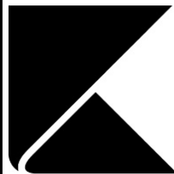
KARELIA UNIVERSITY OF APPLIED SCIENCES
Degree Programme in Business Information Technology

Hamza Kupiainen

EXTENDING UNITY GAME ENGINE THROUGH EDITOR SCRIPTING

Thesis
October 2018



Karelia UNIVERSITY OF APPLIED SCIENCES

THESIS
October 2018
Degree Programme In Business Information Technology

Tikkarinne 9
80220 JOENSUU
(013) 260 600

Author

Hamza Kupiainen

Title

Extending the Unity Game Engine Through Editor Scripting

Abstract

The aim of the thesis was to find out how a developer can use editor scripting to expand the Unity game engine. The thesis focuses solely on expanding the Unity game engine in the practical part of the thesis. Editor extensions can be beneficial to game development in many ways, for example, they can speed up the development. The thesis explores what kind of advantages and disadvantages editor programming has in general. The thesis also compares Unity to the popular game engine: The Unreal Engine.

The thesis is divided into two parts: theoretical and practical. The theoretical part of the thesis explores different ways how the developer can expand the Unity game engine using the editor scripting at the basic level. In the practical part of the thesis, a full-fledged game project supported by several editor extensions was created to support the aims of the thesis.

The thesis covered major topics of editor scripting in the theoretical part of the thesis. The editor scripting is such a big topic and because of that, some parts were left out as result. In the practical part of the thesis, space themed shoot 'em up game was created supported by several extensions. More information about the game project and extensions can be found at the end of the thesis.

Language English	Pages 33

Keywords

Unity3d, game development, editor programming, plugins, extensions

OPINNÄYTETYÖ
Lokakuu 2018
Tietojenkäsittelyn koulutusohjelma

Tikkarinne 9
80220 JOENSUU
(013) 260 600

Tekijä

Hamza Kupiainen

Nimike
Extending the Unity Game Engine Through Editor Programming

Tiivistelmä

Opinnäytetyön tarkoitus oli tutkia kuinka kehittäjä pystyisi laajentamaan Unity pelimoottoria editori skriptauksen avulla. Editori laajennukset voivat olla hyödyllisiä monella eri tavalla peliprojektille esimerkiksi niiden avulla pystyy nopeuttamaan projektin kehitystä. Opinnäytetyö keskittyi vain Unityn laajentamiseen käytännön osuudessa. Opinnäytetyö myös tutkii mitä erilaisia hyötyjä ja haittoja editori ohjelmoinnilla on yleisesti ottaen. Opinnäytetyö myös vertailee Unityä suosittuun pelimoottoriin: Unreal Engineen.

Opinnäytetyö jaettiin kahteen eri osuuteen: käytäntöön ja teoriaan. Teoreettisessa osuudessa opinnäytetyö tutkii millä eri tavoilla kehittäjä pystyy laajentamaan Unity pelimoottoria editori skriptauksen avulla. Opinnäytetyön käytännön osuudessa luotiin peliprojekti ja liitännäisiä, joiden tavoitteena oli tukea opinnäytetyön tutkimuskysymyksiä.

Opinnäytetyö käsitteli suurimman osan aiheista, joiden avulla kehittäjä pystyisi laajentamaan Unity pelimoottoria. Opinnäytetyöstä jätettiin osa aiheista ulkopuolelle, koska laajentaminen on liian iso kokonaisuus käsiteltäväksi tässä opinnäytetyössä. Opinnäytetyön käytännön osuudessa luotiin avaruusteemainen ammuntapeli ja erinäinen määrä laajennoksia. Tietoa projektista ja laajennoksista löytyy opinnäytetyön lopusta.

Kieli englanti	Sivuja 33

Asiasanat

Unity, pelikehitys, editori ohjelmointi, liitännäiset

Contents

1 INTRODUCTION 1
2 TOOLS 2
2.1 Unity 2
2.2 Microsoft Visual Studio 2
2.3 TortoiseGit 3
2.4 Dropbox 4
3 EDITOR EXTENSIONS 5
3.1 Advantages 5
3.2 Disadvantages 6
4 EXTENSIBILITY OF MAJOR GAME ENGINES 7
4.1 Unreal Engine 7
4.2 Extendibility of Unreal Engine 8
5 IMGUI 9
5.1 IMGUI Events 10
5.2 Basic Controls 11
5.3 Advanced Controls 12
5.4 Layout modes 12
5.5 Changing the Appearance of GUI 13
5.5.1 GUIStyle 13
5.5.2 GUISkin 13
6 SERIALIZATION 14
7 EXTENDING UNITY EDITOR 15
7.1 Editor Window 15
7.2 Creating Editor Window 16
7.3 Inspector Window 18
7.4 Custom inspector 19
7.5 Scene View 20
7.6 OnSceneGUI and OnSceneGUIDelegate 20
7.7 Gizmos 21
7.8 Handles 22
8 CASE ASTEROID CRUSHERS 23
8.1 Developed tools 24
8.1.1 ObjectyPool 25
8.1.2 Spawning System 26

8.1.3 Spline path tool 28
9 CONCLUSIONS 30
9.1 Successes 30
9.2 Failures 31
9.3 Development ideas 31

1 INTRODUCTION

This thesis covers how a developer can expand the Unity game engine through editor scripting. The thesis will also show also more advanced ways to modify the game engine. The advantages and disadvantages of making one's own tools are discussed. Unity is also compared to the highly popular game engine: The Unreal Engine and how it performs in extensibility compared to Unity. The thesis also explores what the developer needs to know before the start of developing the tool(s). For example, the developer must know how Unity handles the saving data or how the GUI is drawn to the screen. There are some important factors that must be taken into consideration even before the start of the development.

The idea of this thesis arose from a personal and professional interest in the making editor extensions in the Unity game engine. Several editor extensions were made during the thesis which was used in a game project. The game was developed in the practical part of the thesis supported by the above-mentioned editor extensions. This was done because it would give a more practical example of how editor extensions can be beneficial to the development.

Unity was chosen as target platform because Unity has indisputably the largest marketplace for selling different kinds of assets, it offers great support for their development and it has friendly, large and active developer base which helps their fellow developers.

The idea of this thesis started when I saw the video on YouTube on how to expand the Unity game engine through editor scripting. The idea grew from a simple example to a full-fledged game project supported by several editor extensions. The idea of these editor extensions came from editor extensions which are available for purchase in Unity Asset Store. Space themed shoot 'em up game was created during practical part of the thesis. More information on the project and extensions can be found from the very end of this thesis.

2 TOOLS

2.1 Unity

Unity is a cross-platform game development environment created by Unity Technologies. With Unity, the developer can create three-dimensional and two-dimensional games and simulations which can be translated into 27 different kinds of platforms relatively easily with small changes to the source code. Six major versions of Unity have been released and it has four different kinds of versions: Personal, Plus, Pro and Enterprise. Nowadays Unity has only one scripting language: C#. Boo is deprecated and UnityScript is deprecated now (Wikipedia, 2018a).

One of the core strengths of the Unity game engine is that the developer does not have to write source code again when changing the target platform. The developer can write own editor extensions or even buy them from Unity Asset Store. This makes the Unity game engine very versatile and flexible. The Unity game engine can be used almost in any situation. One of major flaw is/was old and outdated C# scripting language, but there is now some light in the horizon as C# 6.0 can be used even though it is still experimental at the moment (Unity, 2018a).

2.2 Microsoft Visual Studio

Visual Studio is developed by Microsoft. Visual Studio is highly customizable and popular IDE which can be used to develop such as computer programs and websites. It includes a code editor supporting IntelliSense (the code completion component). Visual Studio also offers a different kind of built-in tools: code refactoring, code profiler, an integrated debugger, etc. (Wikipedia, 2018b).

Unity offers native support for three different kinds of IDEs. These are Visual Studio, Visual Studio Code and JetBrains Rider. Visual Studio is delivered by default when Unity is installed on Windows and macOS. Visual Studio and JetBrains Rider does not require any prerequisites to work. Visual Studio Code other hand requires some extra prerequisites. Visual Studio Code also does not support 4.6 .NET framework debugging. Visual Studio Code and Visual Studio are free to use, but JetBrains Rider is trialware and after 60-days of usage developer must buy a license. Visual Studio Code and JetBrains Rider are most lightweight while Visual Studio is massive in size. Visual Studio also requires a more powerful computer to run it while JetBrains Rider and Visual Studio Code can run on an older computer (Unity, 2018g).

There are also unofficial IDEs which are not supported by Unity Technologies. Almost all of these requires extra work and they do not have the same kind of support as the above-mentioned IDEs which are officially supported by Unity. Here is a list which has some form of support when developing games with Unity: Atom, MonoDevelop, Eclipse, Emacs, Vim, etc.

Reasons for choosing Visual Studio as IDE was it does not require any prerequisites to work, it is free to use to use and it is officially supported by Unity. JetBrains Rider, Visual Studio Code and unofficial IDEs did not meet these requirements.

2.3 TortoiseGit

TortoiseGit is a Git revision control client. It is implemented as a Windows shell extension and it is based on TortoiseSVN. TortoiseGit is free and its released under the GNU General Public License (Wikipedia, 2018c).

There are also countless other programs which can be integrated with git repositories. Here is a small list which can do the same at least as TortoiseGit: GitKraken, SourceTree, GitHub Desktop, etc. Requirements for choosing were it must be free to use, previously familiar, easy to install and be able to integrate with Bitbucket. There were only two programs that I am familiar with: TortoiseGit and GitHub Desktop. GitHub Desktop would have been chosen, but it only works for GitHub. TortoiseGit was an only potential program to be chosen.

2.4 Dropbox

Dropbox is a file hosting service operated by Dropbox. Dropbox offers cloud storage, file synchronization, personal cloud, and client software. With Dropbox different kind of files can be shared and they can be accessed through any device which is connected to the account (Wikipedia, 2018d).

There are again countless programs which can do as same as Dropbox. Requirements for choosing a file hosting service was It would have to be free-to-use, familiar and easy integration with Windows. Amount of storage was not a problem as a file hosting service was used to only save and share documents between devices. Google Drive or Microsoft SkyDrive were other alternatives to the thesis. Either of these could have been chosen as they offer the same kind of features. Dropbox was chosen because of the above-mentioned requirements and personal preferences.

3 EDITOR EXTENSIONS

This chapter is going to explore why one should make their own editor extensions. There must be always a reason to develop them and they have also their own advantages and disadvantages. These things must be taken into consideration before the start of the development.

By making own tool(s), it can be an asset for a game development project. Even if game development project is big or small by making own tool, project could achieve better readability for user interface, non-programmers can develop game further without knowledge of programming, created tools can be used in future again, developers could even earn more money by selling them for example in Unity Asset Store and potentially speed up your developing speed. In some cases, developing own tools can be even more harmful than beneficial for the project.

For developing own tool(s), developers must consider which development platform they are using. Some game engines are closed and they cannot be extended any further. On the other hand, some game engines are specially designed for the specific game genre and they have already premade tools build inside the game engine. For example, a universal game engine like Unity tries to be an all-rounder providing the necessary tools to build a game. Developers must keep in mind what is games genre, what is the chosen game engine, what kind of team one has etc. Developers must consider many kinds of technical and practical issues before they can start to develop their own tools.

3.1 Advantages

Developing own tool(s) for a project it could potentially be advantageous for a game development project. For example, in a project where there are repeated work tasks and they are done by hand. The problem arises if the developer is constantly changing these settings manually which could produce errors caused

by human nature. By developing a simple automation tool could potentially save time and money by making these tasks automated (Tadres 2015, 2).

By changing the appearance of game engines GUI, developers could potentially enhance the project's workflow. In the Unity, the developer can alter how GUI looks by using editor scripting (Tadres 2015, 2). By changing how GUI looks like it would be especially good for designers who are non-programmers achieve their goals much faster and easier. It could also make it much easier for the new team members to understand what kind of variables needs to be changed to achieve their goals (Smith & Queiroz 2015, 507-508).

The Advantages of making own tools are the possibility of improving workflow, minimizing errors and speed up the development. Some other benefits are developer could earn money by selling tools and potentially reuse developed tools in other projects (Smith & Queiroz 2015, 530).

3.2 Disadvantages

There are of course disadvantages of developing a tool(s). It takes time from developing the actual game itself and it could even prolong development time even more than developers gain from it. Developers must consider how much time it takes to develop the tool(s). There must be some benefits for developing the tool(s). More complex projects usually benefit more from the tool(s) than smaller projects. Big projects are more complex and more error-prone, for example, by automating some tasks developers could reap huge benefits (Blow, 2004).

If developers are considering buying premade tools for example from the Unity Asset Store, they must consider a couple of things. These premade tools always have a learning curve which takes again time from actual development itself (Petteri Paju, 10-11).

Tools can also be error-prone which can cause data loss and even crash the whole program. It is important the tool is well tested before it is distributed as It could cause more problems and thus increasing again developing time. If also game design changes, the tool would also need to adapt to changes and again it is going to take time from actual development again (Petteri Paju, 10-11).

Before developing the tool(s) there are two things to remember: always evaluate the advantages/disadvantages what is gained from developing the tool(s) and developers must think the user who they are building the tool as nobody wants the tool does not solve the specific problem and is not easy to use (Tadres 2015, 43).

4 EXTENSIBILITY OF MAJOR GAME ENGINES

This chapter will discuss the extendibility of two major game engines. These game engines are Unity and Unreal Engine. Extensibility is a big part of development because this gives developers the freedom to do what they want to. That is why it is good thing developers can extend the game engine by making tool(s) for their projects. These game engines are not made for one type of game and that is why developers must do something if they want to speed up development.

4.1 Unreal Engine

Unreal engine is a game engine developed by Epic Games. Although the engine was developed for first-person shooters still many successful games have been developed by it in a variety of other genres including stealth, fighting games and MMORPGs. Current version of Unreal engine is version 4 and it is designed for Microsoft Windows, macOS, Linux, SteamOS, HTML5, iOS, Android, Nintendo Switch, PlayStation 4, Xbox One, Magic Leap One, and virtual

reality (SteamVR/HTC Vive, Oculus Rift, PlayStation VR, Google Daydream, OSVR and Samsung Gear VR) (Wikipedia 2018g).

4.2 Extendibility of Unreal Engine

Extendibility of Unreal Engine cannot still compare to Unity. Unreal has an equivalent store where plugins can be bought or sold. Still, Unreal Engines Market Place relatively small when comparing it to Unity Asset Store. From Unreal Engines Marketplace, developers can buy assets like code plugins, textures, 3D models, etc. Some of these free of charge and for some must be paid.

While making research how a developer could extent Unreal Engine it was found out it is much harder than Unity because there is nonexistent documentation for it. The developer cannot find really place where could start developing own tools. Unreal Engine offers whole game engines source code. They assume developer itself would learn how built-in tools are made in Unreal Engine. This sets a high bar for new developers to start making new tools for their games. Unity is the clear winner in this part because the learning curve is much easier when a developer starts developing own tools. Developers can find many video tutorials, both official and unofficial ones, documentation is very nice with simple examples which can be used to start developing. Unity really has developed their architecture in this part very well, making their game engine highly extensible. Even though Unity is lacking some parts but it is nothing compared to the Unreal Engines situation.

When comparing which game engines marketplace has more users, asset creators, downloads and assets Unity is the clear winner. Unity has over around 40 million downloads since 2010 when Unity was launched. Unity also has 12 million downloads yearly and its growing bigger every year. Unreal Engine has around 8 million downloads. Unity also has around 56,000 different kinds of asset packages from tenths of thousands of creators in their Asset Store. Unreal Engine other hand has only 5000 asset packages from 1500 creators on its Marketplace. The only thing where at the moment Unreal Engine beats Unity is they give more revenue from sales than Unity. Unity gives you a 70% cut from sales and Unreal Engine gives you 88% (Venturebeat, 2018).

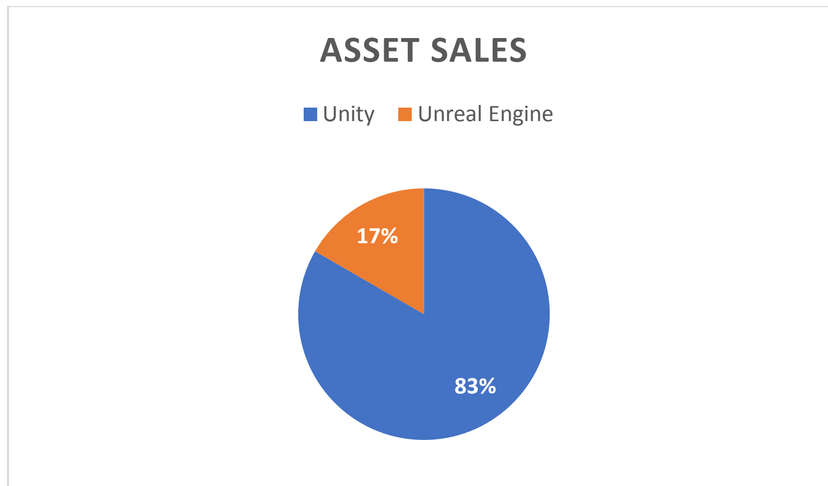


Figure 1. Asset Sales

5 IMGUI

To draw a graphical user interface or add intractability to the custom tool, the developer must use and understand the tools which Unity offers. This chapter will discuss the very core of extending Unity.

IMGUI means the “Immediate Mode” GUI system which is entirely separate nowadays from Unitys GameObject-based UI system. IMGUI was previously Unitys primary system for creating in-game UIs. IMGUI is primarily used nowadays for making in-game debugging tools and extending Unity editor itself. If a developer wants to create IMGUI elements, the code must be written for every GUI element. GUI code goes to a method called OnGUI. IMGUI updates and draws developers written code every frame making it ideal for capturing keyboard and mouse events (Unity, 2018b).

5.1 IMGUI Events

Whatever IMGUI OnGUI-function runs, there is currently “Event running” being handled. This event could be the user has started dragging or the user has clicked the mouse button. Unity has tied up both event system and graphical user interface in the same structure. The basic flowchart of IMGUI is shown in Figure 2. (Unity, 2018b).

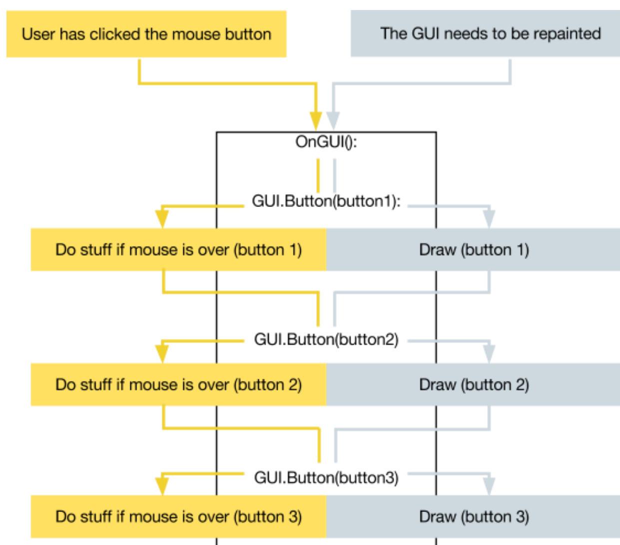


Figure 2. Unity Event System (Unity 2015h)

Unity has tied up Event system withIMGUI, so its functions example basic button also corresponds to users inputs. There is a different kind of event types, but in Table 1. are the most common ones. Rest can be found from Unity documentation for different EventTypes (Unity, 2015h).

Table 1. Event Types

EventType.MouseDown	The user has just pressed a mouse button
EventType.MouseUp	The user has just released a mouse button
EventType.KeyDown	The user has just pressed a key
EventType.KeyUp	The user has just released a key
EventType.Repaint	IMGUI needs to redraw GUI to screen

IMGUI also has a concept of “control-id” which will give one consistent way to refer to a given control across every event type. Every distinct part of UI that has some meaning requests for control-id. It is used example keep a track which UI element has keyboard focus or which UI element has dragged. (Unity, 2015h)

5.2 Basic Controls

IMGUI offers several different kinds of controls, some are for displaying information and some for interaction. Below are all basic controls which IMGUI offers. These can be found in GUI and GUILayout classes. These can be also found partly from EditorGUILayout- and EditorGUI-classes. Developers need to understand what controls have at your disposal to create a GUI for their custom tool (Unity, 2018c).

Table 2. GUI Controls

ControlType	Description
Label	The label is non-interactive. Used to display information.
Button	Typical interactive element. Responds to users clicks on-time per click.
RepeatButton	Variation of a basic button. The only difference will respond to click every frame.
Textfield	Used to display edit strings.
TextArea	Used to display and edit multiline strings.
Toggle	Creates a checkbox with a persistent on/off state. The user can change the text
Toolbar	Essentially a row of buttons
SelectionGrid	Essentially multi-row toolbar
HorizontalSlider	The horizontal sliding knob which is can be used to drag between min and max values
Vertical Slider	Same as above, but in the vertical direction
HorizontalScrollbar	Similar to the slider, but visually resemble scrolling elements of web browsers or word processors
VerticalScrollbar	Same as above, but in the vertical direction
ScrollView	Used to display a viewable area of the much larger set of controls
Window	The draggable container of controls

5.3 Advanced Controls

IMGUI offers also a more advanced set of controls which can be found under EditorGUILayout- and EditorGUI-classes. These set of controls are used to build custom inspectors and editor windows. Without these two classes creating the custom tools would be impossible. Both of classes are way too big to cover in this thesis because these are way bigger than GUI or GUILayout. With these controls, the developer can display and edit your variables from the scripts.

5.4 Layout modes

There are two different ways how a developer can build GUI using the IMGUI system: Fixed and Automatic. Fixed layout uses a class called GUI which requires manual position and size for its controls. The automatic layout uses a class called Layout which handles positioning and element size automatically. It is still possible to use both to build your GUI, but the automatic layout does not need much extra work like calculating sizes and positioning making it faster making tools (Unity, 2018d).

Depending on which layout mode developer are going to use. For the fixed layout, the developer can put controls together using a method called group. Using an automatic layout, the developer can arrange controls together in many ways: Areas, Horizontal - and Vertical groups (Unity, 2018d).

Groups allow defining areas where to group up your controls. The developer can define controls inside the group by using GUI.BeginGroup- and GUI.EndGroup-methods. All controls inside these functions start from the top-left corner. If the developer update groups position on runtime, also other controls position will be updated according to groups position (Unity, 2018d).

Areas are used only in automatic layout mode. The idea behind of area is similar to fixed layouts group as a developer must provide coordinates and size for the area. Controls inside the area are placed at the upper-leftmost point of its containing area. The develop can alter the flow of GUI inside area using Vertical and Horizontal groups (Unity, 2018d).

5.5 Changing the Appearance of GUI

Unity offers two ways to change the look and feel of GUI controls, GUISkin and GUIStyle. Before Unity version 4.6 these two classes were used to change the appearance of the in-game GUI. Nowadays these are used change appearance Editor GUI. Changing the appearance of the tool could potentially give the more professional look and identity to a developers tool (Tadres 2015, 147).

5.5.1 GUIStyle

The GUIstyle class is part of UnityEditor namespace and its used change appearance single GUI control, such as button or label. Most of IMGUIs controls accept GUIStyle as a parameter to override their default style. GUIStyles can be created in code or by creating a GUIStyle file inside Unity Editor (Unity, 2018f).

5.5.2 GUISkin

GUISkin is a collection of GUIStyles. It can be also used to customize how your GUI seems. It is intended to allow one to apply a style to whole GUI, unlike GUIStyle which changes the appearance of the single component (Unity, 2018e).

6 SERIALIZATION

This chapter is going to discuss what is serialization and why it is an important part of creating custom tools. Developers must understand how they can design the structure of their tools because Unity handles the saving of data in unexpected ways, and because of that it is very important to know how tools can be designed in the Unity.

Serialization means the process of converting an object into a stream of bytes to store the object or move it to memory, a database, or a file. The main purpose of serialization is to save the state of an object to able to recreate it when it is needed. The opposite operation is called deserialization where developer extracts data structure from series of bytes (Wikipedia, 2018f).

Unitys serialization was written in C++. Serialization is used widely in Unity game engine. Some of Unitys build-in features use serialization; the inspector windows, saving and loading prefabs, etc. Two components are particularly aware of with serialization are Inspector window and hot reloading (Unity, 2018i).

Unity can serialize any object that derives from Object class. Serialization has some rules that must be followed in order Unity will serialize them. The developer must ensure field in the script is public, non-static, non-const, not read-only and has field type that can be serialized by Unity. Private fields have an exception if they are marked with SerizlizeField attribute. Public fields also have an exception if they are marked with NonSerialized attribute they will not be serialized in the inspector. Serialization works for common data types like integers, strings, floats, doubles, etc. Serialization also works for Unitys own built-in types like Vector2, Vector, Vector4, Rect, etc. Unity also can serialize custom structs and classes that are marked with attribute Serializable (Unity, 2018i).

As stated before Unity cannot serialize everything. Unity lacks serialization support for fields that are static, constant, read-only or hasn't field type which Unity cannot serialize. Unity also lacks support for polymorphism for custom classes or structs. This represents a problem as making complicated data structures which rely heavily on polymorphism: both in-game and custom tools.

It is essentially important to know how serialization works and what limitations developers has when working with Unity. The Developer must understand how serialization works as it affects how tools can be designed inside Unity.

7 EXTENDING UNITY EDITOR

Unity is a great platform for developing games. Unity also provides many built-in tools for developers but as Unity tries to be as flexible as possible Unity cannot cover all possible tools and game genres. Luckily Unity has provided a different kind of tools to solve this problem. Unity has its own API for developing its own editor windows, custom drawers and inspector panels. (Tadres 2015, 2). This chapter covers how a developer can extend Unity in three different ways.

7.1 Editor Window

When a developer needs to interact with multiple objects, using the Custom Editor Window could be the solution. Unity allows for creating editor windows using the EditorWindow-class. In Unity everything is rendered using Editor Window, so one could say editor window is a container to almost all GUI elements which are found from Unity. Editor windows can float freely, resized and can be docked as a tab inside Unity. (Tadres 2015, 86). Editor Windows are opened usually using a menu item. Unity recycles its editor windows so if the menu item is pressed example it would just open the current Editor Window (Unity, 2018j).

There can be any number of editor windows in Unity project. Editor windows are a good way to add new tools for the games. To create custom editor window, it requires three steps: create a script which derives from EditorWindow-class, editor window needs to be triggered by some method (usually using menu item) and lastly developer needs to write custom GUI code for the tool. The created script also must be inside the Editor folder (Unity, 2018k).

7.2 Creating Editor Window

This chapter discusses how to create a Custom Editor Window in Unity. As stated above there are three things needed to get Editor Window working: create a script which derives from EditorWindow-class, it needs to be triggered by some method (usually using menu item) and lastly custom GUI code is required to be written for the tool. Creating an editor window is an alternative way to write a GUI for the tools (Unity, 2018k)

Start by creating a folder inside the Unity project. All editor scripts must be in a folder called editor because Unity will handle its contents in a different way. Secondly, the script is needed to be created inside of the Editor folder by right-clicking and by selecting Create -> C# script. After script can be named for the fitting. After creating it, the next step is to start up a programming development environment (Tadres 2015, 88).

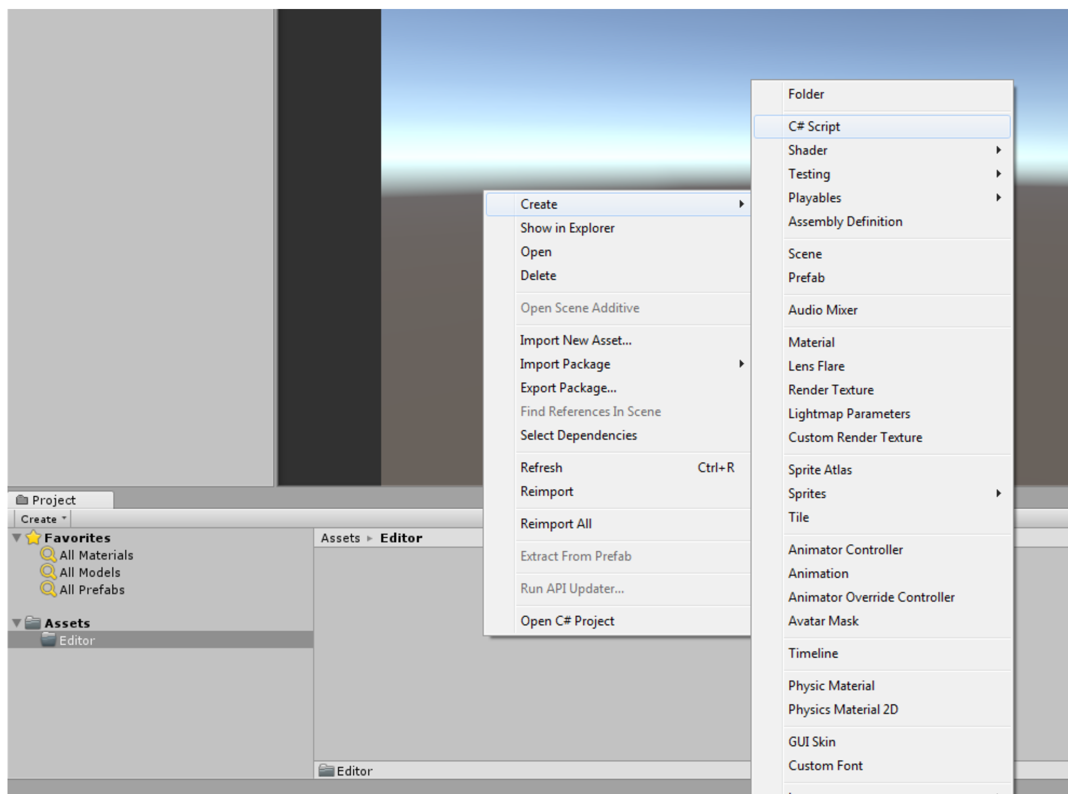


Figure 3. Creating Editor script

To create Editor Window, the class it should be derived from is EditorWindow instead Editor or MonoBehaviour. EditorWindow-class resides in the UnityEditor namespace. A static method is needed to be written which encapsulates GetWindow call. This method handles opening the editor window itself. The GetWindow-method is part of EditorWindow-class and it is responsible for getting an instance of the specified type of window. An instance of this of editor window is needed to be saved to the static variable (creating Editor Window follows singleton pattern) (Tadres 2015, 88).

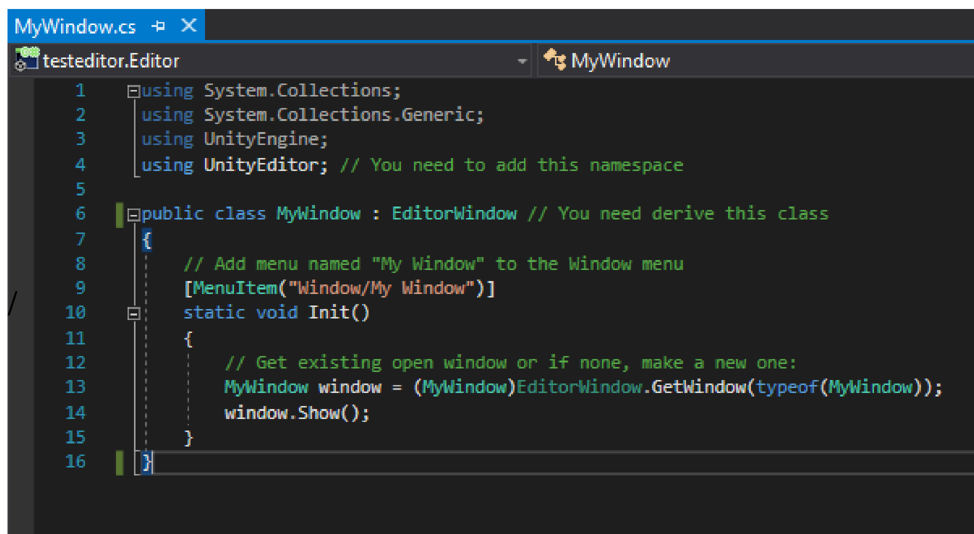


Figure 4. Creating Editor Window

To draw in the Editor Window, Unitys OnGUI-function must be used because it handles all GUI drawing. This part is the biggest challenge because the developer must design how the GUI is going to look like. Methods from EditorGUILayout or GUILayout can be used to approach this problem. These two are easier and faster to do the job, but there are also alternative ways to draw the GUI. Methods from EditorGUI or GUI can be used also, but this requires more work as for these two must be provided position and size for each component manually. Best results are achieved if one is not afraid to jump between two approaches (Tadres 2015, 111).

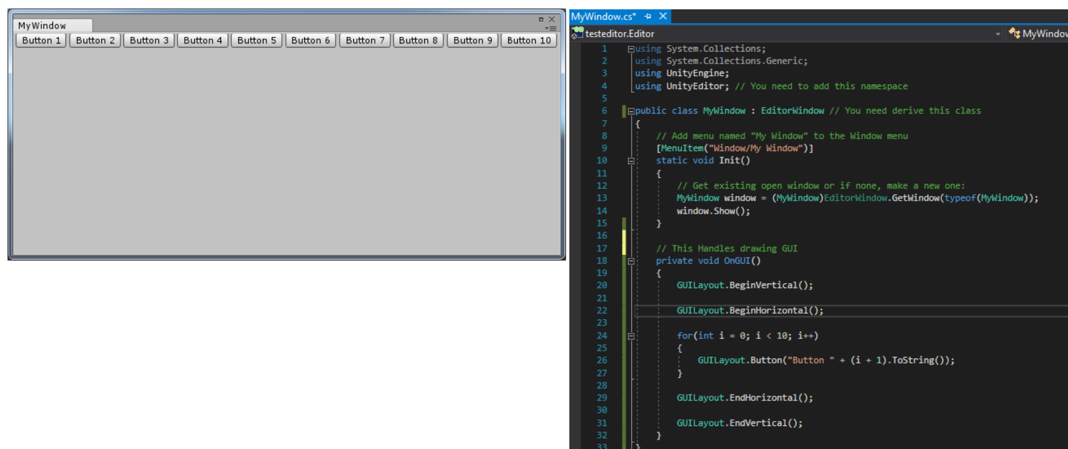


Figure 5. Drawing GUI to Editor Window

7.3 Inspector Window

Before a developer can start customizing the Inspector Window, understanding of what the Inspector Window is and how it works is needed. Inspector Window is used display and edits the selected game object, asset or other preferences and settings in the Editor. When the developer has selected one of the above, the Inspector will show the properties and settings of the selected object. Every Inspector Window can be modified through editor scripting because every Inspector Window is derived from a class called Editor (Unity, 2018l).

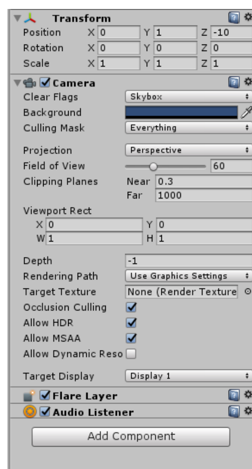


Figure 6. Inspector window

7.4 Custom inspector

The second way to extend Unity is by creating Custom Inspectors. The Custom Inspector uses Editor-class which is part of UnityEditor-namespace. It is necessary to make a Custom Inspector. Three things must be done to get it to work. Every Editor file must reside in an Editor called folder, the created script must be derived from Editor-class and it also needs CustomEditor-attribute. OnInspectorGUI-function must be overridden to draw custom GUI for the Inspector. With these three steps, the developer can alter how the associate class should be drawn in the Inspector. Editor-class also has its own messing methods like the MonoBehaviour class has. These are called OnEnable, OnDisable and OnDestroy (Tadres 2015, 84).

Now the developer can start drawing custom GUI by an overriding method called OnInspectorGUI. This method must be overridden to draw a custom GUI. The developer can use methods from EditorGUILayout and GUILayout to build the GUI. The developer can also use methods from EditorGUI and GUI to solve this problem (Tadres 2015, 84).

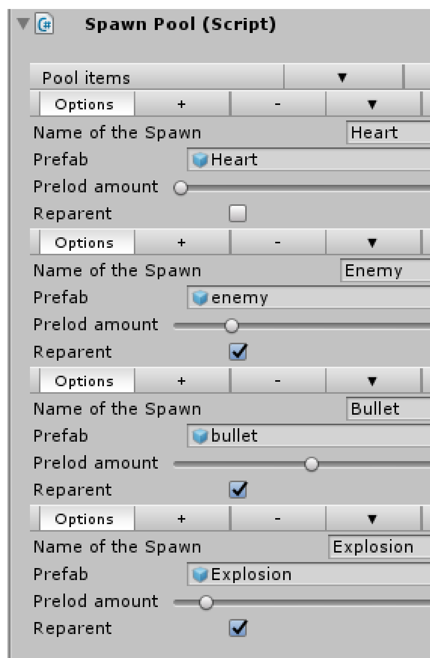


Figure 7. Objecty Pool using Custom Inspector

7.5 Scene View

Scene View is Editor Window that is developers interactive view world what developer is creating. Scene View can be used for example to alter properties of an object, example position, scale and rotation. Almost everything is extendable in Unity and Scene View is no exception. Different kind of events can be intercepted from Scene View, the GUI can be drawn like to any Editor Window and custom 3D GUI called Handles and Gizmos can be drawn (Unity, 2018m).

The developer can extend Scene View by an overriding method called OnSceneGUI which is part of Editor-class. The developer can also extent SceneView by an overriding method called OnSceneGUIDelegate which is part of SceneView-class. The easiest way to interact with SceneView is to create Custom Inspector because OnSceneDelegate is part of Editor-class. One can also alter Scene View using OnSceneGUIDelegate making possible to alter scene from custom Editor Windows. One can also extent Scene View by writing custom Handles and Gizmos (Petteri Paju, 25-26).

7.6 OnSceneGUI and OnSceneGUIDelegate

There are classes which allows one to interact with Scene View through code. To understand how a developer could interact with Scene View through code is very important when start creating more complex and interactive tools for the project. Unity offers two methods which can handle the events from Scene View itself.

OnSceneGUI-function belongs to Editor-class and when creating custom inspectors, it becomes easy to interact with Scene View same time. This method works only when a script is derived from Editor-class and object is currently selected (Petteri Paju, 25).

OnSceneGUIDelegate is an undocumented method which enables interaction with SceneView without regardless of the selected specific object. OnSceneGUIDelegate belongs to SceneView-class and it is called every time scene is drawn (Petteri Paju, 25).

7.7 Gizmos

Gizmos are used in Unity for debugging and visual purposes only. Gizmos are not interactable like methods in Handles-class. Gizmos can be used in MonoBehaviour scripts by implementing OnDrawGizmos- and OnDrawGizmosSelected-methods (Petteri Paju, 26).

The developer can draw with gizmos shapes like lines, spheres, and cubes. The developer still makes own custom gizmos by companying existing shapes to create new ones. Gizmos can also draw icons and textures, they must be located under a folder called Assets/Gizmos (Tadres, 54-59).

Figure 8. Weapon system using Gizmos

7.8 Handles

The handle is a 3D control which can be used to manipulate items in Scene View. There are many built-in Handles, such as tools to change position, scale and rotation via the Transform component. Handle-class is like Gizmos class, but the main difference between these two components is that Handle-class is intractable. The developer can also draw GUI by using methods from IMGUI to Scene View. This is done by using Handles specific methods called BeginGUI and EndGUI. The developer can also write own Handles by using events from chapter 5.1 and HandleUtility-class (Tadres, 136).

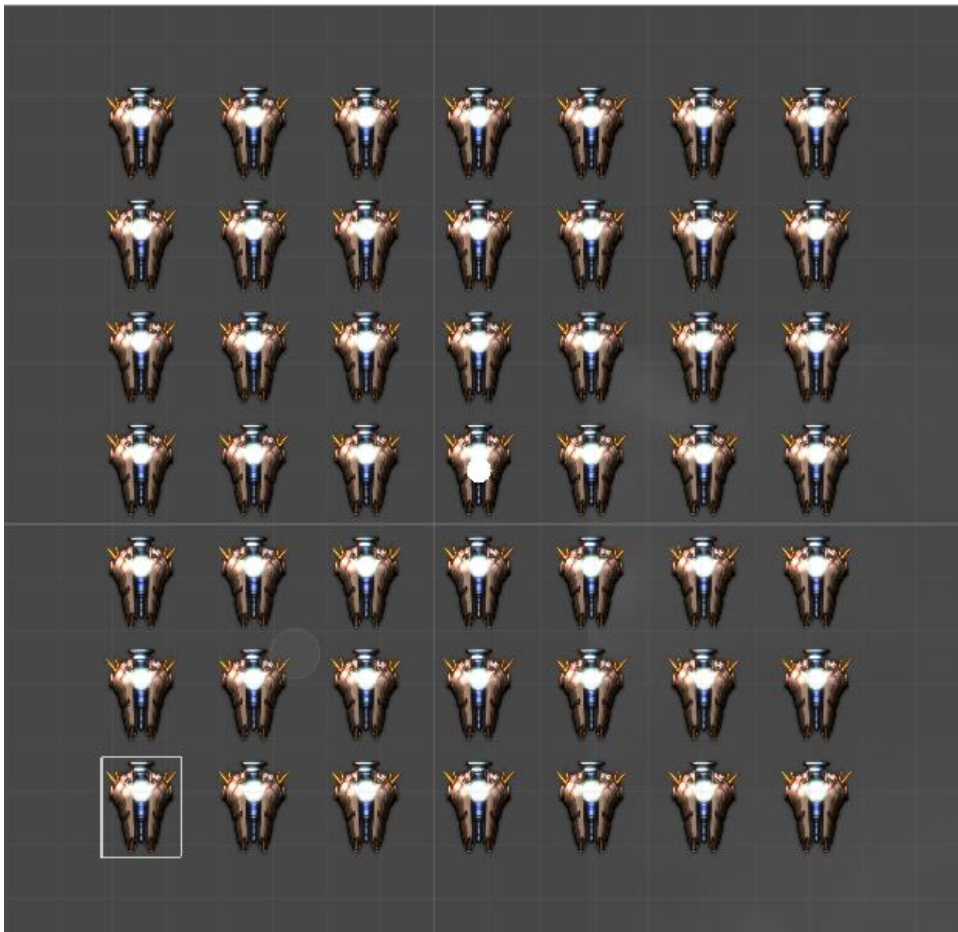


Figure 9. Spawning system using Handles-class

8 CASE ASTEROID CRUSHERS

The idea for this game was from a game project that I was doing four years ago. I still had graphics with me which I bought from GraphicRiver.net. I wanted to make a game that would demonstrate how the editor tools could help development. This project is not for demonstrating game design UI experience or anything else. This project is just to give a practical picture of what advantages of by developing tools have.

The idea of the game is basic space themed shoot ‘em up where player flies a spaceship. The player fights hordes of aliens while trying to survive all the waves. If the player survives he unlocks the next level. Player also has multiple weapons which he can use. These weapons can be picked up from fallen enemies. There are weapons such as a laser, homing missile and rapid fire. The player can also view high scores, change settings, buy new upgrades and ships and select mission of choice.

Tools that were developed for this game project would support development speed primarily. A game designer could try out new features without asking for help from the programmer. For example. if the game would be too easy, a game designer could change easily how spawning system works without changing the code itself. This makes testing much easier and same time it does not eat the development time. The tools would also benefit more from even larger projects where features are changing even more.

During development, I did not have any major problems if you do not count on developing the tools itself. I will tell about my tool specific setbacks in their intended chapters. Most of the work was developing the GUI which took most of my development time. Developing rest of intended features like player controls were an easy task compared to developing the tools itself.

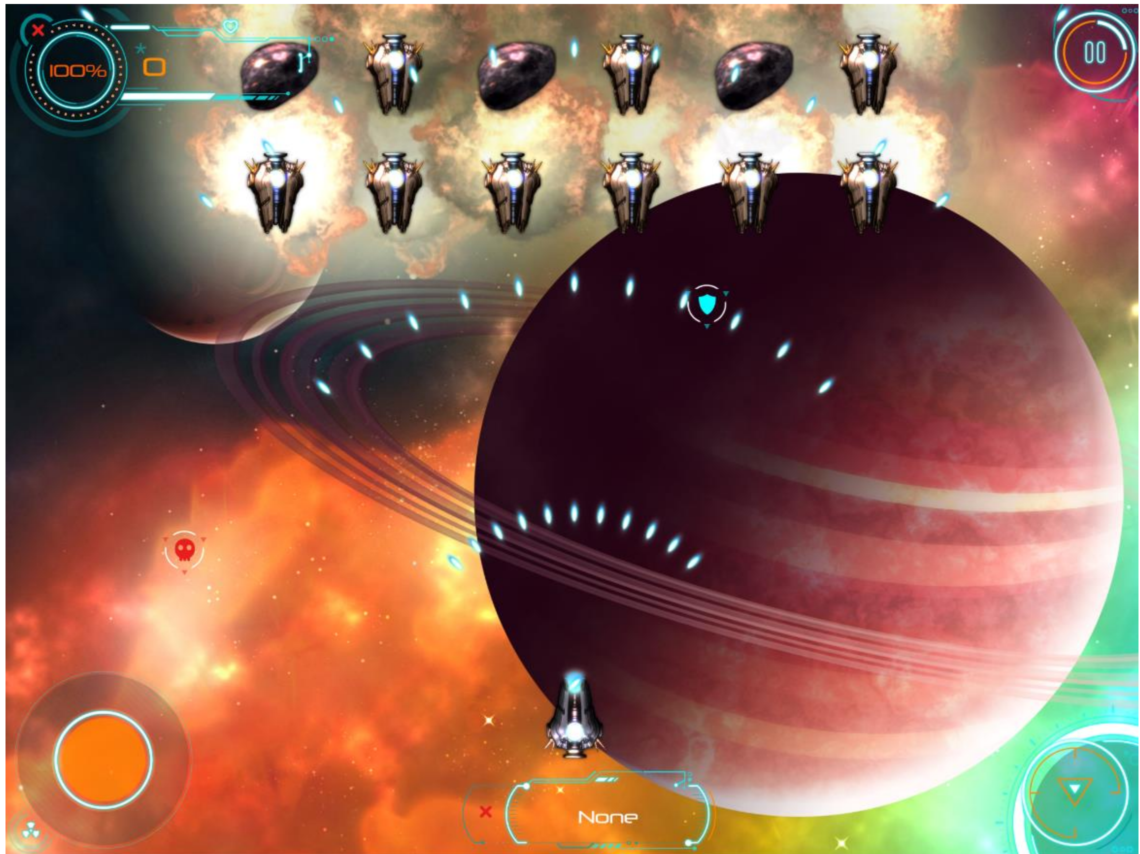


Figure 10. Asteroid Crushers

8.1 Developed tools

For this project, four different kinds of tools were developed. In the following chapters, I will tell more about these tools in more detail. I wanted to make tools which would support development speed and performance. With these tools, a person without technical expertise could develop the game further without the assistance of a programmer. Tools try to be user-friendly and easy to use. I also tried to make tools as generic as possible, so they could be used again in the future.

8.1.1 ObjectyPool

The first tool that was developed for this project was a performance boosting pooling system. The tool was done to improve the performance of the game as the player must deal hordes of enemies and shoot a lot of projectiles. The game would generate garbage collection if I created and destroyed objects all the time. It would be better if I created a fixed number of objects and reused them while the game is running. The tool would be generic and could be used almost in any project.

I started first by making a script that would contain all information regarding spawning. After that, I made a script that holds a list of this class inside of it, this class also would hold all methods for spawning and despawning the objects from the pool. I also wrote singleton because I wanted to access easily the pools without writing variable for them. See Figure 12. for the class diagram.

I also wrote a Custom Inspector to support a more user-friendly experience. By default, the ObjectyPool uses the list as a data container. Default list in inspector seems confusing and hard to edit. I did not have any major technical problems, but I had to think about how I would design the GUI in that way it would easy to use. In Figure 11. you can see the difference between these approaches, how the GUI designed vs. without any editing

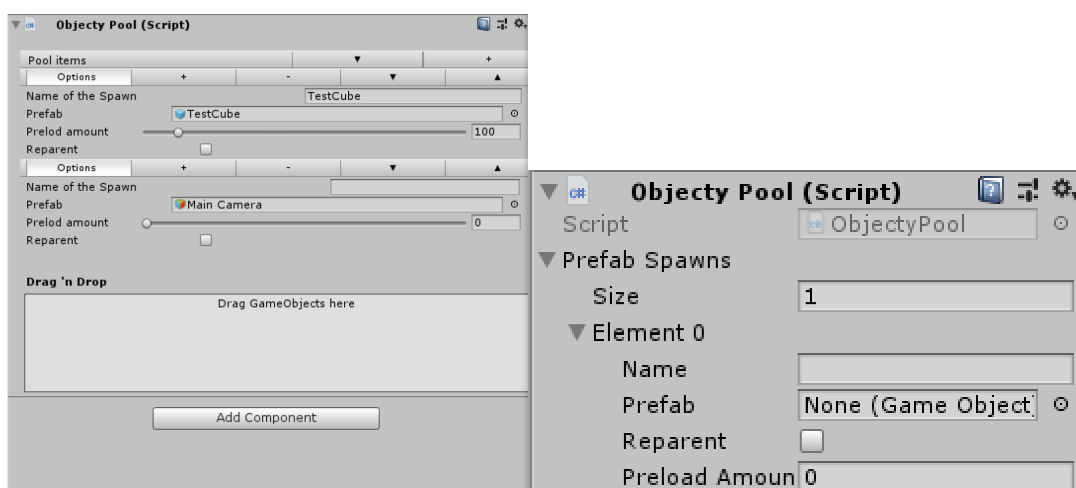


Figure 11. ObjectyPool

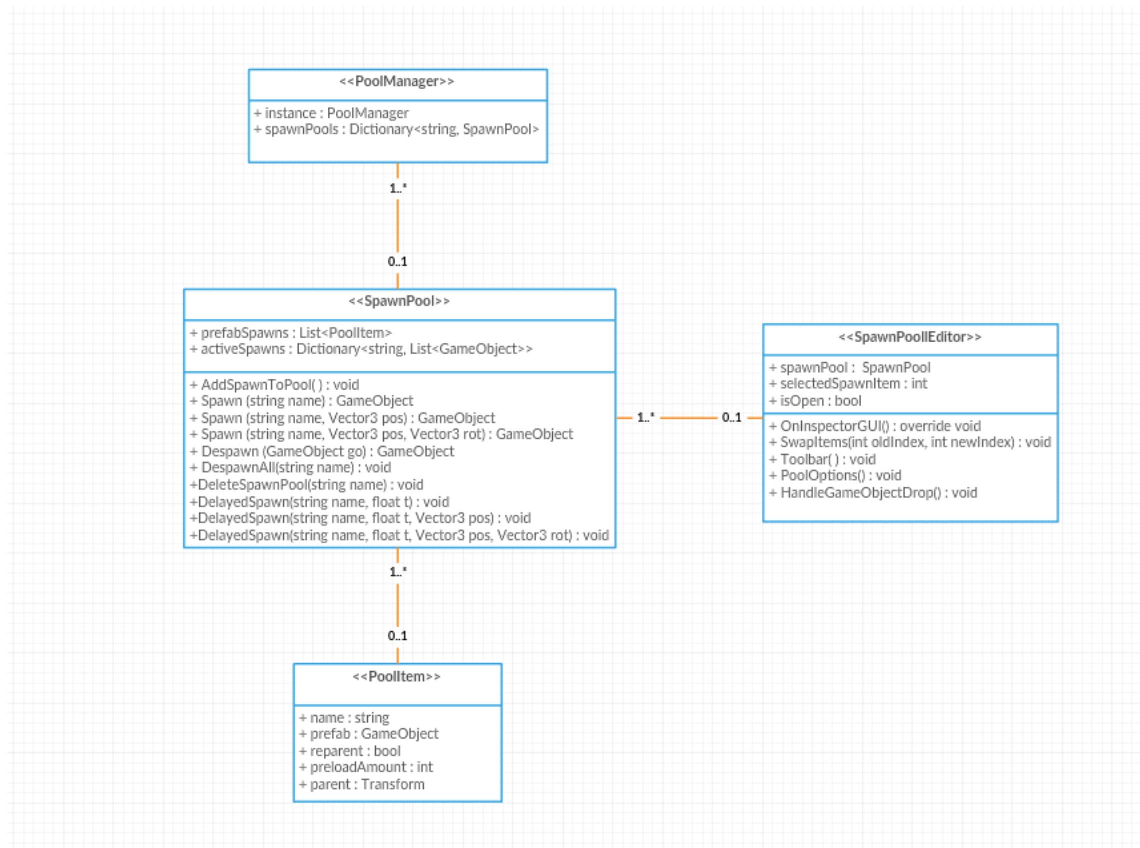


Figure 12. ObjectyPool class diagram

8.1.2 Spawning System

The second developed tool was the spawning system. This tool allows to spawn and visually design different kinds of formations for example lines, grids, circles and triangles. This tool would be the most important tool that was developed for this project. Spawning system was developed to speed up the development and improve workflow inside Unity. This tool was also integrated with PoolManager.

Most important part of the spawning system lies in Scene View. Even developer without programming experience can design spawns using visual tools. The developer can see what they are doing. Developers do not have to a guess how spawn would seem and where it is coming. The designer also can change the sprite of the spawn and behaviour with the click of the button.

I started by creating two classes called WaveItem and Wave. WaveItem-class would contain information about one spawn and Wave-class was a container for a multiple of WaveItems. These two classes would be holding all the data regarding the waves. After that, I created a class called Spawner that would be holding the list of waves and actual functionality for spawning. This way I would have access to all the waves and items inside of it. See Figure 13. for the class diagram.

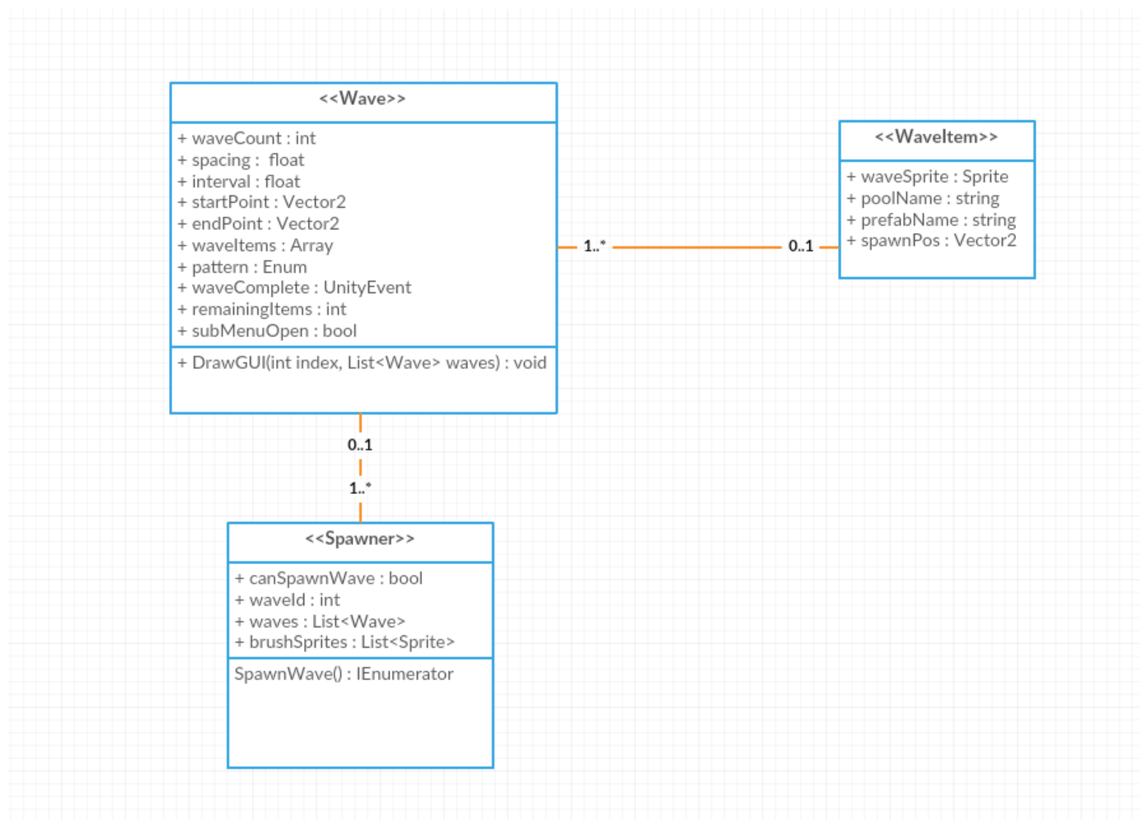


Figure 13. Spawner class diagram

I wrote again Custom Inspectors GUI for the same reasons as I did for Objecty-Pool. I also added features like the order of spawns can be changed with the click of the button and sprites can be dragged and dropped to the Inspector. I also integrated it with ObjectyPool because I wanted all the tools natively with each other. This also makes the spawning system more generic and could be used in other projects.

I had some hurdles while designing this tool. Most of the problems I had with saving the data, especially with Scriptable objects. I first designed the tool support the polymorphism, but I abandoned the idea because the tool would not need it first place. After solving that problem, developing tool became much easier. Another major problem I had in developing this tool was how I would draw a visual presentation of spawns in the Scene View. I used Handles-class and EditorUtility-class to solve this problem.

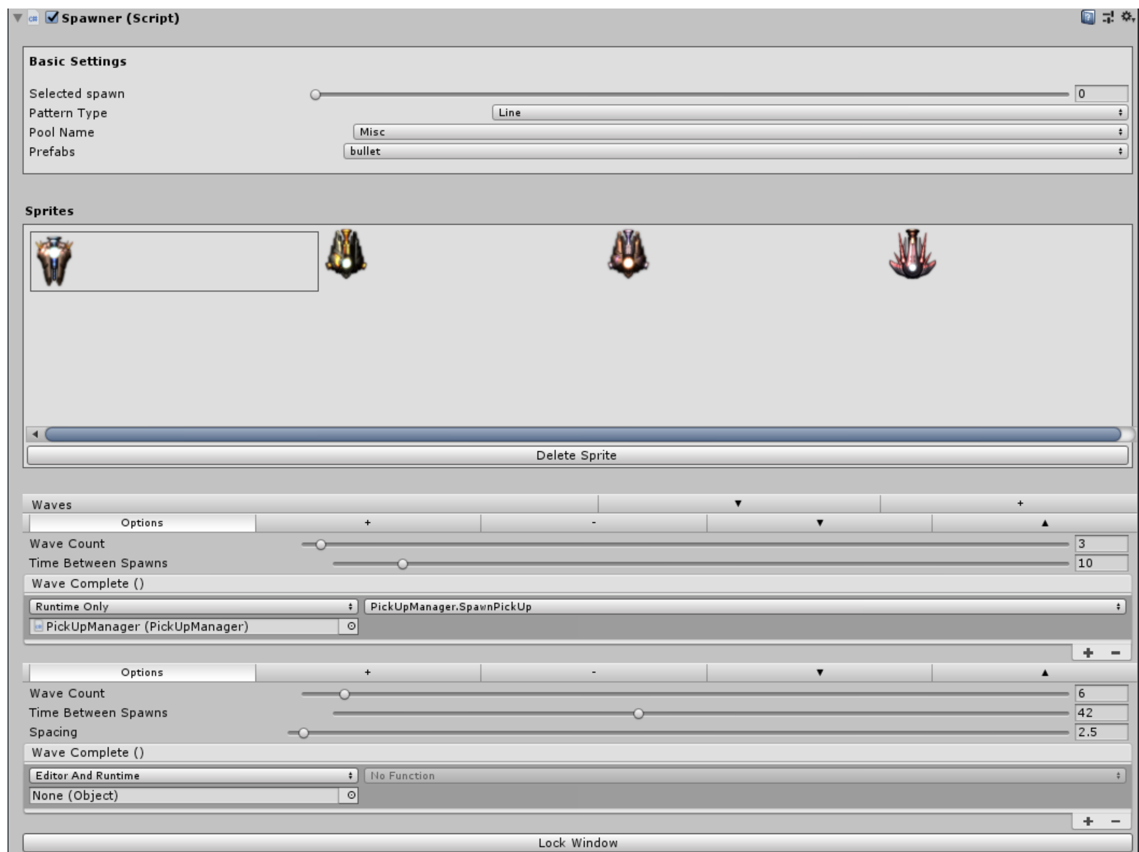


Figure 14. Spawner

8.1.3 Spline path tool

The third and final tool that was developed for this project was a spline path tool. This tool allows developers to create two-dimensional curvy lines in the Scene View. This tool was used to create different kinds of enemy movement. At the moment only one kind of Bézier curve is supported.

I started first by creating a script called CubicBezierCurve which all the information regarding the curve. This class was marked with Serializable-attribute because otherwise, Unity would not save data in the inspector. I created parent script that would hold an instance of CubicBezierCurve-class. See Figure 15. for a class diagram

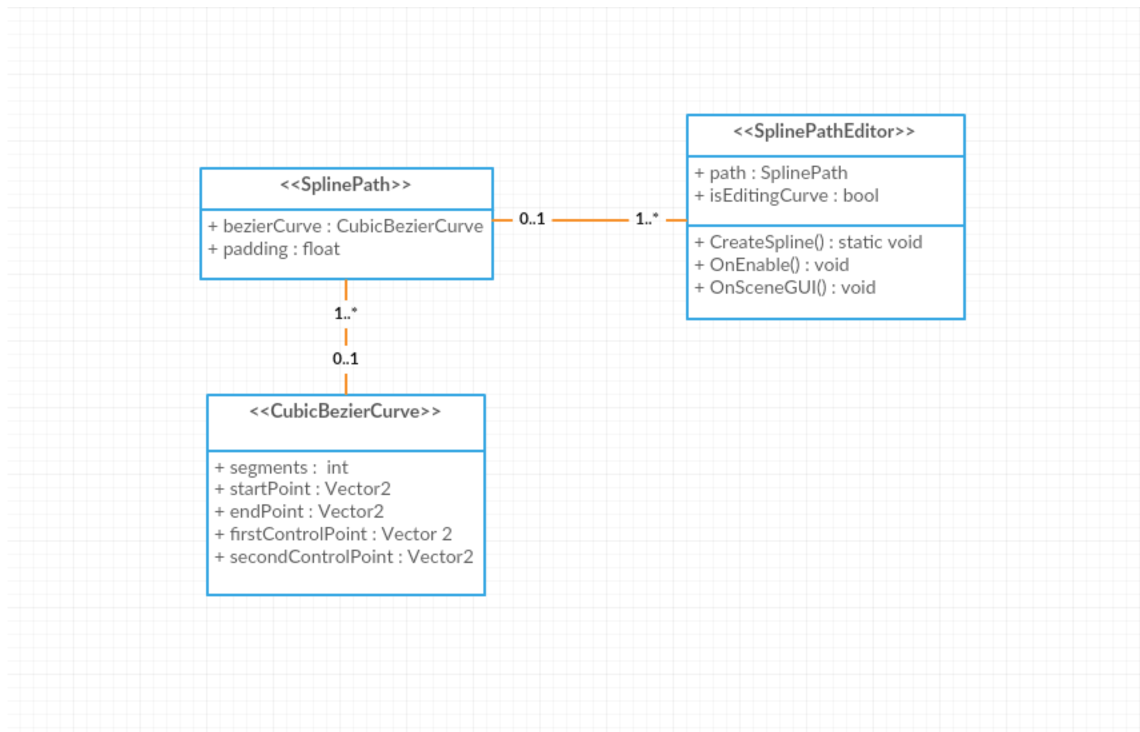


Figure 15. Spline path tool class diagram

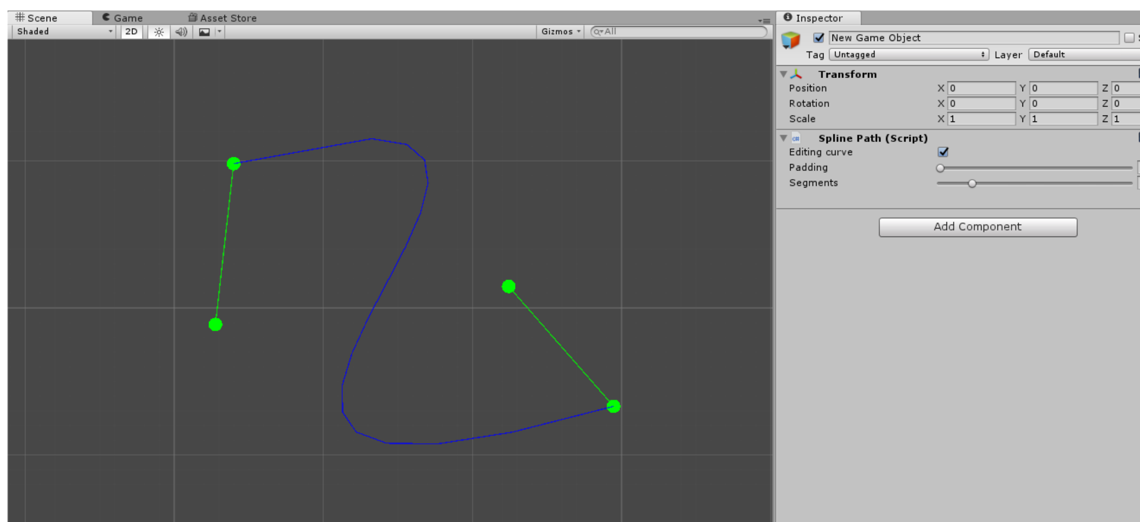


Figure 16. Spline path tool

9 CONCLUSIONS

The purpose of this thesis was to show how one could start developing own tools for their projects. Thesis covered all of the most important ways how one can extend Unity editor for their benefit. The thesis discusses also advantages and disadvantages of developing your own tools has. I myself strongly encourage of making own tools, because it helps to grow as a programmer, it can really speed up development by a mile and gives you the possibility of reusing own developed tools in future projects.

For me, the hardest part of the process really was the writing of the thesis. I did not realize how much time-consuming writing would be and how much time It would take from actual development itself. I would say that I managed very well as I had a very broad subject and I wanted to develop an actual game to support my tools.

From the technical perspective, I would say the hardest part was that I forgot almost everything about programming and Unity during my four-year hiatus. The biggest challenge was getting back to the game but it really helped a lot that I had other school courses before starting doing the thesis itself. Developing game and tools was straightforward and I did not have any major problems developing a game or necessary tool for it.

9.1 Successes

In my opinion, the practical and theoretical parts of the thesis were a success. I was able to develop a working game with many extra features which were not on my development list. I also was able to produce all number of editor extensions while fully working and integrated to the actual game. The first goal was just to develop the necessary tools and provide simple examples to show how developed tools would work. I abandoned the idea because hard drive of mine broke taking down all the tools and theoretical document of this thesis along with it.

9.2 Failures

I did not have any major failures during developing or writing this thesis. Both parts still had their own ups and downs. Had some problems with lack of motivation due to weariness. The practical part of the thesis grew also too enormous as I did not realize how big task would to develop three editor extensions and playable game same time. I was still able to produce more during this thesis than I could hope for.

9.3 Development ideas

There is no such thing as perfection so there is always room for improvement. In this section will tell my thoughts how I could improve in theoretical and practical parts of the thesis.

In the practical part of the thesis, there is much room for the improvement. The game and extensions itself would have to be finished. The game has at the moment two working levels and there are at least ten more to do. The spawner would need some features added, for example, support for spline paths. It would also need different kinds of formations, for example, arrows and spirals. ObjectyPool itself is quite ready at the moment and it does not need anything to be added. Spline path tool would need many features to be added. It only supports one spline at the moment. It would need to support multiple splines in order to build more complex paths. A major amount of development time would still go to developing the actual game as developing extra features to the editor extensions would take only a couple of hours of development time.

REFERENCES

Blow J. 2004. Game Development: Harder Than You Think.

http://queue.acm.org/issuedetail.cfm?issue=971564 1.9.2018

Paju Petteri. 2017. IMGUI EXTENSIONS IN UNITY3D

http://www.theseus.fi/handle/10024/138902 31.8.2018

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

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

Unity Technologies. 2018a. Updated scripting runtime in Unity 2018.1.

https://blogs.unity3d.com/2018/03/28/updated-scripting-runtime-in-unity-2018-1-what-does-the-future-hold/ 15.8.2018.

Unity Technologies. 2018b. Immediate Mode GUI
https://docs.unity3d.com/Manual/GUIScriptingGuide.html 3.8.2018.

Unity Technologies. 2018c. Immediate Mode GUI Control Types
https://docs.unity3d.com/Manual/gui-Controls.html 3.8.2018.

Unity Technologies. 2018d. Immediate Mode GUI Layout Modes
https://docs.unity3d.com/Manual/gui-Layout.html 3.8.2018.

Unity Technologies. 2018e. GUI Skin
https://docs.unity3d.com/Manual/class-GUISkin.html 4.8.2018.

Unity Technologies. 2018f. GUI Style

https://docs.unity3d.com/Manual/class-GUIStyle.html 4.8.2018.

Unity Technologies. 2018g. Integrated development support.
https://docs.unity3d.com/Manual/class-GUIStyle.html 13.8.2018.

Unity Technologies. 2015h. Going deep with IMGUI and Editor customization.

https://blogs.unity3d.com/2015/12/22/going-deep-with-imgui-and-editor-customization/ 15.8.2018.

Unity Technologies. 2018i. Script Serialization.

https://docs.unity3d.com/Manual/script-Serialization.html 20.8.2018.

Unity Technologies. 2018j. Editor Window.

https://docs.unity3d.com/ScriptReference/EditorWindow.html 21.8.2018

Unity Technologies. 2018k. Editor Windows

https://docs.unity3d.com/Manual/editor-EditorWindows.html 20.8.2018.

Unity Technologies. 2018l. Inspector Window

https://docs.unity3d.com/Manual/UsingTheInspector.html 20.8.2018.

Unity Technologies. 2018m. Scene View

https://docs.unity3d.com/Manual/UsingTheSceneView.html 20.8.2018.

VentureBeat. 2018. Unity's asset store boss has big plans to fight Epic's Unreal

https://venturebeat.com/2018/07/18/unitys-asset-store-boss-has-big-plans-to-fight-epics-unreal/ 14.8.2018.

Wikipedia. 2018a. Unity Game Engine

https://en.wikipedia.org/wiki/Unity_(game_engine). 25.7.2018.

Wikipedia. 2018b. Microsoft Visual Studio
https://en.wikipedia.org/wiki/Microsoft_Visual_Studio 25.7.2018.

Wikipedia. 2018c. TortoiseGit
https://en.wikipedia.org/wiki/TortoiseGit 25.7.2018.

Wikipedia. 2018d. Dropbox
https://en.wikipedia.org/wiki/Dropbox_(service) 25.7.2018.

Wikipedia. 2018f. Serialization
https://en.wikipedia.org/wiki/Serialization 2.8.2018.

Wikipedia. 2018g. Unreal Engine
https://en.wikipedia.org/wiki/Unreal_Engine 3.8.2018.