 3.4, using SerializedObjects is recommended, because of automatic Undo and multi-object-editing -functionality.

4.2 Editor Window-class

Editor Window-class is used to create tools, that exist outside of Inspector windows. Editor Windows-tools can be dragged, docked and resized at will, though this behavior can vary depending on the type of editor window in question as well as parameters given to the window (Tadres 2015, 111).

Unlike Inspectors, which inherit from Editor-class, Editor Windows is its own separate class. Different from Editor class, Editor Windows are not tied to any specific object or class. To modify objects inside an Editor window, a developer must supply editor windows with object references manually. An object reference can be passed to EditorWindows using methods like GameObject.Find, Resources.Load, static variables or manually loading the asset from the project. Unlike inspectors, Editor Windows can be used to edit multiple different objects at once, even if they are not the same type. This gives developers the great freedom to create just about any kind of tools necessary.

Besides differences mentioned above, Editor Windows function like custom Inspectors: Both have OnGUI-loop, where tools are created, both use same IMGUI-elements and structures. Because both Editor-types are so similar, migrating tools from one type of editor to another is an easy task. One can draw inspector of any UnityEngine.Object inside Editor Window, by using CreateEditor-method, allowing the developer to reuse editors in multiple locations (Unity Technologies 2017a). Editor Windows are opened by using GetWindow-method. This method can be called by other editor scripts or custom menu-commands using MenuItems (see chapter 4.4).

4.3 Scene View

Scene View is a 3D preview of the game, that user can interact with. Normally this view is used to select and translate objects. Unity Editor makes it possible to alter the behavior of Scene View. Not only can any normal IMGUI-tools be drawn to the screen, Scene View also has a set of 3D tools called Handles, that can only be used inside the Scene View. Compared to menu/button oriented tools built in Inspectors and Editor windows, Scene View tools can be highly intractable and allow intuitive mouse interaction with the scene. Scene View extensions can also be used to display additional information to the user, like displaying weapon ranges for example.

There are at least three ways to draw content to Scene View. The most common way is to use override OnSceneGUI-method of the Editor-class. Since Editor-class is used by the Inspector, any tools implemented this way should be specific to object currently being inspected. Scene View tools are drawn to the Scene View open and close along with the Inspector (Tadres 2015, 140). The second way is to use Gizmos, mostly used for visual aids and debugging (Thorn 2015. 63). The third way is to use undocumented onSce-neGUIDelegate, which is a delegate that belongs to the SceneView-class. On-SceneGUIDelegate is called every time Scene View is redrawn, allowing im-plementation of interactive editor tools that are constantly in view regardless of the selected object.

4.3.1 Gizmos

Gizmos are visual only elements. Unlike most editor tools, Gizmos are defined in UnityEngine-side by implementing OnDrawGizmos and OnDrawGiz-mosSelected -methods inside MonoBehaviour-scripts. Gizmos have an ad-vantage over normal editor scripts since elements created inside OnDrawGiz-mos are always drawn to Scene View unless the actual object is disabled or when the inspector is collapsed. When an object is selected OnDrawGiz-mosSelected is called instead of OnDrawGizmos. OnDrawGizmosSelected al-lows the creation of visually different tools for the selected object, making it easy to distinguish and select objects. (Thorn 2015, 63-68.)

As can be seen in Figure 16 Gizmos can be used to draw simple shapes like lines, spheres, and cubes. The only complex shape that can be drawn is a mesh. The appearance of drawn gizmos can be a solid color or wireframe. Ad-ditionally, Gizmos can also draw icon and textures. Icons must be located in-side folder Assets/Gizmos folder inside the project. (Tadres 2015, 54-59.)

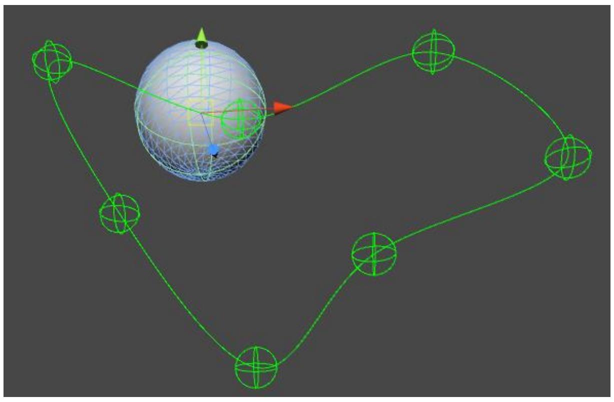


Figure 16. Gizmos (Unify Community Wiki)

Gizmos are useful for debugging. They display things like weapon ranges, the field of view and other things normally invisible. Since Gizmos are not intractable, they can’t really be used for much and its functionality is limited compared to Handles-class. They are mostly used in conjunction with other editor tools as a visual aid. (Thorn 2015, 63.)

4.3.2 Handles

Handles are a Scene View specific tool. Handles can do everything Gizmos can, which include drawing different shapes as visual aids, but also draw interactable controls to the Scene View. Normal IMGUI-tools can be drawn inside Scene View with Handles, by creating a 2D block inside Scene View with BeginGUI and EndGUI-methods and drawing tools inside like you would do any OnGUI-method. Unlike Gizmos, Handles disappear when the object is deselected (unless they are drawn using onSceneGUIDelegate). (Tadres 2015, 159.)

The most prominent advantage of Scene View tools is that they can be interactive. Handles-class includes several interactive elements, that respond to mouse interaction. Everyone who uses Unity is familiar with common handles, shown in Figure 17. Normally handles are used to alter the position, scale, and rotation of game objects (Tadres 2015, 159). These handles can also be used to modify any value, either directly or indirectly. Handles directly modify vectors, quaternions, and floats, but since a script can be used to react to handle interaction, scripts can react to changes any way necessary. Custom handles can be therefore used to change data that has no actual visible representation in the game world.

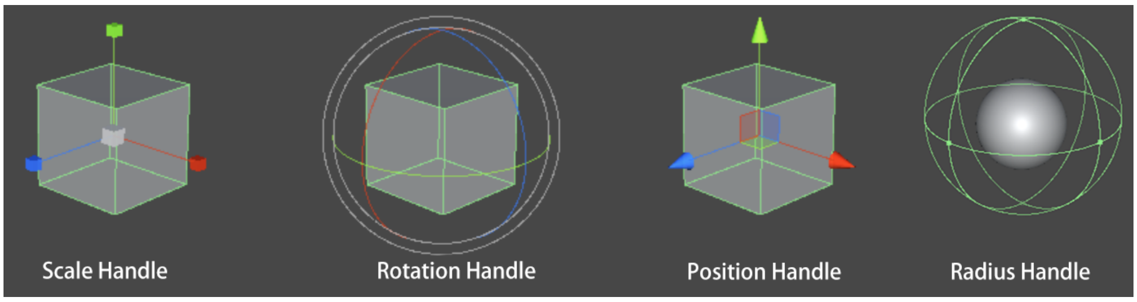


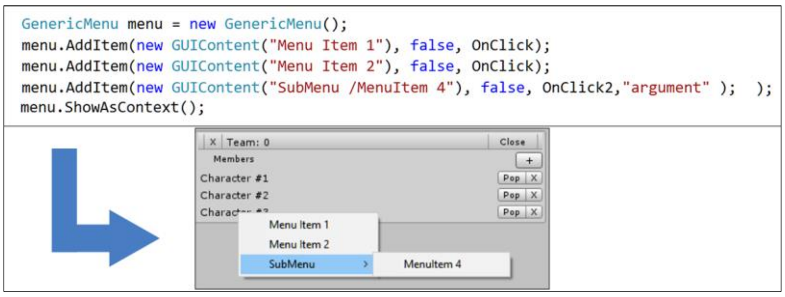
Figure 17. Default Handles

Handle-elements handle mouse interaction automatically, but to implement whole new interactive tools, it becomes necessary to detect and handle mouse interaction manually. Handling mouse interaction is a combination of events discussed in chapter 3.1 and HandleUtility-class. HandleUtility-class is used to translate coordinates between 2D-space on the screen and 3D-space in the game world. Scripts use events to detect user interaction and act accordingly. Events detect both mouse and keyboard events. (Tadres 2015, 145.)

4.4 Menus

Menu-elements can be used to call any method from menus, making them useful for running macros. Replacing manual actions with macros speeds up the development process and reduce a risk of mistakes. Menu items can also be used to open editor windows and creating assets.

Creating new menus inside editor tools is done by using GenericMenu-class. GenericMenus are created inside editor scripts along with other UI elements. Creating custom menus usually happens after a button press, typically right mouse or UI button. Menus are created either under mouse location (Figure 18) or inside a predetermined rectangle on the screen. (Unity Technologies 2017e.)

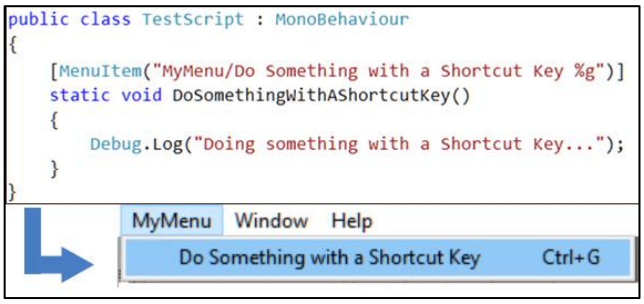


```
GenericMenu menu = new GenericMenu();
menu.AddItem(new GUIContent("Menu Item 1"), false, OnClick);
menu.AddItem(new GUIContent("Menu Item 2"), false, OnClick);
menu.AddItem(new GUIContent("SubMenu /MenuItem 4"), false, OnClick2,"argument" );  );
menu.ShowAsContext();
```

Figure 18. Creation of GenericMenu

Adding menu items is done by using AddItem-method. As demonstrated in Figure 18, AddItem-method accepts following arguments: location and name of the item in the menu, boolean, onClick-callback, and optional generic argument for the callback. (Unity Technologies 2017e.) To improve readability of menus, one can use AddSeperator to create separating slashes between elements.

Modifying built-in Unity menus is done with attributes Menu Item and Context Menu. These attributes precede a method which they invoke when clicked. Menu Items add elements to Unity main menu and Inspector context menu. Context Menu can only add elements to the Inspector-window (Tadres 2015, 95). Since context menus are a part of the Inspector, they naturally work on an object-by-object basis. Thanks to this, the Context menu can call non-static methods of the inspected object. Therefore, Context menu behavior can vary, depending on a state of that object. MenuItems, on the other hand, can only call static methods, like in Figure 19. (Smith. 2015, 522.)



```
public class TestScript : MonoBehaviour
{
    [MenuItem("MyMenu/Do Something with a Shortcut Key %g")]
    static void DoSomethingWithAShortcutKey()
    {
        Debug.Log("Doing something with a Shortcut Key...");
    }
}
```

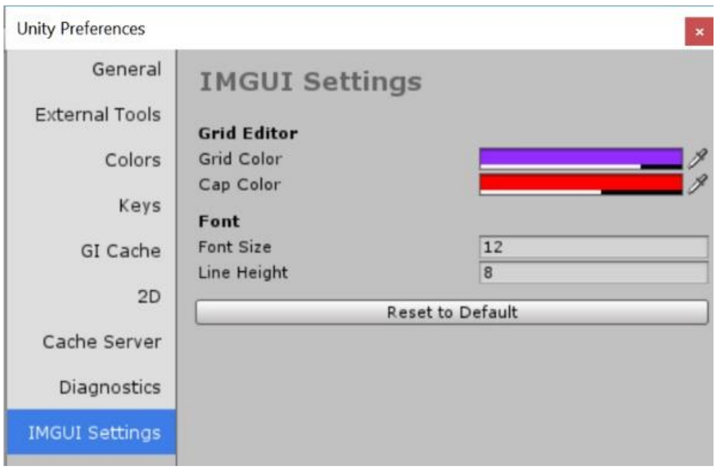
Figure 19. Creation of MenuItem

Location of menu elements is determined by a path, which is given as an argument to the attribute. To add an element to GameObject-menu, a path for the element would be “GameObjects/Item Name”. Most menus allow a creating of submenus, which makes menus more organized. To create a submenu, one must create a path as follows: “ParentMenu/Submenu/Item Name”. (Dickinson 2015. 257.)

One MenuItem specific functionality is an ability to add shortcuts as a part of the menu path. Unity uses special characters to represent the modifier keys: % for ctrl/cmd, # for shift, & for alt, and underscore if shortcut doesn’t use modifier keys. Shortcuts are also displayed in the menu. (Dickinson 2015, 257.)

4.4.1 Settings

More complex the tools become, more it will be necessary to allow users to alter the behavior of the tools to fit their needs. For example, if we have a tool that draws green lines to Scene View, it could be hard to see the line in a green environment. It would be possible to change the line color via code, but the more user-friendly approach to this is to create a settings-element for it, in Preferences-Window (Figure 20). Unity Preference window is a normal settings windows found in most programs.



IMGUI Settings

Grid Editor
Grid Color
Cap Color

Font
Font Size 12
Line Height 8

Figure 20. Custom Preference Section

A developer can add their own sections to Preference windows using PreferenceItem attribute. PreferenceItem is similar to MenuItem; it is placed above static function and takes a path as an argument. Static method following the argument is used to generate the UI inside the Preferences Windows. Preferences-window uses the same IMGUI-tools as Editor Windows and Inspector. (Unity Technologies 2017f.)

Preference Windows can be used to modify static values, but since static values do reset during serialization, they are not an ideal option for preferences. To save persistent data Unity has inbuilt local storages for editor settings, called EditorPrefs and SessionState. Both these classes are used to save and load simple data. Difference between the two classes is that data saved by SessionState reset when the program is closed, whereas data saved by EditorPrefs persist between Unity sessions (Unity Technologies 2017g). Saving and loading data to local storage is only possible for 4 types of variables: strings, integers, floats and Booleans (Tadres 2015, 224). Getting around this limitation is possible by serializing the object as a JSON string and save that to the storage.

5 LAYOUT AND APPEARANCE

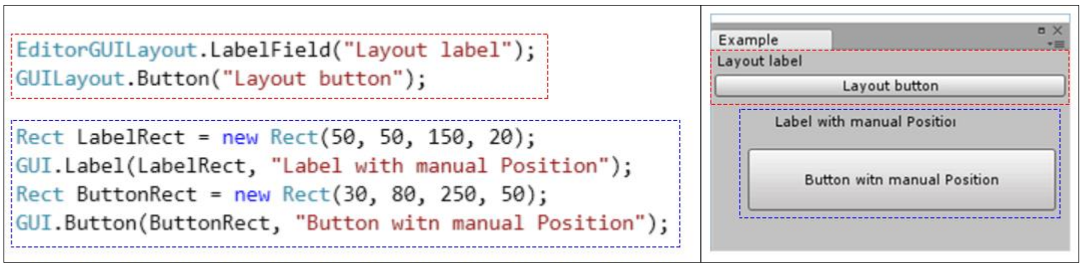
When designing tools, it is important to take in consideration their usability. How visually clear the tool is to use, is very important, especially for the tools that are sold in Asset Store. While in-house tools do not need to be visual masterpieces, a clear, readable UI makes a tool easier to use and learn. Tools can easily be made more organized by improving the layout or adding visual elements like images and icons.

In general, layout-wise the IMGUI-system is like to HTML-markup language, used to create web pages. Both Unity Layout System and HTML work on a box-model system, where all elements are considered rectangular boxes (Vodnik 2015, 84). The boxes determine the size and position of the content. With box-model it is easy to determine where one element ends and other begins.

5.1 GUI and GUILayout

In past IMGUI system was also used to create in-game UI. To keep game tools separate from editor tools, IMGUI-controls are separated to both UnityEngine and UnityEditor -namespaces. In-game UI uses GUI-class and Editor UI uses EditorGUI-class. Both classes are similar to each other but have some tools that other does not. Since functionality between two classes is indistinguishable both classes can be used in inside Editor. (Tadres 2015, 87.)

There are also two variants of each IMGUI class: GUILayout and EditorGUILayout. These classes are auto-layout versions of GUI and EditorGUI-classes respectively. These layout versions include the same tools as their base counterparts, with identical functionality. The main difference is that GUILayout-elements automatically determine their location and size in UI, whereas base-GUI -tools need coordinates in which the elements are drawn (Tadres 2015, 81). Figure 21 demonstrates the difference between the two approaches.

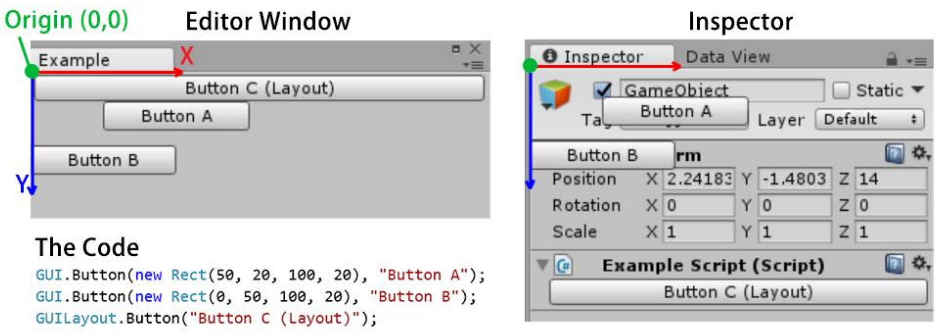


```
EditorGUILayout.LabelField("Layout label");
GUILayout.Button("Layout button");

Rect LabelRect = new Rect(50, 50, 150, 20);
GUI.Label(LabelRect, "Label with manual Position");
Rect ButtonRect = new Rect(30, 80, 250, 50);
GUI.Button(ButtonRect, "Button witn manual Position");
```

Figure 21. GUI vs. GUILayout

Non-layout GUI controls require a rectangle, which dictates its size and position relative the top-left corner of the window. The rectangle is either predetermined by a developer like in Figure 21 or calculated like any variable. Calculating the rectangle allows controls to be positioned differently depending on circumstances. Unlike GUILayout-controls, whose positions are determined by other controls, the position of GUI controls is completely unrestricted, making them ideal for objects that need to be dragged for example. (Doran 2014, 56.)

The problem with GUI is that calculating coordinates can be a hassle. This is especially problematic with custom Inspectors, are not aware of their own size or position. As demonstrated by Figure 22, GUI controls in Editor window works as expected, but in Inspector the controls are overflow outside of the expected area. As evident the GUI-system consider the origin to be at the top-left corner of whole Inspector Windows, instead of just the custom Editor as one would expect.



The Code

```
GUI.Button(new Rect(50, 20, 100, 20), "Button A");
GUI.Button(new Rect(0, 50, 100, 20), "Button B");
GUILayout.Button("Button C (Layout)");
```

Figure 22. GUI-coordinate problem

If the same code is used on EditorWindow, GUI controls work more like expected, with correct positioning and overflow behavior. This is due to the fact, that EditorWindow-class is aware of both its size and position, which GUI uses to perform its calculations (Unity Technologies 2017h). This makes EditorWindow more suited for GUI-controls. GUI-controls are however used in Inspectors with PropertyDrawers (see chapter 6.2.2), where Unity provides the control with a rectangle where controls are drawn.

GUILayout calculates the position of each control, relative to the window and each other. At the start of every frame, the Layout event records all layout elements on the window and calculates their rectangles. Since Layout event runs before any other IMGUI-event, all other events can use this information. Mouse events use the information to check if the mouse has been clicked inside the controls and Repaint-event draws the actual visual controls inside the rectangle. (Unity Technologies 2015.)

By default, all layout-elements are ordered from top-to-bottom in order which they are declared in the script. Aligning controls horizontally is done with Layout-groups. Layout-groups are examined in more detail in next sub-chapter, but they are vital to understanding the example in Figure 23. Essentially Layout-groups tell the Layout-event how elements inside of it should be aligned. Figure 23 below visualizes how layout system sees controls. (Tadres 2015, 90-91.)

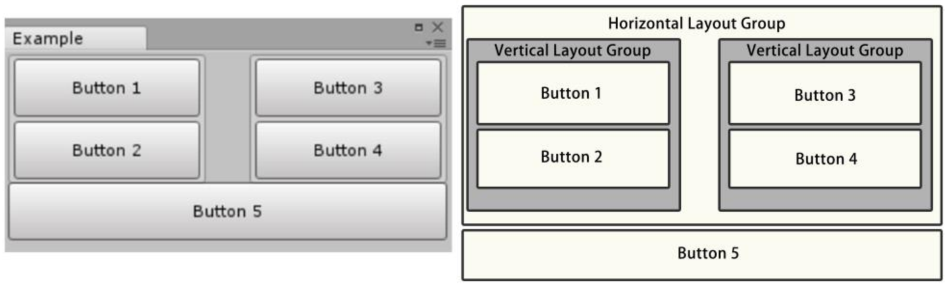


Figure 23. Layout graph

5.2 Element Groups

Element groups (Figure 24) can group multiple elements together. These groups can change how elements are displayed or they can simply be visual containers. Layout groups are an invaluable tool when organizing UI-elements as they can alter how the UI is structured and displayed. Layout groups are methods that tell the UI-system where the group begins and ends. For Horizontal groups, these methods are called BeginHorizontal and EndHorizontal.

All UI-elements between the begin and the end are considered as a part of the group, including nested groups. (Tadres 2015, 90.)

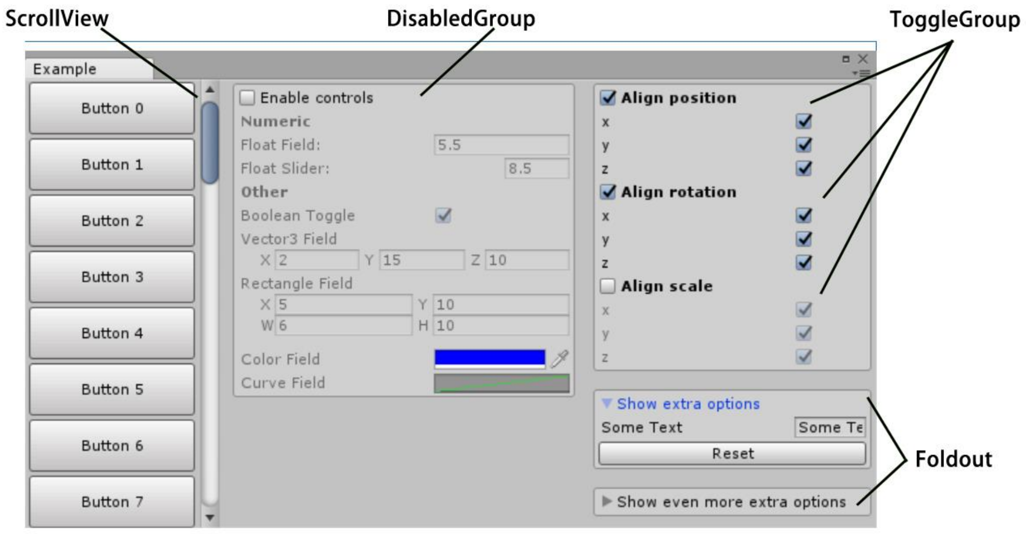
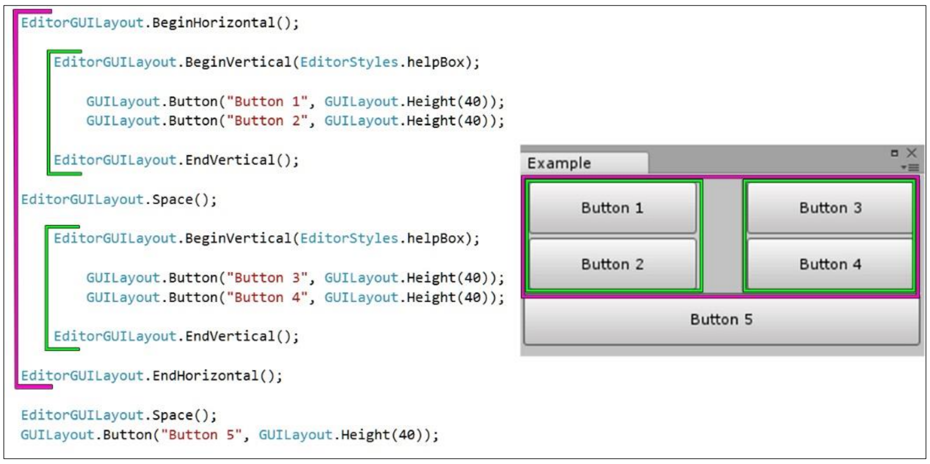


Figure 24. Common Element Groups

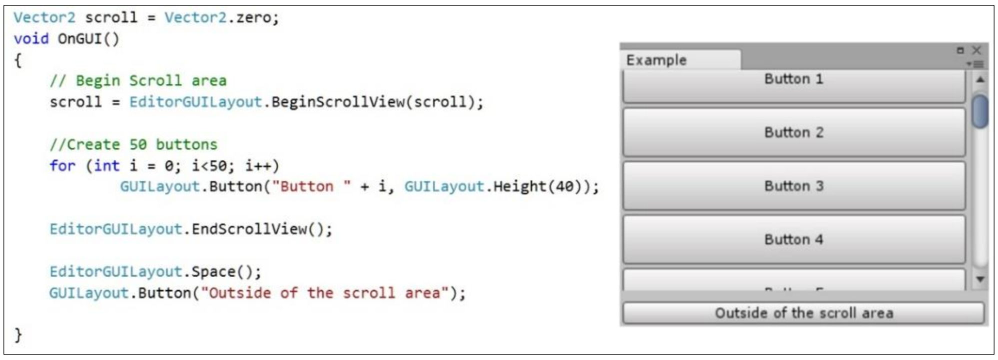
Among the most useful groups are the aforementioned Vertical and Horizontal layout groups. As can be seen in Figure 25, these groups can be nested with each other to create organized groups of controls. Groups can also be given styles, which can give them visually distinct look, further improving the readability of the editor (Tadres 2015, 92.)



```
EditorGUILayout.BeginHorizontal();

    EditorGUILayout.BeginVertical(EditorStyles.helpBox);

        GUILayout.Button("Button 1", GUILayout.Height(40));
        GUILayout.Button("Button 2", GUILayout.Height(40));

    EditorGUILayout.EndVertical();

EditorGUILayout.Space();

    EditorGUILayout.BeginVertical(EditorStyles.helpBox);

        GUILayout.Button("Button 3", GUILayout.Height(40));
        GUILayout.Button("Button 4", GUILayout.Height(40));

    EditorGUILayout.EndVertical();

EditorGUILayout.EndHorizontal();

EditorGUILayout.Space();
GUILayout.Button("Button 5", GUILayout.Height(40));
```

Figure 25. Vertical and Horizontal Layout Groups

As tool became bigger, it becomes hard to fit all elements inside a single window. To fix this one can add a scrollbar to a window by implementing ScrollView group (Figure 26). Scroll view automatically hides any overflowing controls and creates a scrollbar. Scroll view can be added to an entire window or just be a group on its own. Depending on a size of the content the scroll view can be horizontal or vertical. The current position of the scroll is stored in a variable, which gets updated by BeginScrollView-method. (Simon 2015, 21-23.)

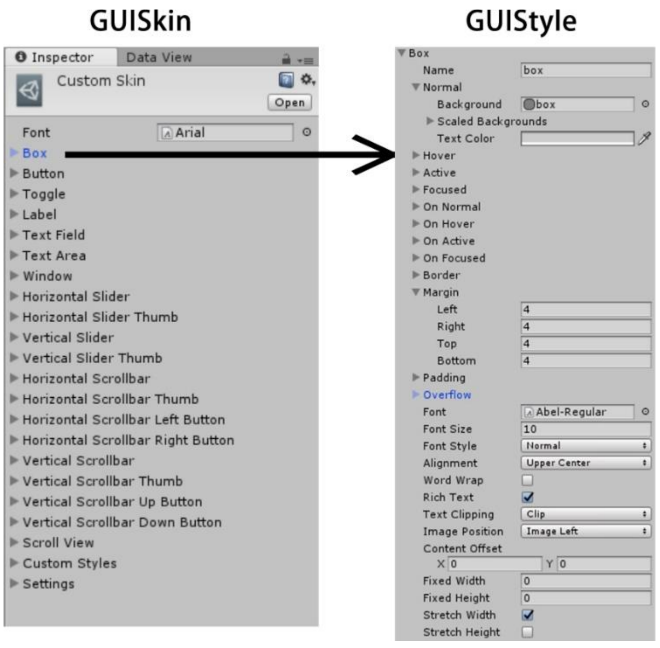


```
Vector2 scroll = Vector2.zero;
void OnGUI()
{
    // Begin Scroll area
    scroll = EditorGUILayout.BeginScrollView(scroll);

    //Create 50 buttons
    for (int i = 0; i<50; i++)
            GUILayout.Button("Button " + i, GUILayout.Height(40));

    EditorGUILayout.EndScrollView();

    EditorGUILayout.Space();
    GUILayout.Button("Outside of the scroll area");

}
```

Figure 26. Scroll View

5.3 GUIStyle and GUISkin

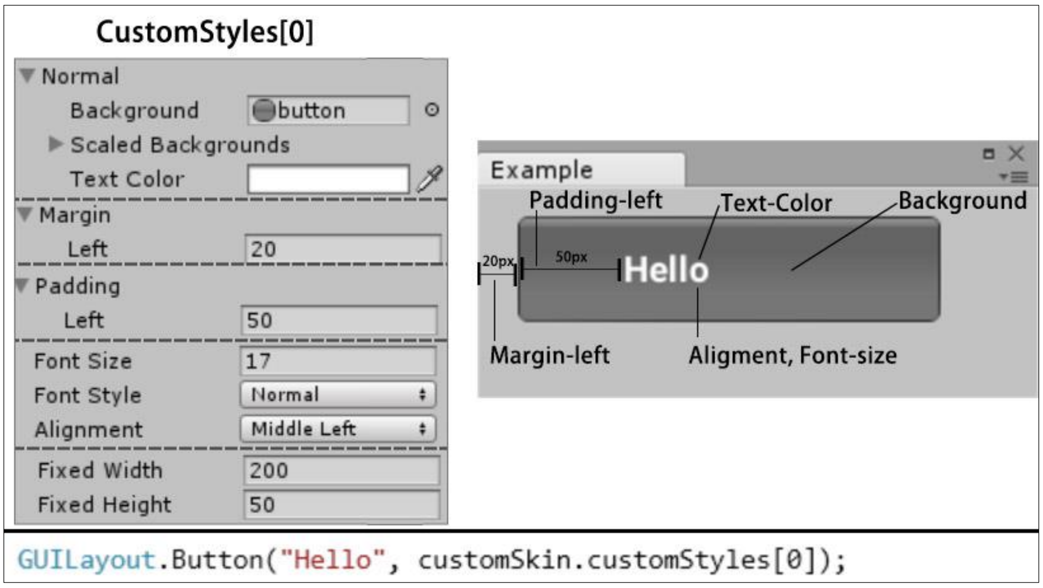
GUIStyles determine the visual styles of individual controls. GUI styles affect visual properties such as fonts, backgrounds, colors, and how the element responds to mouse interactions such as click and hovers. GUISkins (Figure 27) are separate files, containing a collection of GUIStyles and settings. In short, the script determines the structure and style of elements and GUIStyles determine the visual appearance of the content. Both sides can affect the size of the element. (Tadres 2015, 168-170.)



Name	box
Normal	
Background	box
Text Color	
Margin	
Left	4
Right	4
Top	4
Bottom	4
Font	Abel-Regular
Font Size	10
Font Style	Normal
Alignment	Upper Center
Word Wrap	☐
Rich Text	✓
Text Clipping	Clip
Image Position	Image Left
Content Offset	X 0 Y 0
Fixed Width	0
Fixed Height	0
Stretch Width	✓
Stretch Height	☐

Figure 27. GUI Skin

GUISkins include a collection of default styles for common controls like buttons, toggles, and labels. When a new GUISkin is implemented to a script, all controls use their respective GUIStyle by default, without the need to assign the style of each element separately. GUISkin also includes an array of Custom GUIStyles, which is a collection of user-defined styles that can be assigned to controls individually. Most controls accept GUIStyle as an argument, which allows overriding the default style with a custom style (Figure 28). (Tadres 2015, 179-181.)

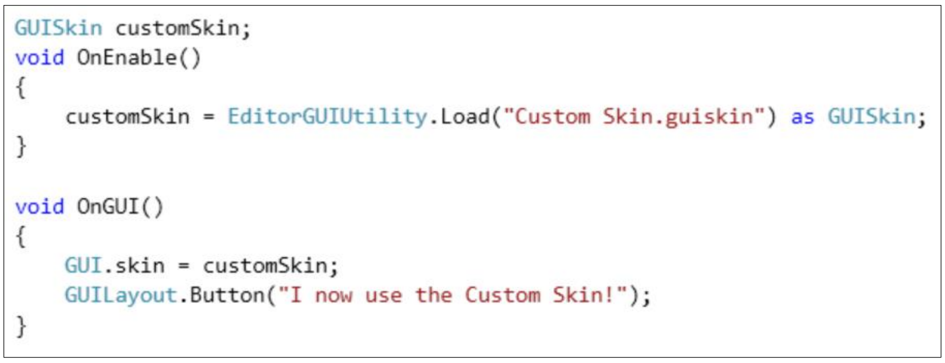

CustomStyles[0]

▼ Normal	
Background	button
▶ Scaled Backgrounds	
Text Color	
▼ Margin	
Left	20
▼ Padding	
Left	50
Font Size	17
Font Style	Normal
Alignment	Middle Left
Fixed Width	200
Fixed Height	50

```
GUILayout.Button("Hello", customSkin.customStyles[0]);
```

Figure 28 Custom style

In legacy UI, GUISkin could be assigned to UI via Inspector, but since that is not an option for editor most scripts, GUISkin-files must be manually loaded via script. There are several ways to accomplish this, but the most straightforward approach is using EditorGUIUtility.Load-method. Load-method can be used to load any asset located in folder Assets/Editor Default Resources (Sapio 2017, 19). Methods require the name of the asset in string form. Loading of the GUISkin should happen in OnEnable-method of the Editor-class (Figure 29), which runs before any other method as the object first becomes active. Applying the GUISkin to the entire editor is done by assigning a GUISkin to the skin-property of a GUI-class, this can only be done inside OnGUI-method.



```
GUISkin customSkin;
void OnEnable()
{
    customSkin = EditorGUIUtility.Load("Custom Skin.guiskin") as GUISkin;
}

void OnGUI()
{
    GUI.skin = customSkin;
    GUILayout.Button("I now use the Custom Skin!");
}
```

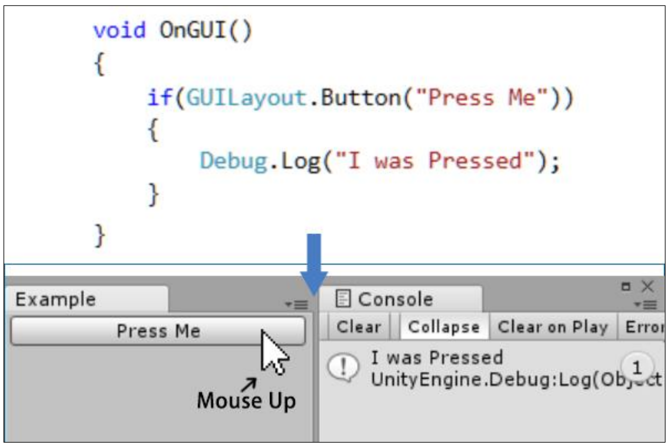
Figure 29. Loading GUISkin

6 UI CONTROLS

Most UI-elements in Unity Editor are familiar tools, seen in just about any piece of software. The user can trigger some functionality with a button or change some value by typing text in an input field. Some UI-elements are non-interactable, that exist to give user information, such as labels and images.

6.1 Basis Controls

Buttons are one of the most basic elements, used most Editor-tools. There are several different types of buttons across both GUI and EditorGUI-classes. The most basic button is oddly enough GUI-class exclusive-element. As seen in Figure 30 the button reacts to the user clicking the button, returning true when user released the mouse button. What happens when button returns true depends on the tool. Some example uses for buttons could be: Opening and closing windows, deleting elements or running macros, basically any complex action that cannot be accomplished by other UI-elements. (Doran 2014, 57.)

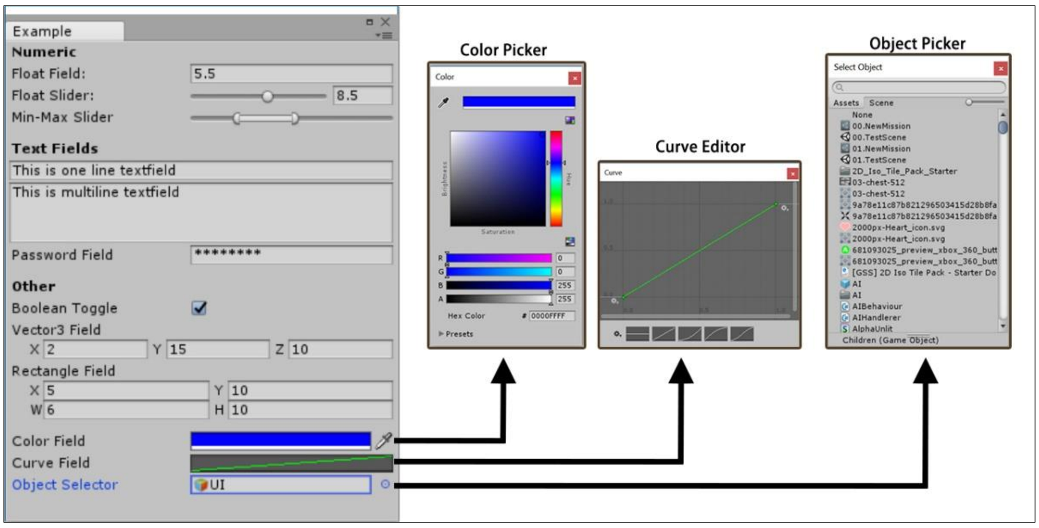


```
void OnGUI()
{
    if(GUILayout.Button("Press Me"))
    {
        Debug.Log("I was Pressed");
    }
}
```

```
I was Pressed
UnityEngine.Debug:Log(Object
```

Figure 30. Basic Button

Input controls can be used to change values of the variables in a script. There are many different Input controls for different data types such as integers string and vectors. String datatypes have text fields, booleans have check-boxes, colors have color selectors and so on. Some data types have several different input controls, for examples, numeric data types can be modified either by an input-field or a slider. Some controls open additional editor windows to make editing easier. Several different Input controls can be seen in Figure 31.



Example

Numeric

Float Field: 5.5
Float Slider: 8.5
Min-Max Slider

Text Fields

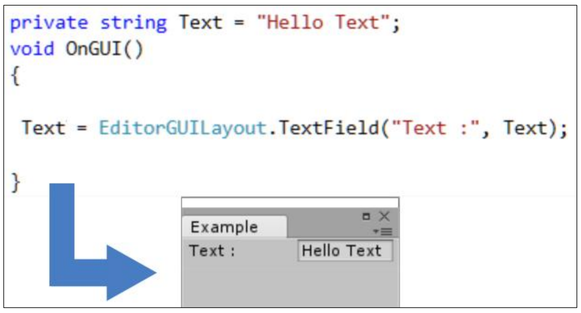
This is one line textfield
This is multiline textfield
Password Field ********

Other

Boolean Toggle ✓
Vector3 Field X 2 Y 15 Z 10
Rectangle Field X 5 Y 10 W 6 H 10
Color Field
Curve Field
Object Selector UI

Figure 31. Basic Editor controls

While all these controls are visually very different their implementation is mostly the same. Input controls are normal methods which, create the control, handle user input and return the final inputted value. An example of this can be seen in Figure 32, where string variable is modified using TextField-control. (Simon 2015, 15.)



```
private string Text = "Hello Text";
void OnGUI()
{

 Text = EditorGUILayout.TextField("Text :", Text);

}
```

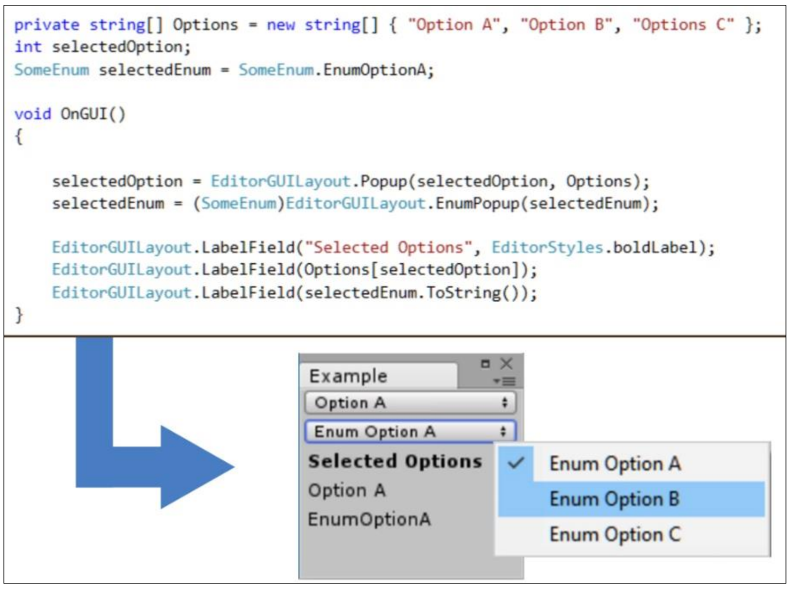
Example

Text : Hello Text

Figure 32. Input Field example

Another common control type is a popup menu, these controls allow the user to pick an option from a pop-up list (Figure 33). Unlike input controls, Popups do not directly change the value to what is currently selected in popup-box.

Instead, the popup-method returns the index of the selected option, for this reason, Popup-methods return either integers or enums. The text of the options come from either user-defined array or are automatically generated from enum-names. (Smith & Queiroz 2015, 516-518.)



```
private string[] Options = new string[] { "Option A", "Option B", "Options C" };
int selectedOption;
SomeEnum selectedEnum = SomeEnum.EnumOptionA;

void OnGUI()
{
    selectedOption = EditorGUILayout.Popup(selectedOption, Options);
    selectedEnum = (SomeEnum)EditorGUILayout.EnumPopup(selectedEnum);

    EditorGUILayout.LabelField("Selected Options", EditorStyles.boldLabel);
    EditorGUILayout.LabelField(Options[selectedOption]);
    EditorGUILayout.LabelField(selectedEnum.ToString());
}
```

Figure 33. Popup example

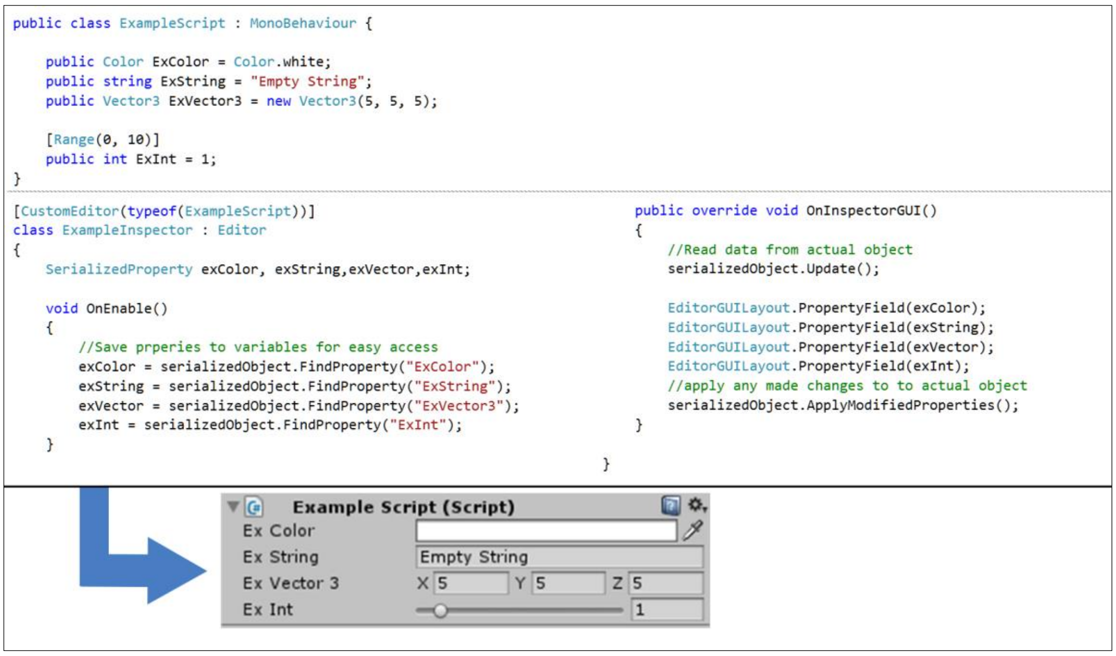
Popups are an easy way to give the user a clear way to choose between multiple choices. Many programs use similar tools in a Settings-window for the same purpose. Without a popup user would have to input the value manually, relying on his own memory to know the right index for the desired option. Giving a user easy way to choose between option makes tools easier to use and reduce the risk of mistakes.

6.2 Serialized Controls

As mentioned in chapter 3.4, there are two approaches to writing editor tools. The first one is modifying script variable directly with normal input controls mentioned above or by modifying the serialized representation of the object. UnityEditor-namespace includes several tools, that are used to modify serialized properties. The tools have a ton of inbuilt functionalities not available for normal editor tools. The most useful parts of these controls are inbuilt Undo and multi-edit -functionalities. The tools are generic and work with any serializable class. (Unity Technologies 2017a.)

6.2.1 PropertyField

PropertyField is a field that automatically determines the type of variable that it is given and generates an appropriate accessor control. As can be seen in the Inspector script in Figure 34 a function call is always the same for every variable regardless of the type. It is important to note, that developer has no direct control over what type of control is going to be shown, so the developer cannot specifically call IntField or IntSlider for example. To have a control over how the property is shown, one can use PropertyAttributes as in Figure 34, with ExInt-variable. (Tadres 2015, 93.)



```
public class ExampleScript : MonoBehaviour {

    public Color ExColor = Color.white;
    public string ExString = "Empty String";
    public Vector3 ExVector3 = new Vector3(5, 5, 5);

    [Range(0, 10)]
    public int ExInt = 1;
}
```

```
[CustomEditor(typeof(ExampleScript))]
class ExampleInspector : Editor
{
    SerializedProperty exColor, exString,exVector,exInt;

    void OnEnable()
    {
        //Save prperies to variables for easy access
        exColor = serializedObject.FindProperty("ExColor");
        exString = serializedObject.FindProperty("ExString");
        exVector = serializedObject.FindProperty("ExVector3");
        exInt = serializedObject.FindProperty("ExInt");
    }
```

```
public override void OnInspectorGUI()
{
    //Read data from actual object
    serializedObject.Update();

    EditorGUILayout.PropertyField(exColor);
    EditorGUILayout.PropertyField(exString);
    EditorGUILayout.PropertyField(exVector);
    EditorGUILayout.PropertyField(exInt);
    //apply any made changes to to actual object
    serializedObject.ApplyModifiedProperties();
}
}
```

Example Script (Script)

Ex Color	
Ex String	Empty String
Ex Vector 3	X 5 Y 5 Z 5
Ex Int	1

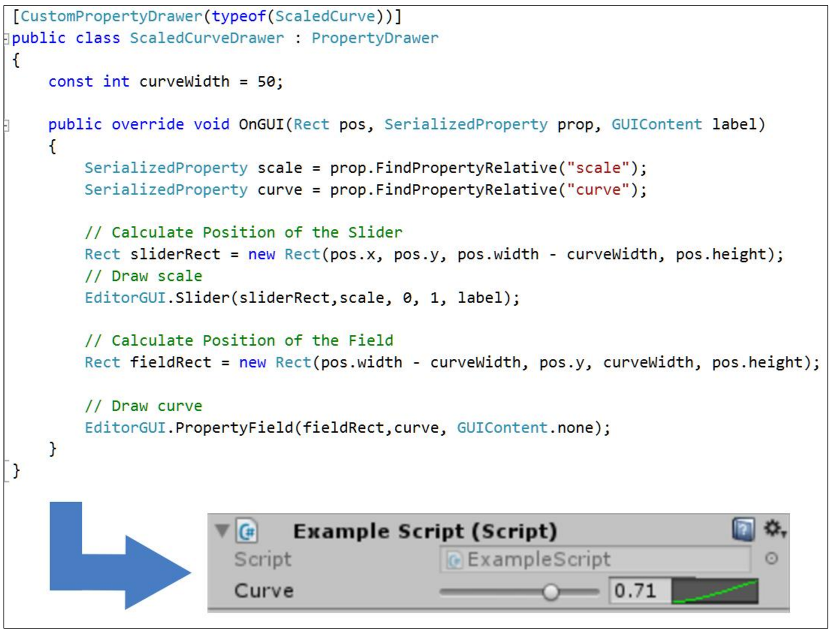
Figure 34. PropertyField Script

To use PropertyField one must use a SerializedProperty, which is a serialized representation of the variable that is being modified. In OnInspectorGUI-method there are two important methods, that are required to make Property-Fields work. First, there is Update-method of the SerializedObject, that holds the variable. This method refreshes the SerializedObject, making sure that all properties are in sync with the actual object. At the end, there is ApplyModifiedProperties-method. As the name suggests this method applies all changes made to the original object. (Unity Technologies 2017a.) This method also records which properties were changed and creates Undo-steps for the changes. (Tadres 2015, 104-105.)

6.2.2 PropertyDrawer

PropertyDrawers determines how properties are shown by a PropertyField. Like custom Inspectors, PropertyDrawers can change how an entire serializable class is drawn by overriding the default PropertyDrawer for the entire class. PropertyDrawers can also be called by attributes, thus changing only appearances of desired properties. As can be seen in the previous example in Figure 34, using Range-Attribute with the integer variable changed accessor control to a slider, which is a PropertyDrawer. (Tadres 2015, 104-105.)

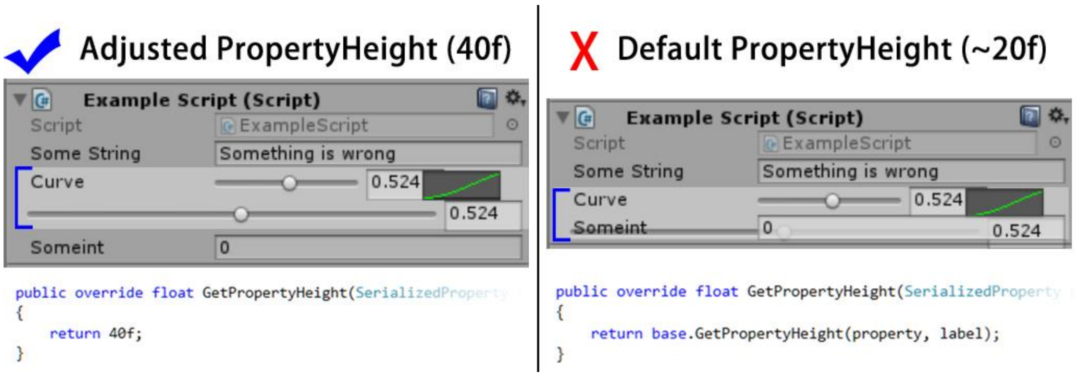
The developer can create his own custom PropertyDrawers. PropertyDrawers can easily be reused between multiple editors since they are automatically used by default Inspectors and PropertyFields. As mentioned above Property Drawers can be used in two ways: Overriding a whole class or with a single property with attributes. The example in Figure 35, overrides an entire class. After this example, all default Inspectors and PropertyFields will use the custom PropertyDrawer to display all instances of the ScaledCurve-class. (Unity Technologies 2012b.)



```
[CustomPropertyDrawer(typeof(ScaledCurve))]
public class ScaledCurveDrawer : PropertyDrawer
{
    const int curveWidth = 50;

    public override void OnGUI(Rect pos, SerializedProperty prop, GUIContent label)
    {
        SerializedProperty scale = prop.FindPropertyRelative("scale");
        SerializedProperty curve = prop.FindPropertyRelative("curve");

        // Calculate Position of the Slider
        Rect sliderRect = new Rect(pos.x, pos.y, pos.width - curveWidth, pos.height);
        // Draw scale
        EditorGUI.Slider(sliderRect,scale, 0, 1, label);

        // Calculate Position of the Field
        Rect fieldRect = new Rect(pos.width - curveWidth, pos.y, curveWidth, pos.height);

        // Draw curve
        EditorGUI.PropertyField(fieldRect,curve, GUIContent.none);
    }
}
```

Figure 35. Custom PropertyDrawer-class

The process of creating custom PropertyDrawers is quite like creating a custom Inspector. The biggest difference between the two is, that for the performance reason, PropertyDrawers cannot use layout-classes. Because of this limitation, the position of all elements must be manually calculated. This also applies to the height of the PropertyDrawer: all PropertyDrawers must return their height with GetPropertyHeight-method. If PropertyHeight is too small, the PropertyDrawer will overlap with others as demonstrated by Figure 36. (Unity Technologies 2015.)



```
public override float GetPropertyHeight(SerializedProperty
{
    return 40f;
}
```

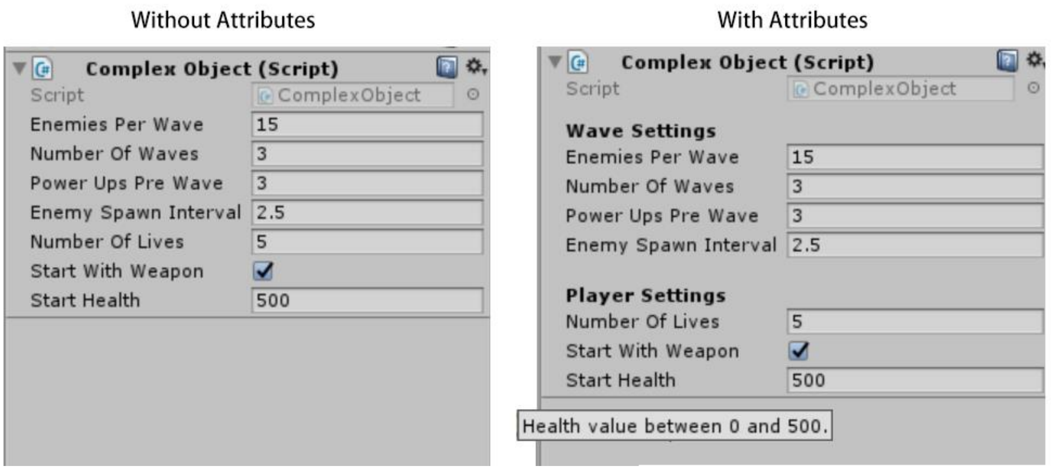
```
public override float GetPropertyHeight(SerializedProperty
{
    return base.GetPropertyHeight(property, label);
}
```

Figure 36. Property Height example

6.2.3 Built-In PropertyAttributes

PropertyAttributes are placed above declaration of a variable (property) in a script. PropertyAttributes are a simple way of doing implementing changes to the editor without rewriting entire Inspector. This chapter discusses built-in attributes, that exist in Unity.

Built-in attributes like Space, Header, Tooltip, and HideInInspector can improve readability of the editor. Space-attribute inserts spaces between elements, this is useful when trying to organize different element groups. By combining this with Header-attribute which inserts a text label above the element, one can improve the readability of the inspector by separating different elements based on their usage. Tooltip-attribute created a helpful tip-text that appears when a hover over the element, providing additional information about the element. This is especially helpful for team members who are not familiar with the object and what each variable is used for. In Figure 37, it is easy to see how much even minor changes to the inspector can improve the readability. (Tadres 2015, 97-99.)



Without Attributes

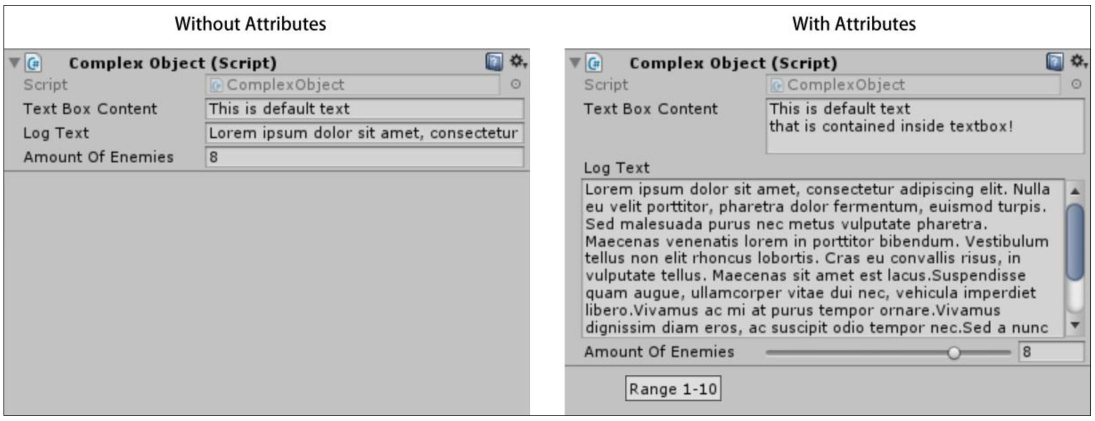
Complex Object (Script)
Script ComplexObject
Enemies Per Wave 15
Number Of Waves 3
Power Ups Pre Wave 3
Enemy Spawn Interval 2.5
Number Of Lives 5
Start With Weapon ✓
Start Health 500

With Attributes

Complex Object (Script)
Script ComplexObject
Wave Settings
Enemies Per Wave 15
Number Of Waves 3
Power Ups Pre Wave 3
Enemy Spawn Interval 2.5
Player Settings
Number Of Lives 5
Start With Weapon ✓
Start Health 500
Health value between 0 and 500.

Figure 37. Layout attributes

There are a few attributes that can actually change which built-in PropertyDrawers is used to draw the field. These built-in attributes are Multiline, TextArea and Range. These attributes are demonstrated in Figure 38. Multiline-attribute allows a user to write string-variables in a multiline textbox, which can contain more text than default input control as well as allows the user to insert line breaks with Enter-key. TextArea is like Multiline but can contain more text thanks to a scrollbar. These two attributes are used with fields that contain a lot of text. (Tadres 2015, 94-95.) The final relevant built-in attribute is the Range attribute, which turns a normal integer and float input control to a slider, with set minimum and maximum values. (Dickinson. 2015, 382.) This is useful if values must be kept within a certain range.



Without Attributes

Complex Object (Script)
Script ComplexObject
Text Box Content This is default text
Log Text Lorem ipsum dolor sit amet, consectetur
Amount Of Enemies 8

With Attributes

Complex Object (Script)
Script ComplexObject
Text Box Content This is default text
that is contained inside textbox!
Log Text
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla
eu velit porttitor, pharetra dolor fermentum, euismod turpis.
Sed malesuada purus nec metus vulputate pharetra.
Maecenas venenatis lorem in porttitor bibendum. Vestibulum
tellus non elit rhoncus lobortis. Cras eu convallis risus, in
vulputate tellus. Maecenas sit amet est lacus.Suspendisse
quam augue, ullamcorper vitae dui nec, vehicula imperdiet
libero.Vivamus ac mi at purus tempor ornare.Vivamus
dignissim diam eros, ac suscipit odio tempor nec.Sed a nunc
Amount Of Enemies 8
Range 1-10

Figure 38 Input attributes

Since implementing these built-in attributes is very easy, they are worth knowing because they can make default inspector easier to read and use. While additional functionality provided by attributes is still relatively simple, they can alter editor without the need to write a completely new custom editor for the object.

6.2.4 Custom PropertyAttributes

Like built-in attributes, custom PropertyAttributes are placed before a variable declaration. By themselves attributes do not do anything, they are simply used to inform the Editor which PropertyDrawer is going to be used to display the next property and store any arguments given to the attribute. Before the property is rendered, the Unity check if the property has an attribute (or an associated PropertyDrawer for the class) attached to it and automatically renders the appropriate PropertyDrawer. Otherwise, Unity uses default PropertyDrawers. The general flow how Unity determines which PropertyDrawer is used can be seen in Figure 39. (Unity Technologies 2012b.)

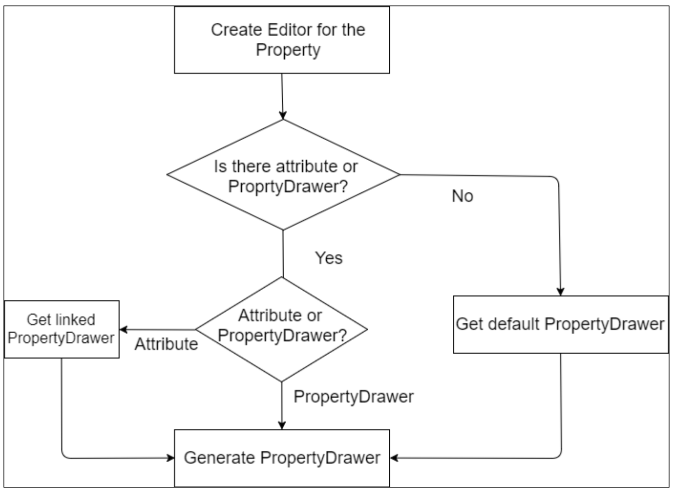
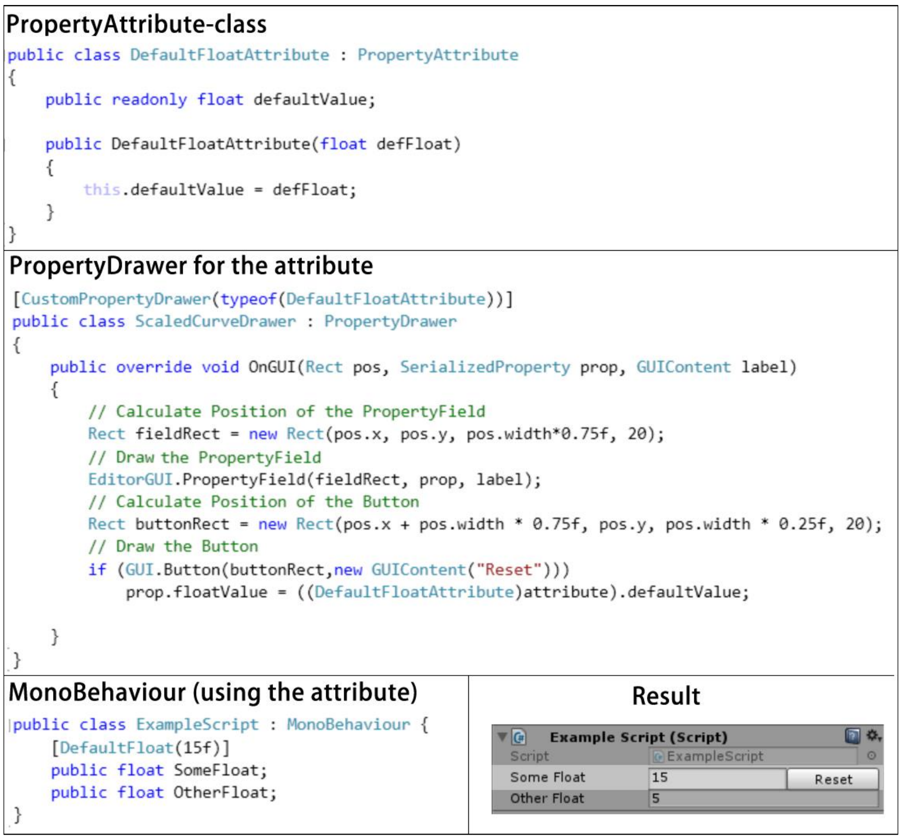

Create Editor for the Property
Is there attribute or ProprtyDrawer?
No
Yes
Get linked PropertyDrawer
Attribute
Attribute or PropertyDrawer?
Get default PropertyDrawer
PropertyDrawer
Generate PropertyDrawer

Figure 39. PropertyDrawer Flowchart

Programming-wise there is nothing too complicated about creating PropertyAttributes. The example below (Figure 40) shows the creation of a simple PropertyAttribute and PropertyDrawer, which can store and reset floats to default values. Custom Attribute-classes inherit from PropertyAttribute class. As can be seen in the example, if the attribute-class has a word “Attribute” in its name, the attribute must be declared without the word. So, in this case, the “DefaultFloatAttribute” is declared simply as “DefaultFloat”.



PropertyAttribute-class

```
public class DefaultFloatAttribute : PropertyAttribute
{
    public readonly float defaultValue;

    public DefaultFloatAttribute(float defFloat)
    {
        this.defaultValue = defFloat;
    }
}
```

PropertyDrawer for the attribute

```
[CustomPropertyDrawer(typeof(DefaultFloatAttribute))]
public class ScaledCurveDrawer : PropertyDrawer
{
    public override void OnGUI(Rect pos, SerializedProperty prop, GUIContent label)
    {
        // Calculate Position of the PropertyField
        Rect fieldRect = new Rect(pos.x, pos.y, pos.width*0.75f, 20);
        // Draw the PropertyField
        EditorGUI.PropertyField(fieldRect, prop, label);
        // Calculate Position of the Button
        Rect buttonRect = new Rect(pos.x + pos.width * 0.75f, pos.y, pos.width * 0.25f, 20);
        // Draw the Button
        if (GUI.Button(buttonRect,new GUIContent("Reset")))
            prop.floatValue = ((DefaultFloatAttribute)attribute).defaultValue;

    }
}
```

MonoBehaviour (using the attribute)

```
public class ExampleScript : MonoBehaviour {
    [DefaultFloat(15f)]
    public float SomeFloat;
    public float OtherFloat;
}
```

Result

```
Example Script (Script)
Script          ExampleScript
Some Float      15              Reset
Other Float     5
```

Figure 40. Custom PropertyAttribute

The process of creating PropertyDrawer for the attribute is same as creating custom Editors. A class is preceded by CustomPropertyDrawer-attribute, which links the PropertyDrawer to the PropertyAttribute. A reference to the PropertyAttribute is stored to the attribute-property of the PropertyDrawer-class, which gives PropertyDrawer access to the attribute and its properties.

6.3 Undo and ChangeCheck

As mentioned before, normal UI-controls do not have a built-in Undo-functionality. This is a very big issued since Undo-functionality is a universally expected function, that exists in almost any program. Undo is expected for a reason, as it reduces a risk of mistakes.

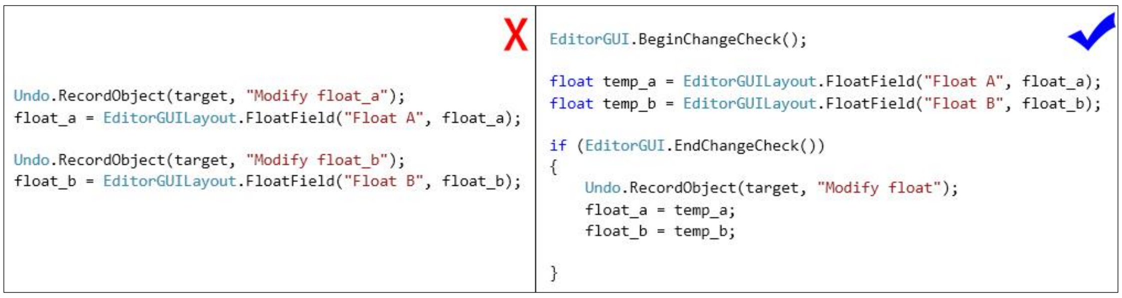
Implementing Undo-functionality is done by using Undo-class, which is part of UnityEditor-namespace. Undo-class takes a snapshot of UnityEngine.Object and save its state. This state is then saved to the Undo-stack and can be recovered by performing Undo. Undo-class can also keep track of object management actions like object creation and deletion. All different methods are explained in Table 2. (Unity Technologies 2017j.)

Table 2. Undo-methods

Method	Description
Add Component	Add Component to GameObject, with Undo
DestroyObjectImmediate	Destroys object, with Undo
RecordObject	Creates Undo a single object
RecordObjects	Create Undo for multiple objects. Same as RecordObject.
RegisterCompleteObjectUndo	Records complete state of the object, any changes made after the call will be ignored upon Undo.
RegisterCreatedObjectUndo	Create an Object, with the possibility to undo the creation.
RegisterFullObjectHierarchy-Undo	Records complete state of the hierarchy object, any changes made after the call will be ignored upon Undo. Similar to RegisterCompleteObjectUndo.
SetTransformParent	Change the parent of the object, with Undo

To create and Undo-step, one must record the state of the object before any changes are made. Placing Undo-method before changes are made creates a snapshot of the object before changes, which is then used to create the Undo-step. Most Undo-methods take 2 arguments, the target object that is being recorded and the name of the Undo-step that will be displayed in Edit-menu. (Unity Technologies 2017j.)

For performance reasons Undo-methods should only be called before the changes are made. The approach on the left script of Figure 41, works well for actions that only happen on when triggered, like button clicks. But calling Undo-methods for Input-fields, which are rendered once per frame, can lead to performance issues with bigger tools. To avoid this issue, one should take advantage of ChangeCheck-groups, which can be used to trigger Undo-methods only when specific controls are changed.



```
Undo.RecordObject(target, "Modify float_a");
float_a = EditorGUILayout.FloatField("Float A", float_a);

Undo.RecordObject(target, "Modify float_b");
float_b = EditorGUILayout.FloatField("Float B", float_b);
```

```
EditorGUI.BeginChangeCheck();

float temp_a = EditorGUILayout.FloatField("Float A", float_a);
float temp_b = EditorGUILayout.FloatField("Float B", float_b);

if (EditorGUI.EndChangeCheck())
{
    Undo.RecordObject(target, "Modify float");
    float_a = temp_a;
    float_b = temp_b;

}
```

Figure 41. Two ways of Undo-implementation

ChangeCheck-groups work like other element groups: all controls between begin and end are considered a part of the same group. If Unity detects that a control inside the group has been changed, the EndChangeCheck-method will return true. By changing the actual property value inside the if-block, it is possible to call Undo-method before the changing the actual value. This way Undo-methods are only called when necessary. As evident by Figure 41 while several properties can be grouped inside a single group, using ChangeCheck groups for Undo does complicate the script quite a bit. To avoid this one must either use property fields or abandon Undo-functionality altogether. (Unity Technologies 2017k.)

7 CASE

The purpose of the case is to demonstrate how IMGUI extensions can be beneficial to the game development process, how the tools work and what kind of solutions were used. For this purpose, a simple turn-based RPG-game was created, with several tools to be used to develop the game. While the game was only created as a proof of concept for editor tools, it is nevertheless developed as a real game. The idea is that the base of the game is fully functional and one could use existing tools to develop a full game.

This chapter discusses each feature of the game in conjunction with the tools that are used for the said feature, reasons why certain features were implemented and advantages provided by editor tools. This, of course, leaves out many gameplay-only features of the game such as pathfinding and AI, which are irrelevant to the topic of this thesis.

7.1 Game Introduction

Before further talking about the tools, it is appropriate to give a brief introduction to the game seen in Figure 42. The game, currently code-named Shogun Tactics, is a turn-based strategy game, similar to games like Fire Emblem and Final Fantasy Tactics. The game takes place in isometric-2D levels, with minimum 2 teams participating, one being a player-controlled team, the rest AI-controlled. To beat a level, the player must accomplish a variety of different objectives such as defeating all enemies or surviving a certain number of turns.

The combat is a standard turn-based combat, where a team takes a turn to move all its units and after that, the next team does the same. During a turn characters can move, attack, guard, and use items. A character can move and attack any tile within a certain range. Movement and attack ranges vary depending on character and weapon equipped.

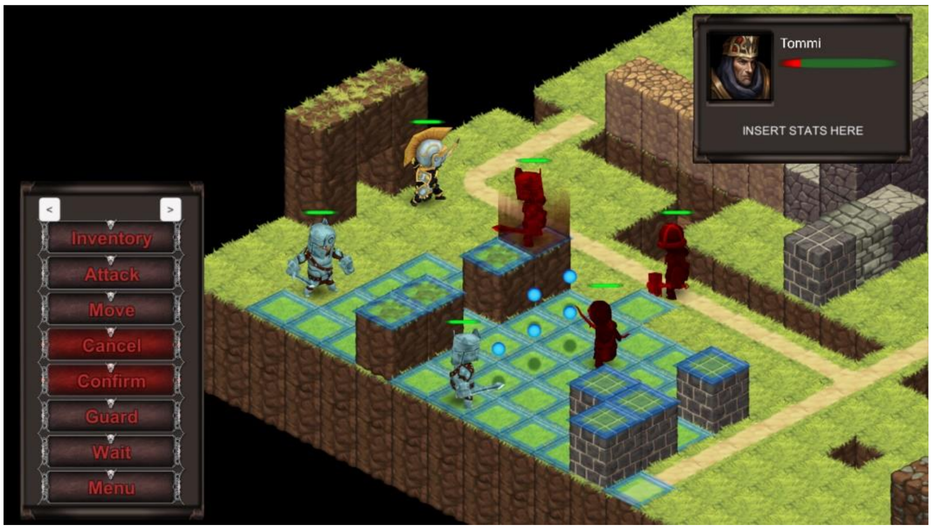


Figure 42. Shogun Tactics

The game is divided into missions, each with different objectives. Upon completing the objective player advances to the next mission, with shops and other in-between scenes between the missions. In shops, the player can buy and sell equipment and recovery-items. The game has been programmed in a way that technically anything can be loaded between missions, allowing flexibility when it comes to game structure. However, currently the game contains only two missions, with one shop in between.

7.2 Tools Overview

There are a ton of tools made for the project. The core idea for all tools was to make them as reusable as possible, meaning that they could easily be called from any other editor. This allows great flexibility with editor layouts and reduces the amount of programming that’s needed to be done when implementing new tools for the project.

Tools, in general, do not use SerializedObject, propertyFields or PropertyDrawers. While these are all extremely useful tools, due to lack of personal understanding and complications that generic nature of the tools caused, use of them was mostly avoided. Two biggest setbacks were automatic Undo-functionality and inability to use PropertyDrawers. While Undo-functionality was implemented manually for all tools, using PropertyDrawers was impractical for most cases.

Editor tools are used to modify two different kinds of objects: normal serializable-classes and objects derived from UnityEngine.Object. This means that some classes are modified with custom Inspectors and others with regular classes which utilize IMGUI-methods. This causes some issues when implementing Undo-functionalities, but otherwise, the tools are indistinguishable. Table 3 describes types of the most important classes.

Table 3. Editors and types

Class-name	Type
Missions	Object
Teams	Serialized
Characters	Serialized
Items	Object
Mission Conditions	Object
Mission Events	Serialized
Maps and GameTiles	Object

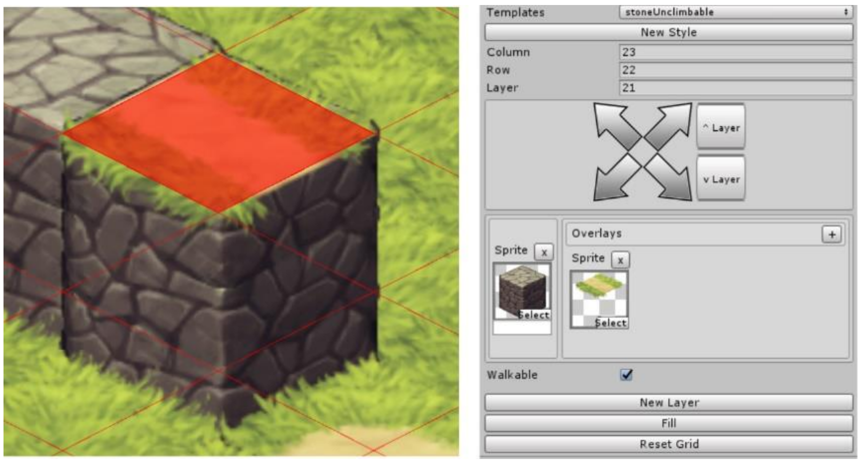
The reason for two different types of editors is the nature of UnityEngine.Objects, namely that these objects must exist in a file or as a gameObject on a scene. For many cases creating a separate file for every instance of the class was considered impractical. In some other cases, alternate methods were used for experimental purposes. The decision of which approach was used usually boiled down to several factors: How many instances of the object are going to be created, nature of the parent object and is polymorphism needed? The final point is exceptionally important since as discussed in 3.3 Unity’s serialization does not work well with polymorphism with not-UnityEngine.Object-derived classes. So, in most cases, if polymorphism was required, the class was made to derive from UnityEngine.Object and saved as files.

7.3 Maps and Game Tiles

The gameplay takes place in tile-based isometric 2D-maps. Each tile is a MonoBehaviour-script represented by a 2D-graphic. Each mission is associated with a single map, that is loaded along with the mission. A character can move around the map in horizontal and vertical direction or jump up and down different layers. It goes without saying, that building these maps manually would be a monotonous task: dragging and cloning each tile to its proper place, assigning each tile to a script, setting proper sorting order, just to name a few problems. It was clear early on, that creating a level creation-tool was one of the top priorities.

On technical level maps are MonoBehaviour scripts, located in a scene. This script, called GameGrid, contains a reference to all Game Tile-MonoBehaviours and is used to relay information of tiles to other scripts. One of the most important functions of GameGrid is to inform another script wherever specified coordinates have a tile or not. The maps work on singleton designs, meaning that only one map can exist at a time.

Map Editor is a custom Inspector for GameGrid-class. The editor is responsible for creating, modifying and deleting tiles. To be accurate, actual modifying of tiles is done with custom Inspector of selected GameTile-MonoBehaviour, which is generated inside the GameGrid-Editor. GameTile Editor supports Multi-Editing so that multiple tiles can be edited at the same time.



Templates stoneUnclimbable
Column 23
Row 22
Layer 21
Walkable ✓

Figure 43. Map-Editor

As can be seen in Figure 43, the UI for MapEditor is basic. The user can create tiles and layers with a button click. The editor is able to detect if currently selected tiles are empty and change its behavior accordingly. When there is no tile user has the ability to create a tile and when there is a tile user is allowed to modify it. Tiles are created with a graphic last selected by the user, removing the need for the user to assign graphics for every created tile.

MissionEditor uses SceneViewGUI-method to draw outlines for tiles in Scene View. This makes it easy to distinguish between tiles, which is extremely helpful with an isometric perspective and multiple layers. To make grid easily visible in every environment, the user can modify the color of the grid from Preference-window.

The user can navigate the grid with a keyboard, by clicking tiles or with pressing arrows in navigation UI, seen in Figure 44. The different approaches are a result of experimentation, as due to perspective and technical limitation no option seemed ideal for every situation. For example, clicking tiles that are behind of another tile is hard to do with a mouse, but easy to do with a keyboard.

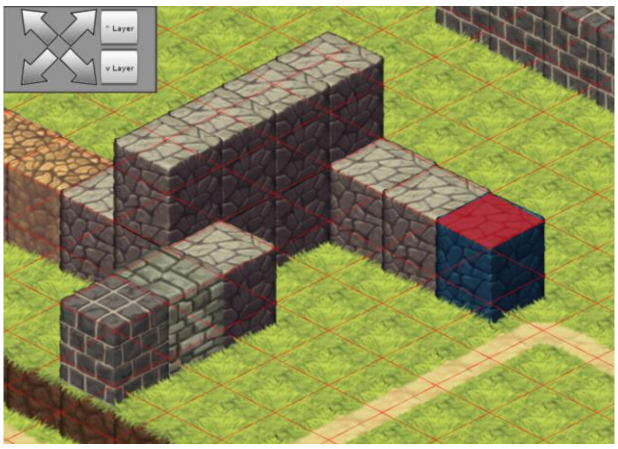


Figure 44. Map Grid

7.4 Missions

The game is separated to multiple missions, each with their own associated map (see Figure 45). To make each map reusable between different missions, missions and maps are created as separated entities. Missions contain information about teams, characters, events as well as winning and losing conditions.

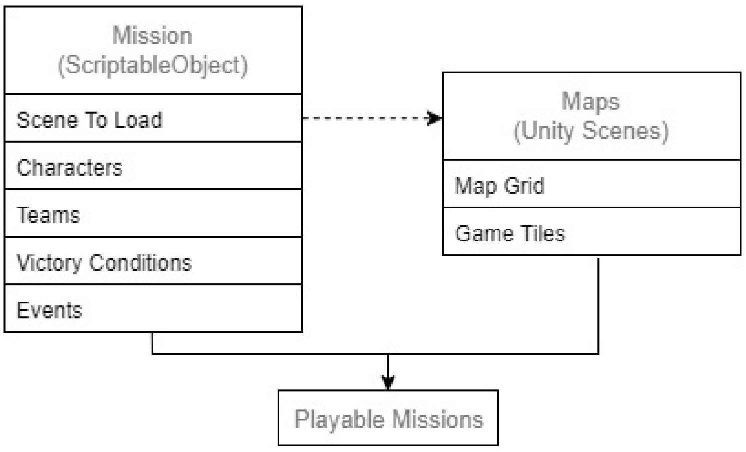
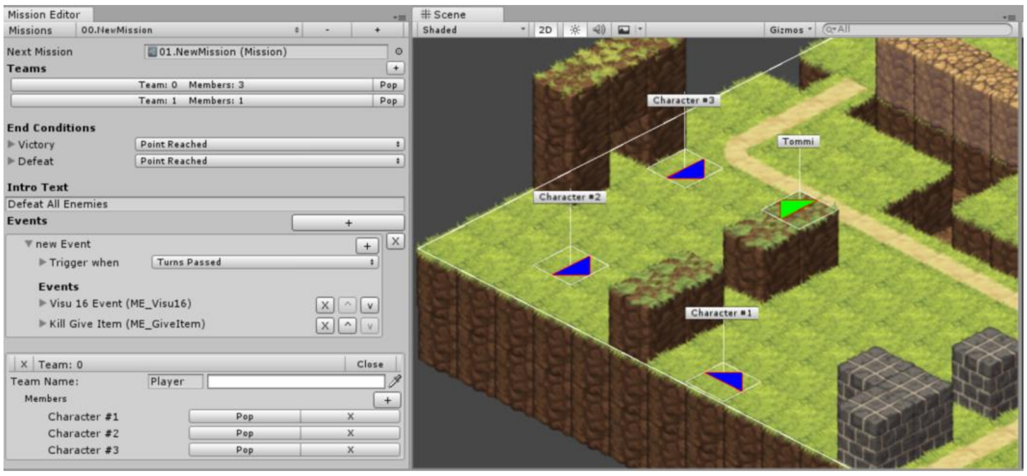


Figure 45. Mission structure

Missions are ScriptableObject-files stored on disk and Maps are stored in Unity's Scene-files This structure causes a bit of a problem since Missions and Maps do not exist in the same scene, making cross-referencing between the entities is not possible. Since Mission-ScriptableObjects do not exist on the scene it cannot refer Maps or GameTiles located on scenes. To get around this, Missions do not refer other tiles directly but instead, use coordinate-system. For example, editor scripts frequently ask GameGrid-script if a specific spot on the grid is empty. Using the coordinates instead of direct references gets around any cross-reference issues. However, without fail check, characters could spawn on top of empty tile or inside tile on top of spawn-tile. This is because coordinates themselves do not indicate if a tile is empty or not and do not respond to changes made to the map.

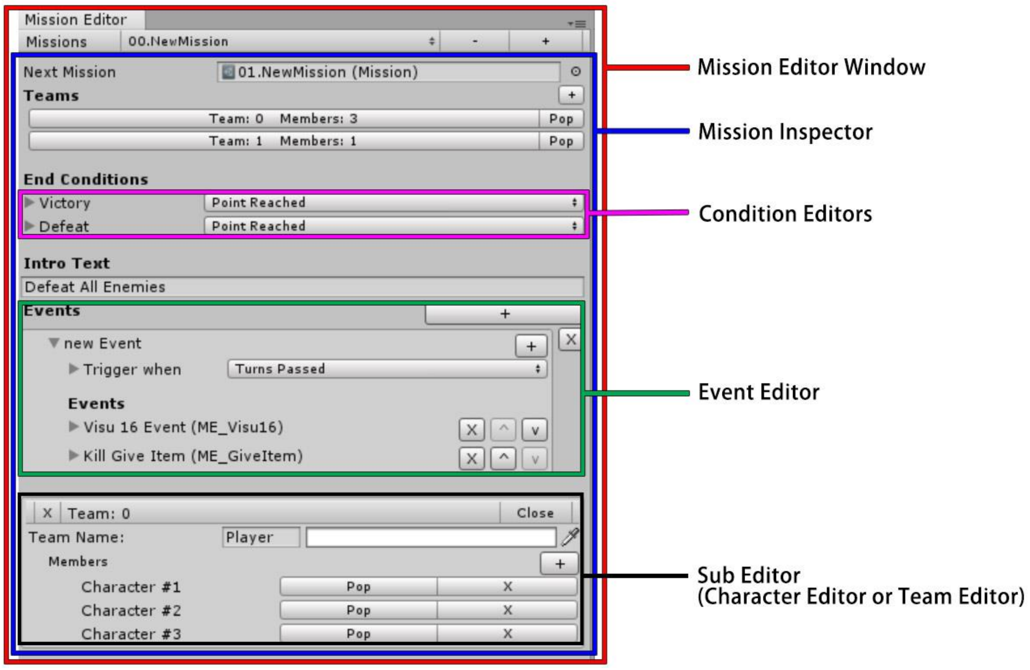
Missions Editor window (Figure 46) is the main hub for almost all other editors made for the game. The structure of the Editor window is illustrated in Figure 47 along with all its sub-editors. Mission Editor is an Editor window, which can load, create and delete Mission-ScriptableObjects and load their related scenes with a button click. After the Mission-object is loaded Unity calls the custom Inspector of the object, which is the actual main editor. Using Editor-Window as a wrapper for all Mission Inspectors allow easy switching between missions as well as free the normal Inspector Window for Map-editor.



Missions: 00.NewMission
Next Mission: 01.NewMission (Mission)
Teams
Team: 0 Members: 3 — Pop
Team: 1 Members: 1 — Pop
End Conditions
Victory: Point Reached
Defeat: Point Reached
Intro Text
Defeat All Enemies
Events
new Event
Trigger when: Turns Passed
Events
Visu 16 Event (ME_Visu16)
Kill Give Item (ME_GiveItem)
Team: 0 — Close
Team Name: Player
Members
Character #1 — Pop — X
Character #2 — Pop — X
Character #3 — Pop — X

Figure 46. Mission Editor

Mission Editor also draws characters to the Scene View. As can be seen in Figure 46, the display for the characters is a simple triangle on the ground, with a button above it. Pressing the button opens the character editor for that character and allows the user to drag a character around, changing the spawn point. The colored triangle tells three things: characters′ spawn position, the direction the character is going to face when spawned and characters′ team. The simplified appearance makes Scene View easy to read, without having a ton of characters filling the screen.



Mission Editor
Missions: 00.NewMission
Next Mission: 01.NewMission (Mission)
Teams
Team: 0 Members: 3 — Pop
Team: 1 Members: 1 — Pop
End Conditions
Victory: Point Reached
Defeat: Point Reached
Intro Text
Defeat All Enemies
Events
new Event
Trigger when: Turns Passed
Events
Visu 16 Event (ME_Visu16)
Kill Give Item (ME_GiveItem)
Team: 0 — Close
Team Name: Player
Members
Character #1 — Pop — X
Character #2 — Pop — X
Character #3 — Pop — X

Figure 47. MissionEditor structure

Going from top-down with one editor at a time. First, there is a Mission Inspector which only contains several controls of its own: Next Mission object-field, Team Selection, and Intro Text-text field. The user can determine which

Loadable-ScriptableObject gets loaded after the player beats the mission, this means other Missions or Shops. Finally, the Intro-text field determines the text that pops up when Mission starts, this is used to tell the player his current objective.

The first child-editors are two MissionCondition-Editors, which are used to create and modify MissionCondition-objects for defeat and victory. MissionConditions are explained more detail in 7.5. Next, there is Event Editor, which is used to edit events, which can be triggered by Mission Conditions. Events can, for example, kill a certain character when a certain amount of turns have passed or to give a character an item when another character is killed. Finally, there is a sub-editor-slot. Sub-editor slot is used by Team Editor (like in Figure 47) or Character Editor depending on what is being edited. Sub-editor slot is used to save space. The user can also open sub-editors as a separate window by pressing the pop-button.

7.5 Mission Conditions and Events

To make the game more varied it is necessary to implement different types of missions. The different mission can have different winning and losing conditions. For example, the most common victory condition is when all enemies have been defeated. But some other mission may require the player to survive x-number of turns or protect certain character to the end. It soon became apparent, that similar system would be useful for triggering gameplay-events. For this purpose, the MissionCondition-class was created. MissionCondition is an abstract base class for all different mission conditions. There are several different types of mission conditions, each with its own logic how the condition is determined. Currently implemented conditions are explained in Table 4

Table 4. Condition types

Condition-name	Description
UnitsDead	Return true when certain characters are defeated
UnitsDeadAmount	Return true when certain number of enemies are defeated
TeamsDead	Return true when entire team is defeated
PointReached	Return true when a member of a team has reached any of determined points.
TurnsPassed	Return true when certain number of turn have passed

Mission conditions are checked when any character performs an action or when a turn ends. By checking conditions as often as possible, it is ensured that response to all condition types is instant. What happens when a condition is true depends entirely on the code, that uses it. When MissionCondition for victory conditions is true the game ends and player advances to the next mission, but if the same condition was set to defeat condition the mission restarts. MissionConditions are also used for spawning characters after the first turn.

On the editor side, all mission conditions have their own custom inspector, which can be drawn inside other editors. As demonstrated in Figure 48, custom Inspectors follow the same class structure as MissionConditions, with MissionConditionEditor as a base class. Having identical class structure is crucial for polymorphic editor tools because now any custom editors for MissionCondition-object can be cast to a base MissionConditionEditor-class. For example, the script in Figure 49 is now able to determine if the custom editor is MissionConditionEditor and call a class-specific initialization method. If a mission condition has no custom editor or doesn't inherit from MissionConditionEditor-class, it is simply drawn without initialization.

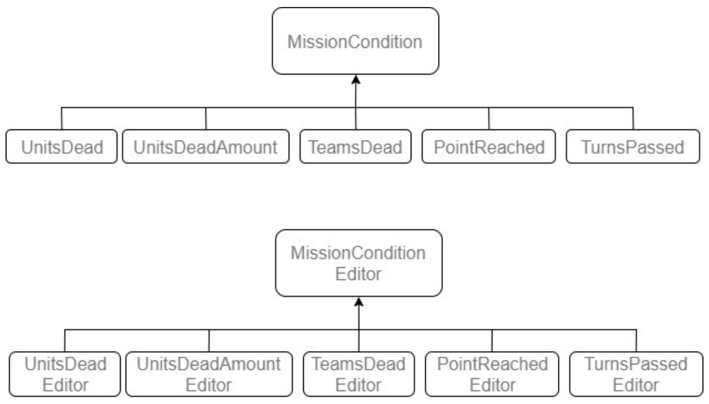
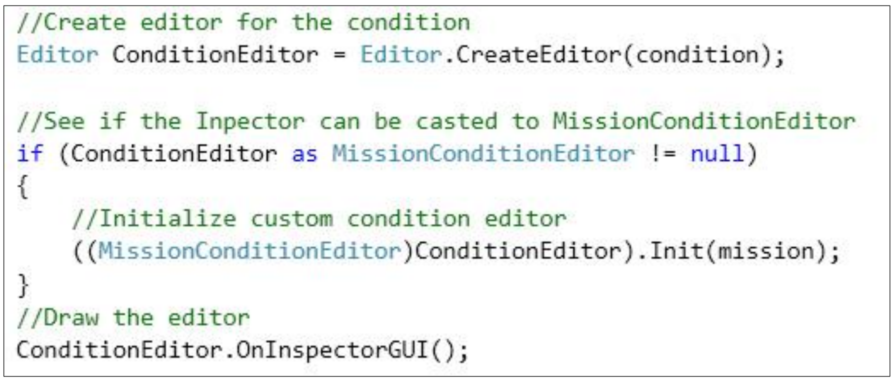


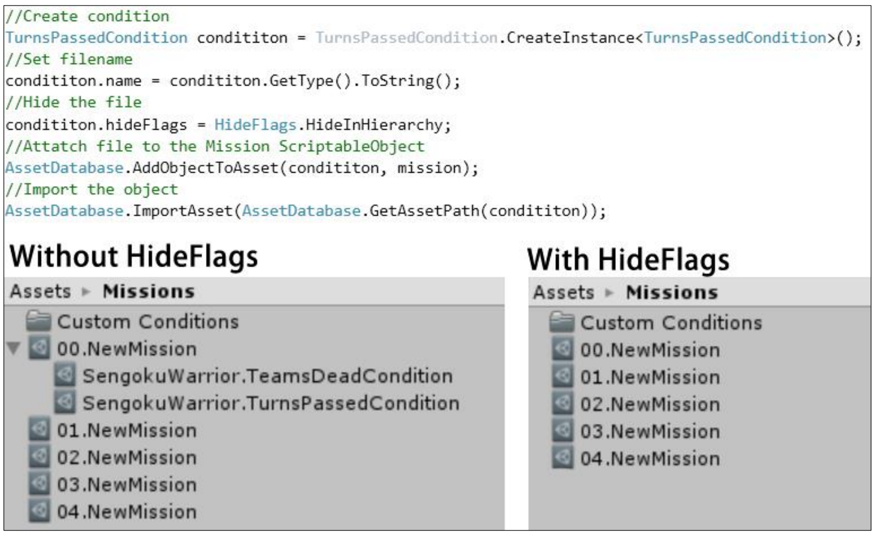
Figure 48. Mission Condition class structure



```
//Create editor for the condition
Editor ConditionEditor = Editor.CreateEditor(condition);

//See if the Inpector can be casted to MissionConditionEditor
if (ConditionEditor as MissionConditionEditor != null)
{
    //Initialize custom condition editor
    ((MissionConditionEditor)ConditionEditor).Init(mission);
}
//Draw the editor
ConditionEditor.OnInspectorGUI();
```

Figure 49. Polymorphism with editors

Implementing Mission Condition to editor provided a couple challenges. Because Mission Conditions are ScriptableObjects, they must be saved to file. To avoid a cluster of files in the project, the files are stored inside the Mission-ScriptableObject itself, by using AssetsDatabase.AddObjectToAsset-method. This method can attach any UnityEngine.Object to another object. As done in Figure 50, by using HideFlags-class it is possible to completely hide these attached objects from the view. The advantage of this approach is that, when Mission-file is deleted, all its MissionConditions are removed along with it, so there is no need to remove each file individually.



```
//Create condition
TurnsPassedCondition condititon = TurnsPassedCondition.CreateInstance<TurnsPassedCondition>();
//Set filename
condititon.name = condititon.GetType().ToString();
//Hide the file
condititon.hideFlags = HideFlags.HideInHierarchy;
//Attatch file to the Mission ScriptableObject
AssetDatabase.AddObjectToAsset(condititon, mission);
//Import the object
AssetDatabase.ImportAsset(AssetDatabase.GetAssetPath(condititon));
```

Figure 50. HideFlags

To make MissionConditions easily usable in different contexts, a new control called ConditionSelector (Figure 51) was created. This control handles creating and deleting mission conditions as well as generating the custom Inspectors for conditions. The actual control contains three elements: foldout, label, and popup. The control can be used anywhere as long as it can refer to a mission, which many conditions require anyway. For example, TeamsDead-condition inevitably needs information about how many teams the mission contains.

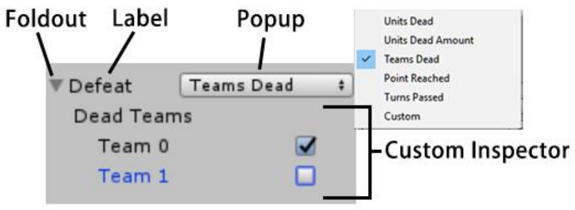


Figure 51. ConditionSelector-control

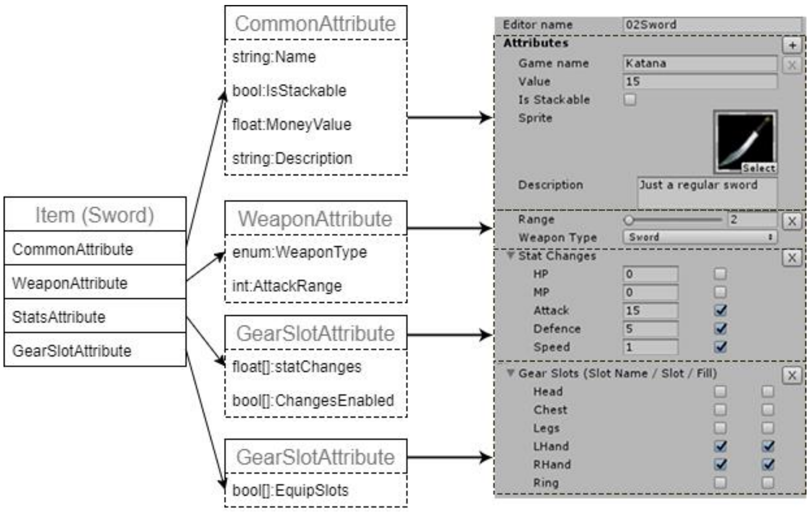
One of the main use of Mission conditions is Mission Events. Mission Events can trigger certain behavior when conditions determined by MissionCondition are fulfilled. Currently, there are only few event types: Visu16-events, GiveItem-events, and KillCharacter-events. Visu16-events use external Visu16-plugin to display conversation-cutscenes. GiveItem events add a certain item to characters inventory and KillCharacter-event kills certain characters.

7.6 Items

Items are built with attributes. This means there is only one Item-class and items are defined entirely by attributes that they hold. Attributes are ScriptableObjects attached to an Item-ScriptableObject, similar to how MissionConditions are attached to a Mission (See 7.5). ItemAttributes contain data and behavior that affect how item behaves in the game. This system is very flexible, instead of having separate classes for every item type, having each item defined by their attributes makes it easy to create combinations of different kind of behaviors. Each item can have any of attributes described in Table 5, with example item shown in Figure 52.

Table 5. ItemAttribute types

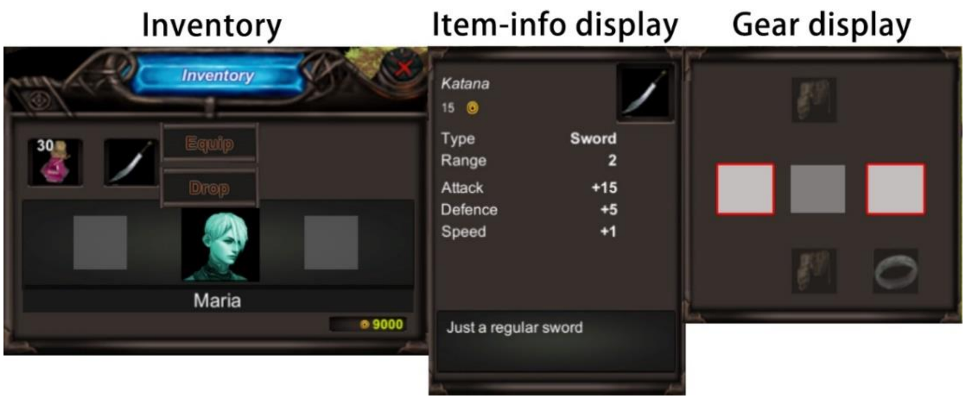
Name	Description
CommonAttribute	Contains base information common to all item such as name, value, and description. Must be present in all items.
WeaponAttribute	Weapon range and Weapon-type.
GearSlot-attribute	Determines slots that can the item be equipped to.
StatsAttribute	Determines changes item makes to Characters stats when equipped.
Use_RecoverAttribute	Allows the item to be used in Inventory, to heal Character.



```
Editor name        02Sword
Attributes
  Game name        Katana
  Value            15
  Is Stackable     ☐
  Sprite           Select
  Description      Just a regular sword
  Range            2
  Weapon Type      Sword
▼ Stat Changes
    HP             0      ☐
    MP             0      ☐
    Attack         15     ✓
    Defence        5      ✓
    Speed          1      ✓
▼ Gear Slots (Slot Name / Slot / Fill)
    Head                  ☐    ☐
    Chest                 ☐    ☐
    Legs                  ☐    ☐
    LHand                 ✓    ✓
    RHand                 ✓    ✓
    Ring                  ☐    ☐
```

Figure 52. Example Item

When items are loaded in a script, the script can check if the item has specific attributes attached to it and act accordingly. The most prominent example for this is characters inventory. In the inventory, the player can equip and unequip items. If the player clicks an item in inventory a popup-menu shows up. The content of this menu depends on the attributes of the item. If the item has gearSlot-attribute, the menu shows the equip-button. If the item has Use_Recovery-attribute, the player can choose to use the item. The inventory also displays player information about the item. Just like the popup menu, how this view is constructed depends on the attributes. The Figure 53 shows how the sword created in Figure 52 will show up in the game.



```
Katana
15
Type       Sword
Range          2
Attack       +15
Defence       +5
Speed         +1

Just a regular sword
```

Figure 53. In-game item in Inventory

Items are stored in ItemDatabase-ScriptableObject. ItemDatabase has a simple custom Inspector, which generates individual item-editors for all stored items. By using ItemDatabases editor it is easy to modify and create new items. When Item-editor is created it loops through every ItemAttribute-ScriptableObject and creates the associated Inspectors. Attributes are created and destroyed using Item-Editors.

7.7 Characters

Characters in the game are built of two components: Character-data and visual GameObject. GameObjects are only created in-game, based on Character-data. In the editor user never sees the character visually, instead, they are represented by arrows and indicators. Character data is what one would expect, it contains information like character name, stats, and inventory.

There are two types of characters: Unique and not-Unique character. Unique characters are essentially player characters, characters whose data need to be saved and loaded between missions. As demonstrated by Figure 54, how data is loaded varies a bit depending on the character type. The reason for differentiating the two-character type is twofold: to tell the game which characters are saved and to tell the editor what properties of characters are editable. Because unique-characters are out of developers direct control after first in-game usage, it makes no sense to allow users to modify unique characters on a mission by mission basis.

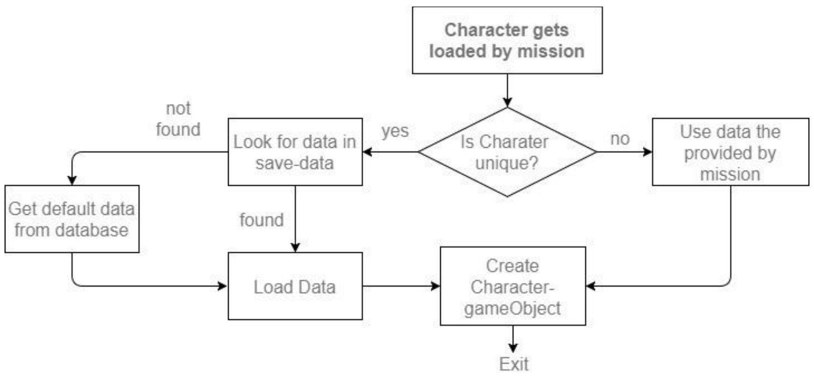
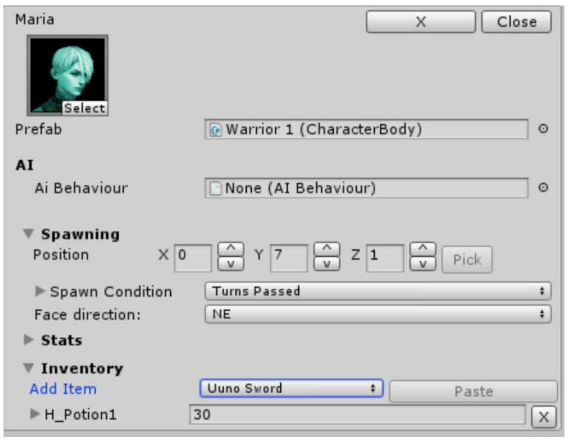

Character gets loaded by mission
Is Charater unique?
yes
no
Look for data in save-data
Use data the provided by mission
not found
found
Get default data from database
Load Data
Create Character-gameObject
Exit

Figure 54. Loading Character-data during gameplay

In editor-side, the data for unique characters are stored in ScriptableObject-database, called CharacterDatabase, along with template-data of reusable not-unique characters. When the game first uses a unique character, the game loads the information of the character from this database. Non-unique characters get their data directly from the mission.


Maria

X Close

Select

Prefab: Warrior 1 (CharacterBody)

AI

Ai Behaviour: None (AI Behaviour)

Spawning

Position: X 0, Y 7, Z 1 — Pick

Spawn Condition: Turns Passed

Face direction: NE

Stats

Inventory

Add Item: Uuno Sword — Paste

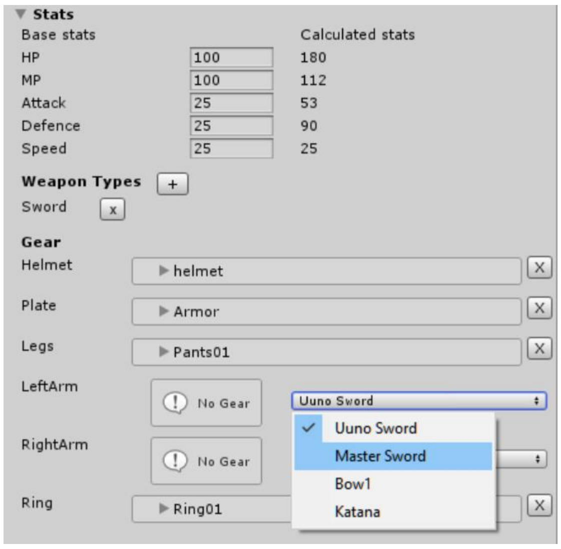
H_Potion1: 30

Figure 55. Character Editor

Character Editor (Figure 55) is used by Mission Editor and Character Database to modify properties of Character data. The Character editor has several child editors for stats, spawn conditions (Mission Conditions) and inventory. These two editors are only used by Character editor but are nevertheless treated as separate editor in code. This is to allow any possible future reusing of editors elsewhere.

7.7.1 Stats

Stats represent a strength of a character based on numeric values. Stats determine how many life points the character has, how much damage he does and what kind of weapons he can equip. Stats are also affected by items character has equipped, so for example, chest plates improve character defense and weapons improve attack. Stats and equipped items can be edited with the Stats-editor (Figure 56).


Stats

Base stats		Calculated stats
HP	100	180
MP	100	112
Attack	25	53
Defence	25	90
Speed	25	25

Weapon Types +

Sword x

Gear

Helmet	helmet
Plate	Armor
Legs	Pants01
LeftArm	No Gear — Uuno Sword
RightArm	No Gear
Ring	Ring01

Figure 56. Stats Editor

Stats editor has several features, that demonstrate how useful editor extensions can be. Stats-editor has several input-fields for each stat, but next to the there is also non-interactable label field, which previews the actual final stat, which takes account items that character has equipped. This feature allows designers to get a more accurate idea of how strong the character is, without having to do all manual calculation themselves, saving a lot of time.

Choosing items from a popups list is easy. Since each popup only displays items valid to current equipment-slot, there is no possibility of equipping a wrong type of item. Without editor tools, this process would be much more tedious. An Item-object would have to be manually dragged to the editor or selected from a list containing every single item. To make this process work without editor tools all items would have to be well organized in the project and have a very informative naming conventions. Also without tools, there would be no way to check if the item can even be equipped to a specific equipment slot.

8 CONCLUSIONS

The purpose of this thesis was to illustrate how IMGUI extensions work and how they can be implemented. Overall thesis covered all crucial elements of an extensions creating process, with exception of potential releasing of extensions in Asset Store. Since this topic was a fairly obscure it was important to go over the basics and then display how editor extensions can revolutionize how games can be developed. While creating editor extensions is strongly encouraged, they can take time to create and sometimes an cause new problems. However, as demonstrated by many examples in the thesis, editor extensions can both speed up the game development process and allow even non-programmers to take part in the technical development of the game.

If one follows Unity’s development, developers are going to keep adding new functionalities to the editor. According to Unity roadmap, there are plans for IMGUI debugger tools, Visual scripting and a whole new type of editor tools called UIElements. However, due to lack of reliable sources and the fact that they are features under development, they were left out of the thesis. (Unity Technologies 2017l.)

Building a game with editor tools was a major undertaking. While it is hard to estimate exactly how long each part of the project takes, overall developing of the game took most likely half of the development time and tools took the other half. While that sounds like a lot, it is important to keep in mind, that now with all tools in place, one could easily develop the full game with these tools. If the game was developed further, the time saved with editor tools would become greater, longer the development process is.

The biggest challenges, as far as editor extensions are concerned, came from advanced editor issues, such as designing data-structure ideal for both game and editor, figuring different ways to reuse editors and working with serialization. While there are a couple of issues, that could need further development, things learned from this project were enormous. Thesis provided a way to deepen personal knowledge of both editor-scripting and game programming in general.

REFERENCES

Blow J. 2004. Game Development: Harder Than You Think. Available at: http://queue.acm.org/issuedetail.cfm?issue=971564. [Accessed: 30.9.2017].

Cogut V. 2015. Unity 5 for Android Essentials.
Birmingham: Packt Publishing.

CG Channel Inc. 2016. Unity Technologies to make Anima2D available for free. Available at: http://www.cgchannel.com/2016/12/unity-technologies-to-make-anima2d-available-for-free/. [Accessed 20.11.2017]

Dickinson C. 2015. Unity 5 Game Optimization.
Birmingham: Packt Publishing.

Doran J. 2014. Unity Game Development Blueprints. Birmingham:
Packt Publishing.

Enterbrain Inc. 2017. RPGMaker VX Features. Available at: http://www.rpgmakerweb.com/products/programs/rpg-maker-vx-ace [Accessed 30.9.2017].

Hutong Games. 2017. PlayMaker-website. Available at: http://www.hutong-games.com/. [Accessed 20.11.2017].

Kogent Learning Solutions. 2009. Java 6 Programming Black Book.
New Delhi: Dreamtech Press.

Mandarina Games. 2015. Anima2D Features. Available at: https://anima2d.com/features/. [Accessed 20.11.2017].

Meier R. 2014 Custom editors in unity3d - part 6: serializedobject, serializedproperty, propertydrawer. Available at: http://www.ryan-meier.com/blog/?p=67. [Accessed 30.9.2017].

Miles J. 2016. Unity 3D and PlayMaker essentials: game development from concept to publishing. Boca Raton: CRC Press.

Sapio F. 2017. Getting Started with Unity 5.x 2D Game Development. Birmingham: Packt Publishing.

Simon J. 2015. Unity 3D UI Essentials. Birmingham: Packt Publishing.

Smith M. & Queiroz C. 2015. Unity 5.x Cookbook.
Birmingham: Packt Publishing.

Pierce G. 2012. Unity iOS Game Development.
Birmingham: Packt Publishing.

Tadres A. 2015. Extending Unity with Editor Scripting.
Birmingham: Packt Publishing.

The Knights of Unity. 2016. Asset Review – Editor Console Pro. http://blog.theknightsofunity.com/asset-review-editor-console-pro/. [Accessed 20.11.2017].

Thorn A. 2015. Mastering Unity Scripting. Birmingham: Packt Publishing.

Unity Technologies. 2012a. Unity Serialization. Available at: https://blogs.unity3d.com/2012/10/25/unity-serialization/. [Accessed 30.9.2017].

Unity Technologies. 2012b. Unity Serialization. Available at: https://blogs.unity3d.com/2012/09/07/property-drawers-in-unity-4/. [Accessed 30.9.2017].

Unity Technologies. 2014. Serialization in Unity. Available at: https://blogs.unity3d.com/2014/06/24/serialization-in-unity/. [Accessed 30.9.2017].

Unity Technologies. 2015. Going deep with IMGUI and Editor Customization. Available at: https://blogs.unity3d.com/2015/12/22/going-deep-with-imgui-and-editor-customization/. [Accessed 30.9.2017].

Unity Technologies. 2017a. Unity Scripting Reference. Available at: https://docs.unity3d.com/ScriptReference/Editor.html [Accessed 30.9.2017].

Unity Technologies. 2017b. Unity Scripting Reference. Available at: https://docs.unity3d.com/ScriptReference/SerializedProperty-property-Type.html. [Accessed 30.9.2017].

Unity Technologies. 2017c. Unity Scripting Reference. Available at: https://docs.unity3d.com/ScriptReference/HelpURLAttribute.html. [Accessed 30.9.2017].

Unity Technologies. 2017d Unity Scripting Reference. Available at: https://docs.unity3d.com/ScriptReference/PopupWindow.html. [Accessed 30.9.2017].

Unity Technologies. 2017e. Unity Scripting Reference. Available at: https://docs.unity3d.com/ScriptReference/GenericMenu.html. [Accessed 30.9.2017].

Unity Technologies. 2017e. Unity Scripting Reference. Available at: https://docs.unity3d.com/ScriptReference/ScriptableWizard.html. [Accessed 30.9.2017].

Unity Technologies. 2017f. Unity Scripting Reference. Available at: https://docs.unity3d.com/ScriptReference/PreferenceItem.html. [Accessed 30.9.2017].

Unity Technologies. 2017g. Unity Scripting Reference. Available at: https://docs.unity3d.com/ScriptReference/SessionState.html. [Accessed 30.9.2017].

Unity Technologies. 2017h. Unity Scripting Reference. Available at: https://docs.unity3d.com/ScriptReference/EditorWindow-position.html [Accessed 30.9.2017].

Unity Technologies. 2017i. Unity Scripting Reference. Available at: https://docs.unity3d.com/ScriptReference/EditorGUILayout.html [Accessed 30.9.2017].

Unity Technologies. 2017j. Unity Scripting Reference. Available at: https://docs.unity3d.com/ScriptReference/Undo.html. [Accessed 30.9.2017].

Unity Technologies. 2017k. Unity Scripting Reference. Available at: https://docs.unity3d.com/ScriptReference/EditorGUI.BeginChangeCheck.html. [Accessed 30.9.2017].

Unity Technologies. 2017l. Unity Scripting Reference. Available at: https://unity3d.com/unity/roadmap. [Accessed 1.12.2017].

Vodnik S. 2015. HTML5 and CSS3, Illustrated Complete. Boston: Cengage Learning.

Wagner B. 2014. Effective C# (Covers C# 4.0): 50 Specific Ways to Improve Your C#. London: Pearson Education.

LIST OF FIGURES

LIST OF TABLES