Tampereen ammattikorkeakoulu

Additional Unity Tool Creation for Game Development Using Unity Editor Scripting

Heikki Gauffin

OPINNÄYTETYÖ
Joulukuu 2022

Tietojenkäsittelyn tutkinto-ohjelma
Game Production

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittelyn tutkinto-ohjelma
Game Production

GAUFFIN, HEIKKI:
Unity-lisätyökalujen luonti käyttäen Unity Editor -ohjelmointia

Opinnäytetyö 44 sivua, joista liitteitä 0 sivua
Joulukuu 2022

Opinnäytetyössä tutkittiin pelituotannon lisätyökalujen tarvetta ja luontia. Tavoitteena oli antaa opinnäytetyön tilaajalle, Tampereen ammattikorkeakoulun pelituotannon yksikölle, perusteluja Unity-pelimoottorin lisätyökalujen tarpeellisuudesta ja niiden luomiseen liittyvistä mahdollisuuksista ja haasteista. Opinnäytetyön tarkoituksena on antaa aloitteleville pelinkehittäjille ja opinnäytetyön tilaajalle ideoita ja ehdotuksia lisätyökalujen käytöstä pelituotannossa ja pelituotannon opetuksessa.

Opinnäytetyön alussa perehdyttiin pelimoottorien historiaan ja kehitykseen, joka on johtanut tämän päivän korkeasti mukautettaviin pelimoottoreihin. Lisäksi opinnäytetyössä selvitettiin, miksi pelimoottoreihin on tarvetta kehittää lisätyökaluja ja miksi niitä hankitaan ulkopuoliselta taholta. Opinnäytetyössä keskityttiin tarkemmin Unity-pelimoottoriin, koska sitä hyödynnetään tilaajan pelituotannon opetuksessa. Opinnäytetyöprosessin aikana työn kirjoittaja toteutti muutamia käytännön esimerkkejä Unity-pelimoottorin lisätyökaluista omaan peliprojektiinsa.

Opinnäytetyön tuloksena syntyi käytännön esimerkkejä ja ehdotuksia opinnäytetyön tilaajalle ja muille sidosryhmille siitä, miksi ja miten Unity-pelimoottorin lisätyökalujen luonti voi tukea pelituotantoa ja pelituotannon opetusta.

Opinnäytetyön tuloksista voidaan havaita lisätyökalujen hyödyt, mutta myös lisätyökaluihin liittyvät tarve- ja resurssihaasteet. Nämä pitää huomioida lisätyökaluja kehittäessä tai hankittaessa. Opinnäytetyössä luotuja lisätyökaluja voidaan jatkokehittää ja hyödyntää myös pelituotannossa.

Asiasanat: peli, pelimoottori, pelituotanto, Unity, työkalu

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Game Production

GAUFFIN HEIKKI:
Additional Unity Tool Creation for Game Development Using Unity Editor Scripting

Bachelor's thesis 44 pages, appendices 0 pages
December 2022

The thesis studied the need and creation of additional tools for game development. The commissioner of the thesis was the Game Production study path of Tampere University of Applied Sciences. The objective of the thesis was to provide reasons for the creation of the additional tools for the Unity game engine and the practical challenges and opportunities associated with it.

The research focused on the reasons and concerns for additional tool creation or acquisition from third parties for game development. The thesis concentrated more closely on the Unity game engine which is utilised by the commissioner of the thesis in their game production education. The practical case study of the thesis involved the practical creation of few additional tools for the Unity game engine.

The results of the thesis are practical examples and recommendations for the commissioner of the thesis and other stakeholders on why and how additional tool creation for Unity can assist in game development and game development education. The findings indicate that while these additional tools can help with game development, there are concerns regarding the need and available resources that must be addressed.

Key words: game, game engine, game production, Unity, tool

SISÄLLYS

1 INTRODUCTION

The Unity Game Engine is one of the leading game engines utilised by both big and small game developers worldwide. Due to the easy availability and popularity of the Unity Game Engine, institutions of higher learning are providing courses and lectures on using the Unity Engine. The Unity Game Engine however does allow the customization of the Unity Editor that is used to create the games that run on the Unity Game Engine. This applies both to additional tools created by the developers themselves or tools that have been created by third parties that the developers can utilise.

The commissioner of this thesis was the Tampere University of Applied Sciences which is a Finnish higher education institution that is oriented towards working life. During the thesis author's studies in Tampere University of Applied Sciences in games production, the author saw the need and personal professional curiosity to look further into the possibilities of extending the Unity Editor with additional tools for game development. This would serve the objective of the commissioner of the thesis of providing reasons for the creation of the additional tools for the Unity game engine and the practical challenges and opportunities associated with it. This would also help to gain a better picture of how the Unity Editor extensions are perceived by new starting game developers as well as satisfy the need of the author of the thesis to gain more professional knowledge in the subject.

The purpose was to give starting game developers and the commissioner of the thesis ideas and recommendations for the utilisation of additional tools to as-sist game development and game development education. This is achieved through research into the game engine history, their utilisation in game projects today and through a practical example case provide a concrete understanding of what the creation of additional tools for the Unity Editor entails.

Due to time and resource constraints, the thesis does not aim to create big component or tool examples but rather investigate the practicalities by utilising smaller examples to give a springboard for ideas for further development and things to look out for.

2 GAME ENGINES

2.1 Game Engines in general

Games of varying types which utilise different technologies have existed for hundreds of years. Even before the advent of the digital games most gamers these days associate with gaming, at the core of the games has always been a game engine. Something that makes the game function as intended by their creators. (Williams 2017.)

Despite of this, the term game engine was not widely used or understood as a separate concept until as late as the mid 1990’s. The arcade games which launched the popularity of digital games in the 1970’s and 1980’s was highly specialized and customized to work on the hardware they were created for. All the core software of a game were created within the specifics of the game and device in question which led to difficulties with the reusability of the same approach to different games. (Gregory 2018.)

The term game engine, and indeed the concept of a game engine we understand to be today, can be attributed to the success of the video game Doom from id Software in the 1990’s. The key to this was the separation of some of the core components of the game software utilised in the games creation, such as the graphics and physics systems. This allowed the utilisation of the separated components in other projects by allowing the modification of the existing game or when building new games. This led to the creation of the later games in the late 1990’s and the engines they run on to support modification and customization via scripting out of the box. (Gregory 2018.)

Therefore, the term game engine is these days used to describe the tool sets that are highly modifiable and customizable for a specific game project. The core components of the game are not directly tied and hard coded only for that specific game or device. This has led to the licensing of the game engine toolkits to other developers to create their games and modify the tools available to fit their specific needs. (Gregory 2018.)

2.2 Game Projects and extending game engines

Since the game engines these days are highly customizable and can be utilised to create practically any kind of a game, the question of what engine to use and how it can be made more efficient becomes more pertinent.

Jonathan Blow (2004) outlined already back in the year 2004 how complex the video game projects have become since their inception and highlighted the importance of proper development tools or the lack of them that causes issues in game development projects. The technologies might have changed and progressed, but the developers of the day are still faced with the same questions when it comes to successfully creating a video game. The biggest questions concern the foundational tools of the game project, the game engine, but also due to their customizability these days, extends to the possible single components or tools as well. (Blow 2014.)

2.2.1 Own propriety game engines

One possibility when choosing a game engine for a game project is to create one from scratch to support the needs of the game project. Many of the leading game developers of the biggest games have gone to create their own game engines to support their game project needs. (Gregory 2018.)

When the game engine is created from the ground up by the developers, it allows for much more customization and the engine can be built to specifically support a certain game genre or components and it can be tailored to fit the specific needs of the developer. (Gregory 2018.)

The main advantage of creating your own game engine is keeping the knowledge of how it functions within the company. Since the knowledge of incorporating the technology the game runs on resides within the developers themselves, this does not cause the same possible issues as utilising a third-party game engine might where that knowledge lies outside the company. (Blow 2004.)

The creation of a completely new engine may also rise from or lead to innovation of making some game concepts better than ever before. It may also unlock an additional revenue stream for the developers by being able to license the new engine for other developers to use as well. However, the scope of creating a completely new game engine is a huge task and requires a lot of resources, at least if the engine is supposed to support anything than rudimentary type of game functionality. Independent and small developers most of the time simply do not have the resources to create a completely own game engine, so they need to look for alternative options. (Martin 2020.)

2.2.2 Third-party game engines

The reality for most individual game developers and studios is that creating a new game engine from scratch is an impossible task to accomplish. Whether it be due to lack of resources or the knowhow, utilising a third-party licensed game engine or an open-source game engine for a game project considerably reduces the number of required resources. (Gregory 2018.)

The most enticing option for a game engine for a game project might be the free open-source game engine alternatives which are multiple. Open-source means that the source code for the game engine is usually open for all to edit which allows developers to modify and make changes to the engine as needed. However, this can also lead into issues as the actual functionalities which these open-source game-engines come with might be lacklustre or missing completely. It also requires knowledge to utilise such an engine and documentation, or guides might be lacking. Therefore, the open-source engines vary in quality greatly and careful consideration should be given into it and which open-source game engine to utilise in a game project. (Gregory 2018.)

The benefits of utilising a licensed game engine developed by a third-party is the obvious immediate access to most of the tools that are needed to create a wide variety of games for different platforms. Usually, the licensed game engines have started out as an in-house game engine that has been created for a specific game title and has then been worked on further to license it to other developers, usually for a fee. Utilising these licensed third-party game engines means that time available for the project can primarily be used to create the actual in-game content rather than the tools. The licensed game-engines also usually feature a more robust documentation, guides and due to their popularity also offers a community guide and help that might not be available in other game engines. (Martin 2020.)

While the benefits of utilising a licensed third-party game engines are clear, it also poses certain questions that needs to be considered. First, there usually is a license fee that needs to be paid to the game engine's developer to utilise the game engine for a game project and release it to the market. The fee can be contrasted with the required resources to build a completely new game engine and how much resources it requires. Luckily some of the bigger licensed game engines such as Unity and Unreal Engine have taken into consideration the upfront cost of utilising their game engines by offering smaller scalable and easily achievable tiers for newly starting game developers or studios to use their game engines where the license cost is waived until certain revenue figures are hit or taking of a small percentage of the revenue the released games generate. (Gregory 2018.)

Secondly the utilisation of a third-party licensed game engine still requires a lot of knowledge of what type of issues or things the game needs to do to achieve the goals of the game project and successfully utilise the tools of the engine to reach that. The game systems have become more and more complex and the licensed game engines can only go so far with fulfilling the needs of every possible scenario and game systems the developers want. This means that to implement the systems as the developers envision, the game engines need to be customized and utilised in certain ways to achieve those goals or in some cases look for other third-party or custom solutions for the specific functionalities. (Blow 2004.)

2.2.3 Game development tools and third-party components

Since the creation of your own game engine is a very resource incentive task and the open-source or licensed game engines cannot fulfil all the possible needs for a specific game project, the need for additional game development tools is an increasing requirement to deliver games as they have been envisioned by the developers.

Since the game engines and specific game engine components have become quite modular, it has allowed the developers to either customize or completely re-create certain game engine components to better suit their specific game project. Even the bigger gaming studios who might utilise their own propriety game engines as a core in their game development process are sometimes opting to use a third-party component for a specific issue that they feel the need the current tools are not capable of solving or do not support the fulfilment of their vision. One example of this is the audio system where multiple different high quality third-party solutions such as FMOD and Wwise exists which can be utilised if the default tools of the game engine are not up to the task. There also are other rather large third-party solutions for other specific game components such as physics engines, terrain generation and others. (Barclays 2021.)

Like the questions and considerations that needs to put into the selection of the game engine, the choice of creating your own game development tool, component or whether to utilise third-party solution also has its own benefits and drawbacks. The reason to utilise additional third-party components for game development is to reduce the cost of the development so that the limited resources can be directed into creating content for the actual game and usually the most successful third-party components are higher quality than what is already available. On the flipside, licensing and utilising someone else’s component for a game project can also lead to unexpected issues in trying to integrate the components to the actual game and how it should function. It might take much more resources to integrate and troubleshoot someone else’s implementation of the game component than it would have taken if you had created it yourself. (Arendt 2008.)

There can also be non-technical issues that needs to be considered when utilising third-party component or tool solutions since in certain instances the source code might not be available at all or there are certain stipulations in the license agreements that prevent the developer from trying to fix any unforeseen issues themselves. This leads to developers relying on updates and support from the components creator which might not always be straightforward and might take a long time which hinders the game project development. (Barclays 2021.)

While the third-party or own solutions might focus on a larger piece of the game, there are also smaller tools that can be developed or acquired to help in the development process. Especially in larger game projects, when the development team grows, it is important to have the correct tools at hand. Correct tools can help in reducing errors, ensure the correct settings for imported assets and help automate repetitive tasks thus reducing the overall development time. (Tadres 2015.)

Ultimately the question of if an additional tool or component is needed and whether it should be created by the developers themselves or to acquire a third-party solution needs to be carefully considered. The available resources and knowledge need to be carefully weighed against the actual gains from the component or tool and what possible issues may arise regardless of which solution the developers have chosen.

3 UNITY GAME ENGINE

3.1 Unity introduction

The Unity Game Engine is one of the most successful game engines that game developers are using to create their games today. From smaller indie developers to the big gaming studios, many game developers have harnessed the engine to deliver both 2D and 3D games ranging across different genres and gaming devices. (Kok 2021.)

Unity is no longer utilised solely for game development either. It has been adopted as a tool in several other industries such as 3D animated films and virtual reality applications due to its adjustability. (Kok 2021.)

One of Unity’s main advantages is, while the engine is simple to use yet powerful at the same time, that it features the Unity Editor that can be extended to further support your workflow through editor scripting. (Tadres 2015.)

Another contributing factor to the success of Unity is the accessibility for anyone looking to get into game development. The free version of Unity and the huge user community brings game development from the reach of only large gaming studios to individuals. (Blackman 2014.)

3.2 Unity Editor introduction

At the heart of the games created by Unity is the Unity Editor which provides the tools for the game developers to import the game assets and the tools which are used to construct the game which the Unity Engine will run.

The Unity Editor by default provides a layout of the most useful windows and tools which are required to start creating a game in Unity. The basic Unity Editor interface layout which can be seen in Figure 1 contains the scene view, hierarchy window, inspector window, project window and the toolbar. (Unity Technologies 2022a.)

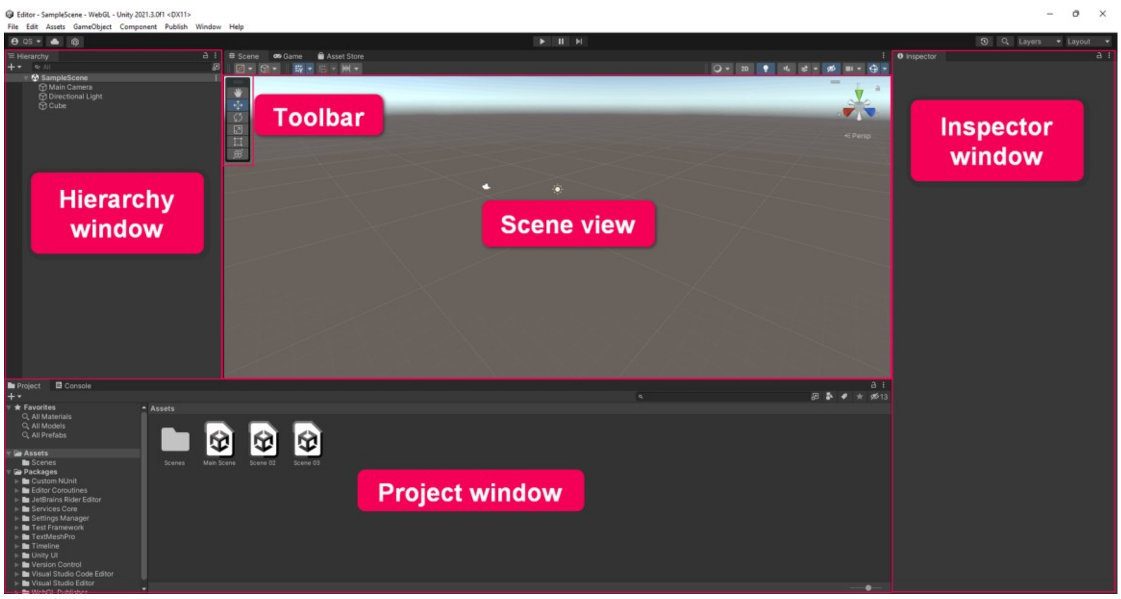


FIGURE 1. Basic Unity Editor interface layout (Unity Technologies 2022a.)

.

These are the most basic tools which are available for a game developer in the Unity Editor. The scene view is used to move and change objects rotation as well as view these objects from different perspectives by utilising the toolbar. The hierarchy view displays all the objects in the scene view as a list for easy accessibility and the inspector window is used to view a single objects information when they are selected. The project window displays to the user the files which are imported and available in the project just like operating system's file explorer views. (Unity Technologies 2022a.)

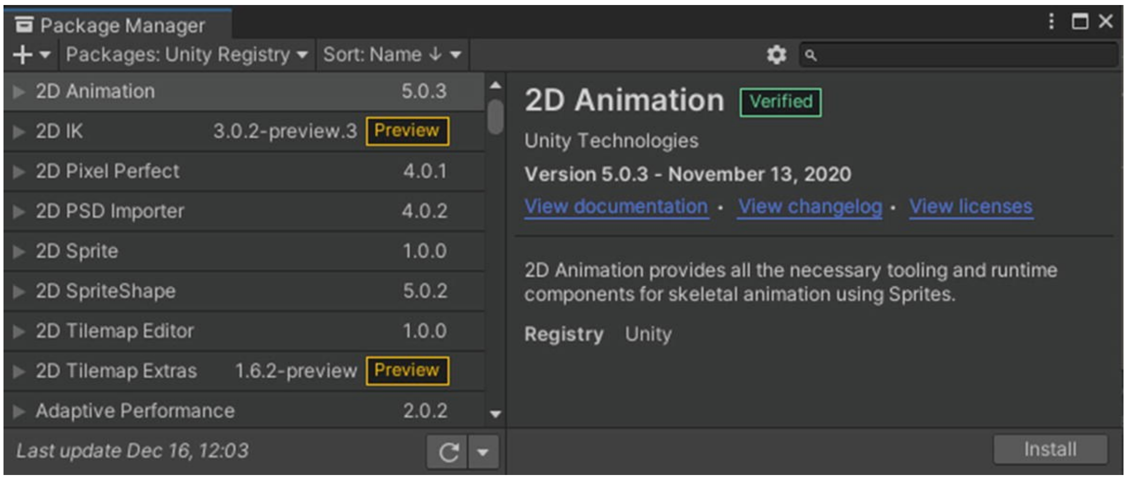
This however is only the default layout of the Unity Editor. The users can customize the different windows sizes and positions as well as hide and enable different editor windows depending on which of them are relevant to the task at the hand. This means that the Unity Editor tools which are available by default can already be customized to fit the user's needs.

3.3 Unity package manager, asset store and third-party tools

3.3.1 Unity Package Manager

The primary additional way of extending the default Unity Editor, which Unity themselves also utilises to provide additional tools for the developers, is the Unity Package Manager.

The Unity Package Manager is a solution to Unity's aims to provide a more easily customizable approach to using the Unity Editor. Separate tools or Unity Editor extensions can easily be installed and updated through this Package Manager whenever a need for them arises as can be seen in Figure 2. The primary aim is to provide the game developers with additional tools that suit their specific situation, but which are not required for the basic functionality of the Unity Editor and thus are not required for every game project. (Kok 2021.)



2D Animation	5.0.3	
2D IK	3.0.2-preview.3	Preview
2D Pixel Perfect	4.0.1	
2D PSD Importer	4.0.2	
2D Sprite	1.0.0	
2D SpriteShape	5.0.2	
2D Tilemap Editor	1.0.0	
2D Tilemap Extras	1.6.2-preview	Preview
Adaptive Performance	2.0.2	

2D Animation Verified

Unity Technologies

Version 5.0.3 - November 13, 2020

View documentation · View changelog · View licenses

2D Animation provides all the necessary tooling and runtime components for skeletal animation using Sprites.

Registry Unity

FIGURE 2. Example view of Unity Package Manager (Kok 2021.)

Since the packages are separated, they can also be updated individually without affecting the functionality of each other. It is however good to keep in mind that the packages and modules acquired through the Package Manager might have different compatibility versions with the Unity Editor version which needs to be considered if the Unity Editor is updated during the game project's lifecycle. The package manager also offers the option for the developers to share their tools and assets between different projects by creating packages from them and importing them to other projects. However, this does create an additional layer of obfuscation to the source code which can hinder possible bug fixing and integration. It might be wiser to utilise a Git repository directly for sharing tools and assets between projects as it allows for faster access to changes to the code and assets if needed. (Tadres 2015.)

3.3.2 Unity asset store and third-party tools

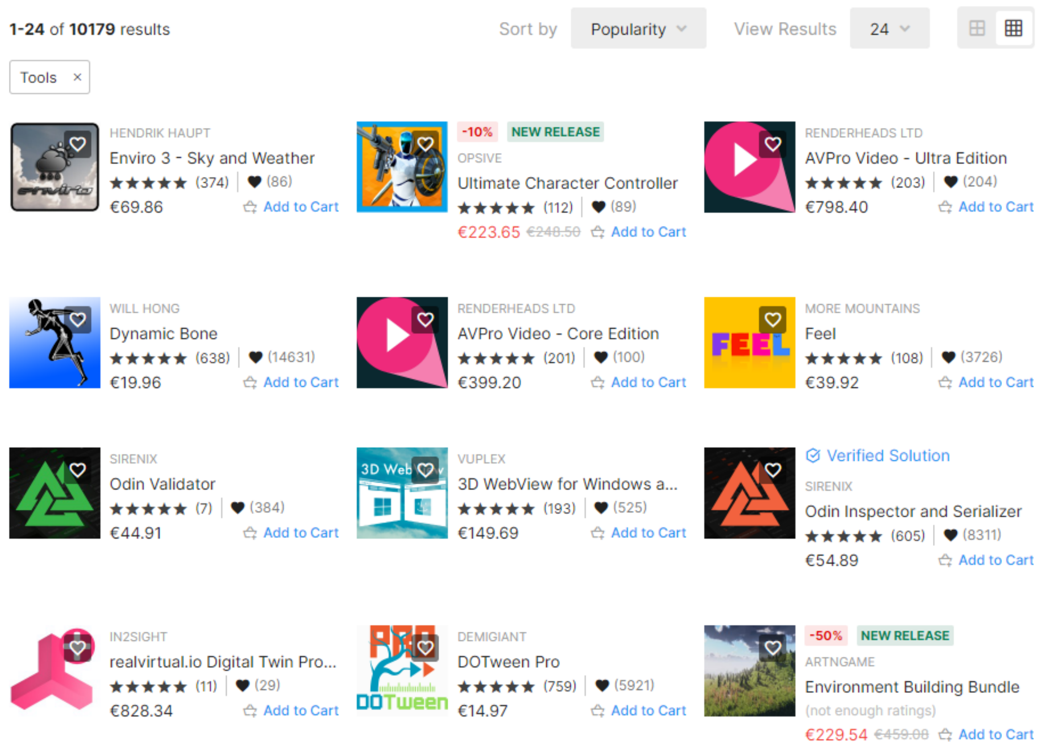
One of the reasons for the success of the Unity Game engine being increasingly the choice of game engine for starting out game developers and experienced developers alike is the community that Unity has managed to build around itself. This is culminated in the Unity Asset Store which allows developers to find and purchase prebuilt assets and tools to assist developers in their game projects. The assets or tools from the Unity Asset Store can directly be downloaded and accessed in the Unity project via the package manager. (Tadres 2015.)

The tools in the Unity asset store can greatly help developers solve problems that have already been solved by others and in general help with speeding up the development process. A tool a developer has developed for themselves might be universal enough that it could benefit others as well. This presents a new avenue for developers to generate extra revenue by commercializing their tool solution in the Unity Asset Store. There are several success stories of game developers making their ends meet with just developing new tools and solutions for the Unity Asset Store. (Kok 2021.)

Indie developers have also managed to fund their game development through publishing their tools on the Unity Asset Store. By making the tools of the game available for purchase already in development phase, the developers can generate revenue from the start, help the developers to create clean and modular code, learn from other tool developers and gain visibility for your game project. It is however good to keep in mind that releasing the tools on Unity Asset Store also sets certain expectations from the customers for support and guidance. This might interfere with the time available for the development of the actual game that the developers are working on. (Hougaard 2013.)

According to Unity's own metrics, the Unity Asset Store has over 1.7 million monthly users and over 12 000 active publishers. While Unity takes a 30% cut of the revenue generated by the tools, such huge user base provides a great opportunity for developers to leverage their expertise through it. (Unity Technologies 2022h.)

The discoverability in the Unity Asset Store in the tools section is handy in finding and evaluating different tools that are available to the developers. The tools can be sorted by categories, popularity, price, and various other parameters. By sorting the tool section by popularity, as can be seen from the Figure 3, the most popular tools in the Unity Asset Store are rather large in feature size and the pricing reflects that. Most of the popular tool's deal with either creating better visual assets easier, their easier manipulation within the game world or tools that can be utilised to develop other tools more efficiently. (Unity Technologies 2022i.)


1-24 of 10179 results — Sort by Popularity — View Results 24

Tools ×

HENDRIK HAUPT
Enviro 3 - Sky and Weather
★★★★★ (374) | ♥ (86)
€69.86 Add to Cart

-10% NEW RELEASE
OPSIVE
Ultimate Character Controller
★★★★★ (112) | ♥ (89)
€223.65 ~~€248.50~~ Add to Cart

RENDERHEADS LTD
AVPro Video - Ultra Edition
★★★★★ (203) | ♥ (204)
€798.40 Add to Cart

WILL HONG
Dynamic Bone
★★★★★ (638) | ♥ (14631)
€19.96 Add to Cart

RENDERHEADS LTD
AVPro Video - Core Edition
★★★★★ (201) | ♥ (100)
€399.20 Add to Cart

MORE MOUNTAINS
Feel
★★★★★ (108) | ♥ (3726)
€39.92 Add to Cart

SIRENIX
Odin Validator
★★★★★ (7) | ♥ (384)
€44.91 Add to Cart

VUPLEX
3D WebView for Windows a...
★★★★★ (193) | ♥ (525)
€149.69 Add to Cart

Verified Solution
SIRENIX
Odin Inspector and Serializer
★★★★★ (605) | ♥ (8311)
€54.89 Add to Cart

IN2SIGHT
realvirtual.io Digital Twin Pro...
★★★★★ (11) | ♥ (29)
€828.34 Add to Cart

DEMIGIANT
DOTween Pro
★★★★★ (759) | ♥ (5921)
€14.97 Add to Cart

-50% NEW RELEASE
ARTNGAME
Environment Building Bundle
(not enough ratings)
€229.54 ~~€459.08~~ Add to Cart

FIGURE 3. Unity Asset Store tool section sorted by popularity (Unity Technologies 2022i.)

For an individual or small developer, the pricing of the most popular and needed tools might be out of the range of their budget. Therefore, consideration should be given to the idea to developing the needed tools in Unity Editor by themselves which is described further in chapter 4.

4 EXTENDING THE UNITY EDITOR

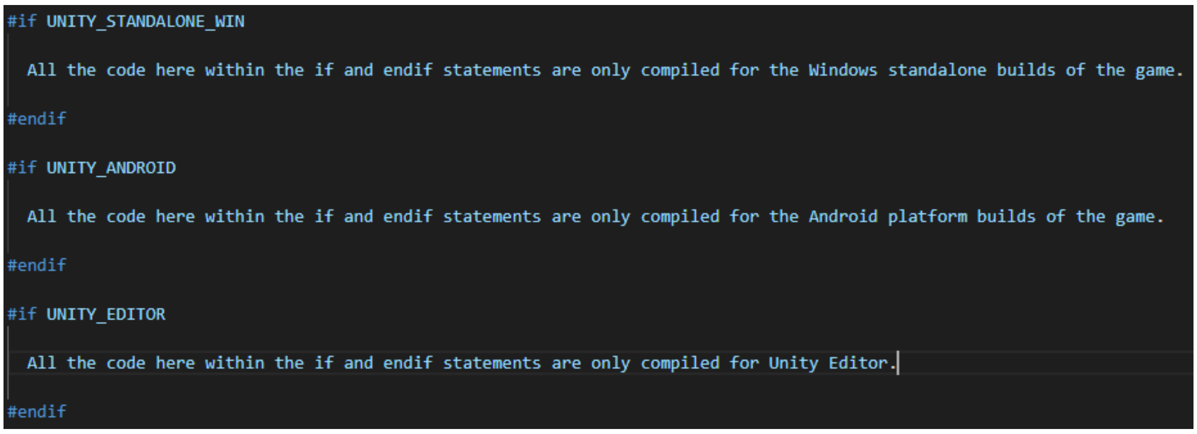
The Unity Editor can be customized by the users with the default tools that are provided with the Unity Editor out of the box. In addition to the default tools Unity offers to the developers and possible tools created by third parties, the most powerful aspect of Unity is the possibility of modifying or creating new additional tools yourself by code.

4.1 Unity script compilation

One important aspect of coding in Unity Editor is that the Unity Engine is constructed using C/C++ coding language. This means that while the primary coding language used in the Unity Editor is C#, there exists a wrapper that converts the C# code into a language that the Unity engine and the target platform devices which the game will run on understands. That is why the code libraries need to be compiled first for the game to function. Unity currently supports two different kinds of scripting backends to achieve this but details of those fall outside the scope of this thesis. (Unity Technologies 2022b.)

4.1.1 Conditional compilation

Since Unity needs to compile the C# code, as noted in chapter 4.1, it also allows the developers to use conditional compilation by utilising C# pre-processor directives. The directives can be used to specify certain parts of code that should not be compiled during the compilation process as can be seen in Figure 4. (Unity Technologies 2022c.)



```
#if UNITY_STANDALONE_WIN

  All the code here within the if and endif statements are only compiled for the Windows standalone builds of the game.

#endif

#if UNITY_ANDROID

  All the code here within the if and endif statements are only compiled for the Android platform builds of the game.

#endif

#if UNITY_EDITOR

  All the code here within the if and endif statements are only compiled for Unity Editor.

#endif
```

FIGURE 4. Conditional compilation example (Gauffin 2022a.)

The directives can be used for example to change logo images depending on the game's target platform and only execute certain functions on one or multiple platforms.

The directives are extremely useful and required when working on Unity Editor code and tool extensions. Unless explicitly defined, all the C# language code scripts, in which the Editor extensions are also made in, are included in the script compilation when the game builds are built. The UnityEditor namespace, which the Unity Editor extensions use and derive from, is one of the exceptions to this rule. Because of this, when creating the additional tools for the Unity Editor it is very easy to run into errors in code if the Unity Editor specific code is not properly isolated. Code that might work in the editor just fine might prevent the build from being completed because any references to the UnityEditor namespace and its functions are automatically excluded from the build process. One way to isolate the Unity Editor specific code is to use the UNITY_EDITOR directive as shown in the figure 4. This allows the developers to specify that this code is only to be run in the Unity Editor and not in the build versions of the game where the editor specific functionalities are not available. (Kok 2021.)

The other option to isolate the Unity Editor specific code is the editor folders which are described in chapter 4.1.2.

4.1.2 Unity assemblies and the Unity Editor assembly

In addition to the conditional compilation, Unity also utilises assembly definitions for categorizing the developer's scripts. Assembly definitions are a .NET concept which is the result of the compilation of all the scripts. In the case of Unity and C# code, the result is one or multiple .DLL files which contains a grouping of certain scripts. (Kok 2021.)

Essentially by grouping the scripts using the assemblies, it is far easier to block out certain group of scripts or create a group of scripts targeting a specific platform only. Since the scripts need to be compiled, changes to one script means that the whole assembly also needs to be recompiled which takes time. By utilising the different assemblies, groups of scripts can be made more independent from each other and to require direct referencing between the assemblies. (Unity Technologies 2022d.)

Unity also has a few different special folder names within the Unity projects, such as Resources folder and Plugins folder, which are treated differently from normal folders. One of the special folder names is the folder called Editor. Unity automatically groups any scripts in Editor folders to a separate script assembly which is then excluded from the project's build process. By default, Unity creates two different assemblies which separate the runtime scripts from editor scripts which is illustrated in Figure 5. (Tadres 2015.)

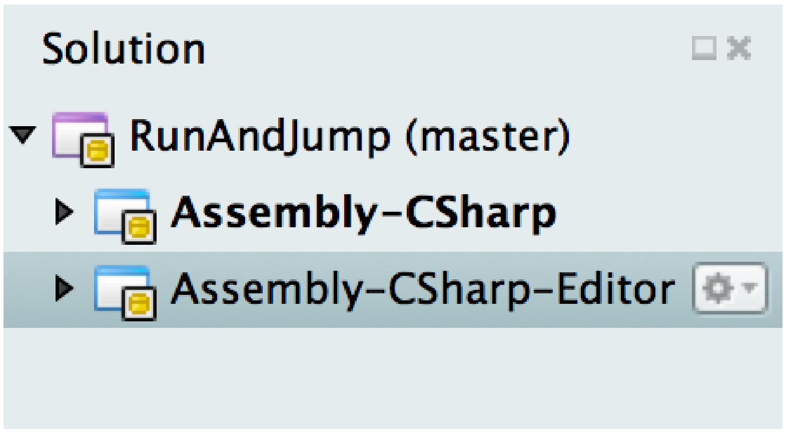


FIGURE 5. Unity default assemblies (Tadres 2015.)

When creating Unity Editor tools and code, developers should either utilise the special Editor folder name and creating the editor scripts there to group them to the Editor assembly or by using the editor conditional compilation directives to mark the editor code as such. This ensures that the editor code will not be compiled do the built versions of the game and avoid potential issues with the game build process.

4.2 Unity game data

To work on Unity Editor code and tools, there are two concepts which are important to consider, serialization and scriptable objects which are covered in this chapter.

4.2.1 Serialization

Serialization is one of the key concepts when working with the Unity Editor and developers are constantly working with it. Unity also automatically serializes certain data when scripts are created and edited inside the Unity Editor.

The main purpose of serialization is to store data in a format that allows it to easily be saved and retrieved when needed. It converts an object into a stream of bytes which allows it to be easily saved, whether to memory or files. Process called deserialization is then used to convert the stream of bytes into data which can be then used to for example restore the state of the object that was serialized. The most obvious case where developers run into serialization is when a newly created script in C# which extends from the MonoBehaviour scripting backend which Unity uses is attached to a game object in the Unity Editor. If the script contains any public variables, Unity Editor automatically serializes the fields and exposes them in the editor where the developer can directly edit the variables from the game object it has been attached to. (Tadres 2015.)

Serialization in Unity depends largely on how the data has been organized in the specific Unity project and it has a great impact on the performance of the project.

While Unity automatically serializes public variables, there are several other requirements and considerations when it comes to serialization in Unity. It is possible to serialize private or other access type variables as well with a special SerializeField attribute if the attribute is not static, constant, or read only. In addition to this, the field type also needs to be a field type that Unity can serialize such as primitive data type, enum or one of the Unity's own built-in types such as Unity Engine vectors. While Unity supports the serialization of lists of these serializable types, Unity does not support the serialization of multilevel types such as dictionaries or multidimensional arrays which are regularly used in Unity projects. Unity also normally does not serialize C# properties automatically but allows the developers to serialize a backing field for the property manually if it is needed. (Unity Technologies 2022e.)

Because of these restrictions to the types, Unity does support the serialization of custom classes that do not extend from the MonoBehaviour Unity class. This is extremely useful, and a lot of the serialization restrictions can be overcome by creating of different kinds of custom class wrappers to store for example game save data or other types of data that might not be directly serializable. Unity itself uses serialization when saving and loading the projects scenes, assets, and asset bundles into the device's memory that the project is running on. (Unity Technologies 2022e.)

The concept of serialization is important to keep in mind when creating additional Unity Editor tools. The Editor class in Unity Editor for example utilises a serialized object which administrates the data editing for Unity objects. It aids in supporting the undo operations in the Unity Editor and assists with Graphical User Interface drawing (GUI) which is a code driven way to create custom GUI within the Unity Editor. (Kok 2021.)

4.2.2 Scriptable objects

Scriptable objects in Unity are a specific kind of specialized object that is used as a container to store data. Unlike the C# scripts which extend from the MonoBehaviour Unity object, the scriptable objects do not need to be attach to any specific game object in the game scene. This means that they are treated more like assets in the project, like any graphic or other asset the project uses. While it is possible to store the game data in different formats, such as XML, JSON or .txt files, and read the data from them, storing the same data to scriptable objects eases the burden of creating separate parsers or relying on third-party tools to read the data correctly from these other file types. Unity can automatically serialize and deserialize the scriptable objects so access to the serialized values is much more straight forward. (Tadres 2015.)

In addition, since the scriptable objects do not need to be attached to a game object, the same scriptable object can be referenced in multiple different game scenes which lowers the overhead compared to having the same data included in all the different scenes that require it. Creating game objects with MonoBehaviour scripts attached to them creates a separate class instance for each of the objects which multiplies the data that needs to be stored in memory compared to a scriptable object which only saves the data once and which can be then reference by multiple different objects. For consideration regarding additional tool development for the Unity Editor, the scriptable objects can also be run in both during the game runtime in the play mode or just in editor unlike the scripts attached to the game objects. (Kok 2021.)

The reason the scriptable objects can save data in the Unity Editor is that it uses the Unity Editor scripting and namespace which allows it to access the Unity Editor specific functionalities. Therefore, they can be used to save and store data during a Unity Editor session but not during runtime in the deployed builds of the project. The saved scriptable object data can of course be utilised during the game builds, but any changes made to them during runtime are not saved. Since it is an asset in the project, the data can be saved just like any other asset to the disk, so it stays the same between Editor sessions. (Unity Technologies 2022f.)

4.3 Customizing the Unity Editor

The Unity Editor can be customized in multiple different ways. Unity offers some built in custom property attributes commands that can be for example added to the serialized fields in MonoBehaviour scripts which show in the inspector. These commands can be used to add additional functionality to these fields such as adding a range limit to numerical values or slider handles to quickly change the values.

The powerful aspect of the Unity Editor is the possibility to redesign the default inspector views and create new ways to display the information. This support for customization also allows the developers to create completely new windows, tools and to customize the scene view with additional functionalities which are described in this chapter.

4.3.1 Execute MonoBehaviour script in edit mode

One of the ways that might assist developers in their project, that does not directly customize the editor by utilising the editor tools, is the possibility to execute scripts also in Unity Editor's edit mode. This allows scripts attached to MonoBehaviours that would normally require the game scene to be in a play mode to be executed in also in the edit mode. (Unity Technologies 2022g.)

This can be helpful seeing the impact of certain type of scripts already in the editing phase since launching the game in play mode takes time and resources.

4.3.2 Custom editors

When you create a new C# script in a Unity project, the newly created script automatically extends and inherits from the MonoBehaviour class. The MonoBehaviour inherited components must be attached to a game object in the Unity scene for in order them to function. When the C# scripts that are attached to a game object, the Unity Editor automatically displays the script's serialized fields in the inspector window as can be seen in Figure 6 where a public Unity construct Vector3 with three different float values has been serialized to be shown in the inspector view. (Unity Technologies 2022g.)

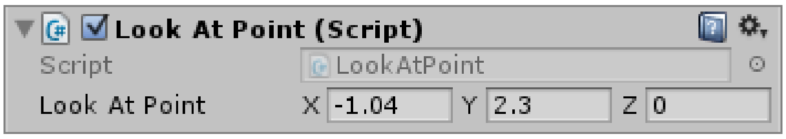

Look At Point (Script)
Script LookAtPoint
Look At Point X -1.04 Y 2.3 Z 0

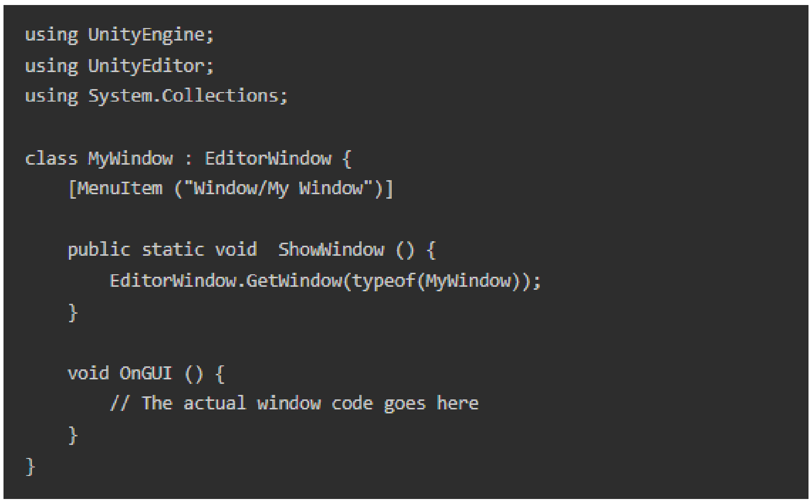
FIGURE 6. A default Inspector with a public Vector3 field (Unity Technologies 2022g.)

The inspector default view in Figure 6 is however only the default window layout which can be overridden. While the original actual LookAtPoint script extended from the MonoBehaviour class, the developers can also create a separate script that instead extends the Unity Editor class. This editor script can then be configured to point that this script is a custom editor for the LookAtPoint script and allow the user to modify how the LookAtPoint script appears in the Unity inspector window by overriding the Editor class's default OnInspectorGUI method with a custom solution. Although it might not be apparent to especially new developers starting with Unity, all the Unity's different editor views and windows, even the complex looking ones, utilise this same unity Editor extending to achieve the different kind of editor views. (Unity Technologies 2022g.)

4.3.3 Unity Editor windows

While the Custom Editors described in chapter 4.3.2 can be used to either customize or completely override the existing inspector window in the Unity Editor. it is also possible to create new separate windows into the Unity Editor as well. These windows can be switched around and docked into the Unity Editor layout like any other editor window that are there by default. (Kok 2021.)

Whereas the C# scripts attached to the game objects, which extended and inherited from the MonoBehaviour class, and the custom editors which extended from the Editor class, new editor windows are created by extending and inheriting the Unity EditorWindow class. As illustrated in Figure 7, Utilising the UnityEditor namespace and extending the EditorWindow through inheritance, the scripts gain access to the Editor specific methods, most importantly the OnGUI method, which can then be utilised to display the information in the new window as the developers wish. Using the MenuItem property attribute also adds a new selection to the Unity Editor menu to open this new window. The ShowWindow method on the other hand ensures that the same window will be opened so that multiple ones are not created needlessly. (Unity Technologies 2022j.)



```
using UnityEngine;
using UnityEditor;
using System.Collections;

class MyWindow : EditorWindow {
    [MenuItem ("Window/My Window")]

    public static void  ShowWindow () {
        EditorWindow.GetWindow(typeof(MyWindow));
    }

    void OnGUI () {
        // The actual window code goes here
    }
}
```

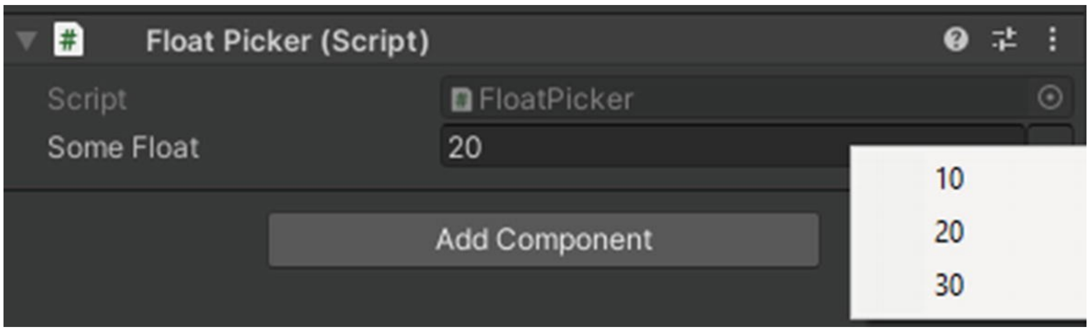
FIGURE 7. Unity Editor Window basic code example (Unity Technologies 2022j.)

4.3.4 Unity Editor property drawers

By default, Unity Editor offers certain built-in property attributes that can be attached to serialized fields to change how they appear in the inspector or for example add a new menu option as described in chapter 4.3.3. Primarily, the property attributes are a fast and easy way to customize the Unity Editor's inspector view as it is only altering and customizing the already existing built-in inspector view as opposed to creating a completely new custom inspector from scratch. There are several other built-in property attributes that can be used for the different serialized variable types to change how they appear in the editor's inspector view. (Tadres 2015.)

In addition to the built-in property attributes, the Unity Editor allows developers to create their own custom properties and supports the ability to specify how the Unity Editor draws those custom properties. (Unity Technologies 2022k.)

To create a custom property attribute, the developer needs to create a separate C# script that extend and inherits from the PropertyAttribute class in Unity Editor. By creating a separate script this way, it will automatically function similarly to the Unity's own property attributes where you can utilise it in similar way as the built-in properties such as the MenuItem attribute. Then another script can be created that extends the PopertyDrawer class which allows the developer to define how this custom property is shown and handled in the Unity Editor inspector. This can for example be used to create a custom property drawer for a float variable, where using the custom property attribute above the float variable field allows the developer to specify a list of float values which can be assigned to that field. Building a custom property drawer for this property attribute could then allow the user to select the wanted float value from the list of float values that have been created for that float variable. This allows developers to quickly shift and select between predetermined float values for the field as illustrated in Figure 8. (Kok 2021.)



Script FloatPicker
Some Float 20

FIGURE 8. Custom Unity property attribute and property drawer example (Kok 2021.)

4.3.5 Unity Editor scene view additions

The Unity Editor scene view is the representation of the actual game objects and what the player sees when playing the game. While the scene view with all the game objects and scripts attached to them provides the developers an easy way to switch the positions or rotations of game objects and getting a good overview, it does not necessarily show all the things that happen on the code side until the code is being executed in the game play mode. For example, a waypoint system used by a game object to move from one position to another may only exist in code and this is not visualized in anyway by default. Unity Editor however does allow the developers to modify and make the scene view more robust either by utilising the gizmos functionality or by creating their own custom editor tools for the scene view. (Tadres 2015.)

The gizmos in the Unity Editor are used to display a graphical view of the game objects in the Unity Editor scene view. By default, some of these gizmos are always shown whereas other are normally hidden when the specific game object is not selected. Some of them are just informative, indicating only additional information while others are also interactable such as the move, scale and transform tools. The most seen gizmos by starting developers using the Unity Editor are the camera and light direction gizmos that are shown in Figure 9. While the light source gizmo indicates the direction of the lighting, the camera gizmo can be used to change the camera's view using the gizmo handles. (Unity Technologies 2022l.)

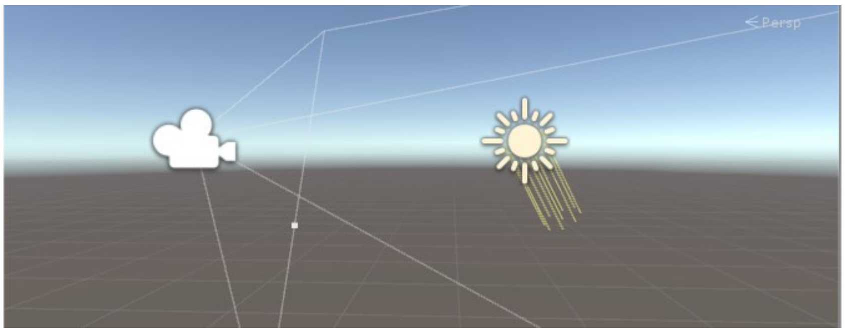


FIGURE 9. Camera and light direction scene view gizmo example (Unity Technologies 2022l.)

While the default gizmos can be displayed and changed in the Unity Editor quite easily by just selecting them and changing the built-in options for the gizmos, the gizmos offer even more flexibility by allowing them to be created and modified through code. Since the C# scripts attached to the game objects in the game scene view are extended and inherited from the MonoBehaviour class, they automatically have access to the same gizmo methods that are used to draw the default gizmo implementations. This allows developers to also create their own implementation in code for the component's OnDrawGizmos and OnDrawGizmosSelected methods. Implementing these methods allows the developers for

example to draw icons to locations in the scene view or draw lines or other graphical representation between coordinates to visualize things more clearly in the scene view. It is also possible to create the gizmo related functionality as a separate script which allows even more customization since they are not directly related to the component they are attached to. (Tadres 2015.)
In addition to drawing and customizing the gizmos in the Unity Editor, developers also have the possibility of creating separate scripts that extend and inherit from the EditorTool class which allows the creation of separate tools for the scene view. Once created, these tools will be accessible from the same toolbar as all the default Unity Editor scene tools like in the Kok (2021) example of a customized object spawner tool shown in Figure 10. (Kok 2021.)

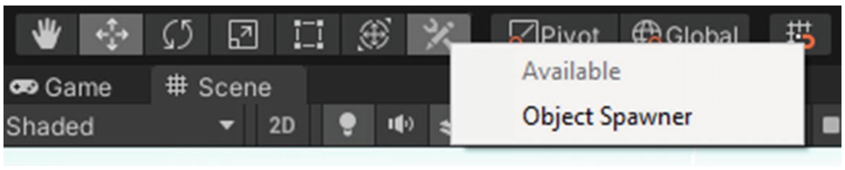


FIGURE 10. Example of a customized object spawner editor tool in the scene view (Kok 2021.)

Both the gizmos and the actual new editor tools can also both take advantage of the Unity's Handles class, which on top of the classes which allow additional graphical information to be displayed in the scene view, enables the creation of the same kind of handles that Unity Editor has built-in by default. Developers can for example create similar handles that Unity's move tool does which enables the dragging of a game object along the different axis' by just clicking on the handles and dragging the object in the scene view. (Unity Technologies 2022m.)

5 CASE AURORA PIRATES

5.1 Chosen technologies and the case project

The case example project Aurora Pirates is developed using the Unity Editor version 2021.3.12f1 which is one of the latest long term support (LTS) versions of the Unity Editor. The long term support version was chosen due to the stability they offer. The editor tools were created using the primary Unity Editor scripting language C# and the chosen code editor was VS Code due to the familiarity of the thesis author with the VS Code as well as it being free to use. The VS Code can also be extended with plugins for Unity and other C# related coding conventions easily. The GitHub version control and GitHub desktop client were utilised for the version control of the project.

The project case Aurora Pirates is a game concept that the author of the thesis has had in mind for a while, and which could benefit greatly from the usage of additional tools to make the project more easily manageable and realizable. The project is in its early stages where the benefits of the created tools will carry on all through the development cycle.

Aurora Pirates is a blend of action-adventure and strategy genres with an open world like its inspiration, Sid Meier’s Pirates. The game is going rely heavily on different systems within the game world that simulate different factions vying for the control of the galaxy by battling each other and trying to grow their influence. The player can choose to join the different factions or make their own mark by becoming the most sought-after pirate in the game.

The main challenge in creating the project is to try and balance all the different factions and resources in the game in a way that allows the player to progress but at the same time simulate a balanced war between the factions. The Unity Editor on its own does allow the creation of data containers and systems without much of an issue. The big issue with the project is the ability to view all this data since all the individual pieces affect each other and the game balance and often this information might be hidden inside the individual code scripts. Therefore, the chosen editor tools to be implemented for the project focus largely on how this amount of different game balancing data can easily be displayed and manipulated.

5.2 Implemented editor tools for the project

5.2.1 Property drawer for easier viewing and changing of enum values

The property drawers which were described in chapter 4.3.4 seem like an easy way to customize the inspector view to show the game object's serialized variables in a desired way. Enums are one of the frequently used types in game programming where they are used to represent different statuses and types in a more readable form instead of their underlying numerical integer values.

By default, Unity Editor shows the serialized enum variables as a drop-down list which allows the developer to choose the wanted enum from the list. When the amount of enum variables on a game object starts to grow, reading all the individual variables from the fields and selecting the correct ones from the list does consume time and mistakes are harder to spot. Therefore, there was a need to create a better way to faster identify and select the correct enum for the loot table variables which are in the game and indicate the amount and type of loot the player and the non-playable characters can gain from the target. The loot table script was created as a separate custom serializable class containing a public loot type enum variable and a public float variable for the amount. The custom serializable class allows the creation of a custom property drawer for this class. The default Unity implementation of the loot table class in the Unity Editor inspector can be seen in Figure 11 where the loot table class has been added to the asteroid game object which contains a serialized list of the loot table class.

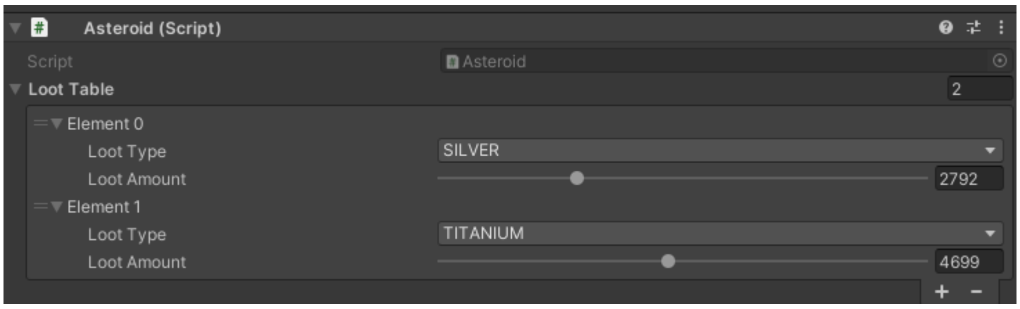

Asteroid (Script)
Script: Asteroid
Loot Table: 2
Element 0
Loot Type: SILVER
Loot Amount: 2792
Element 1
Loot Type: TITANIUM
Loot Amount: 4699

FIGURE 11. Default Unity inspector view of the loot table class (Gauffin. H. 2022b.)

By creating a custom property drawer for the loot table class through an editor script, the default Unity inspector view of this class can be changed to a toggle instead, which automatically allows the developer to see the possible enum values for the field and due to their uniform layout, can distinguish fast which of the values are selected as can be seen in Figure 12. The custom property drawer also allows to removal of the element names and the nested list which reduces the amount of space needed for the value fields.

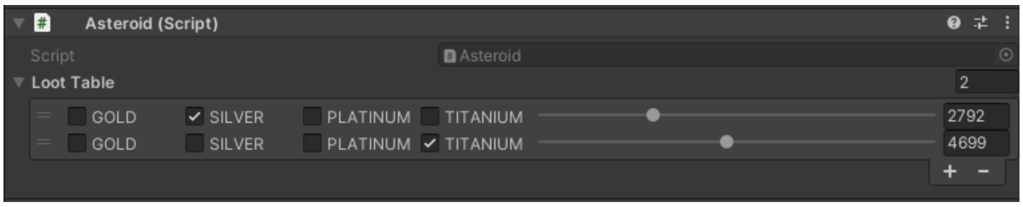
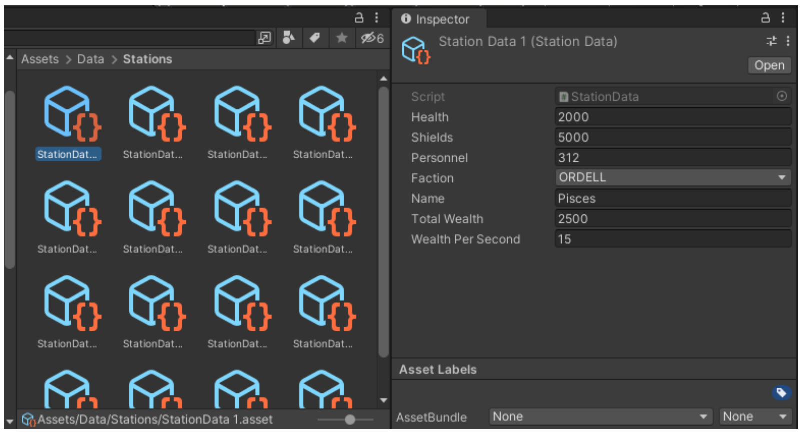

Asteroid (Script)
Script: Asteroid
Loot Table: 2
GOLD [✓ SILVER] PLATINUM TITANIUM 2792
GOLD SILVER PLATINUM [✓ TITANIUM] 4699

FIGURE 12. Custom property drawer for the loot table class (Gauffin. H. 2022c.)

The biggest hurdle when creating the custom property drawer is calculating and choosing the different sizes for the fields that the developer wants to draw to the inspector using the EditorGUI methods. The created tool works for this need very well, but it might require further changes and configuration if the enum value count raises and changing the inspector width also causes issues with the current implementation. The implementation can however still be utilised in other custom serializable classes that require the same kind of functionality in displaying the information in the inspector view.

5.2.2 Editor window to view and edit data from multiple scriptable objects

The scriptable objects serve as a great container for storing data in Unity. In the project, the data for the space stations which exist in the game world and interact with the player and other units was created as a scriptable object. One issue with the Unity default editor is that the inspector displays the information from one scriptable object at a time as can be seen in Figure 13 which displays the default Unity inspector view of one of the station data scriptable objects.



Station Data 1 (Station Data)

Script	StationData
Health	2000
Shields	5000
Personnel	312
Faction	ORDELL
Name	Pisces
Total Wealth	2500
Wealth Per Second	15

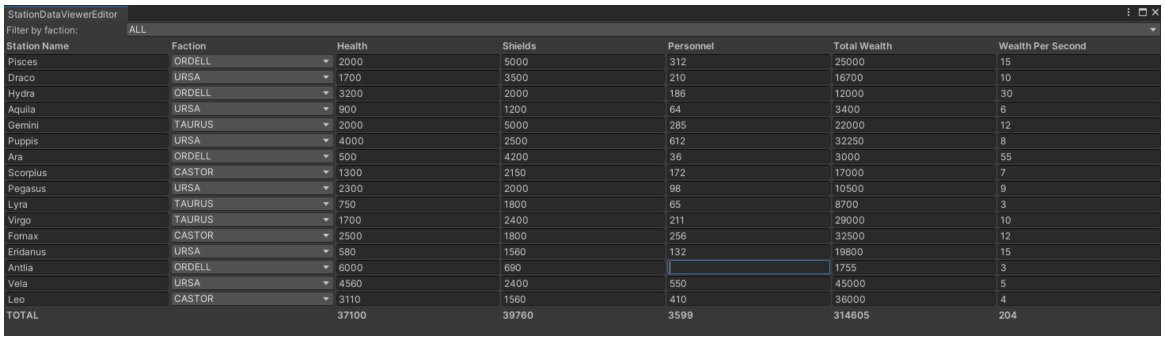
AssetBundle None None

Assets/Data/Stations/StationData 1.asset

FIGURE 13. Unity default inspector view of a project's scriptable object (Gauffin. H. 2022d.)

The project has a huge number of different stations for different factions which means keeping track of the whole is rather hard and changing values for the scriptable objects need to be done by one scriptable object at a time. Therefore, there was a need to create a tool to view multiple scriptable objects at the same time and allow the editing of the scriptable object values.

Since this required the access to multiple scriptable objects at a time, a new editor window script was created for this purpose. The UnityEditor namespace contains a separate class called AssetDatabase that can be used to search all the project's assets to locate every station data scriptable object regardless of their folder location within the project assets folder. Then the list can be drawn in a station data viewer script which extends and inherits from the EditorWindow class. The class was used to create a new editor window which draws the scriptable object's data to the window utilising the OnGUI method and connecting the scriptable object's variables to the drawn fields in the window. Unlike the custom property drawer, the station data window utilises the methods contained in the EditorGUILayout class instead of EditorGUI and this allows the creation of the horizontal and vertical rows of the window much more automatically by using the built-in layout options. The custom editor window for viewing and editing the station data can be seen in Figure 14 which in the current implementation allows for viewing and editing all the station data or filtering based on selected faction.


StationDataViewerEditor

Filter by faction: ALL

Station Name	Faction	Health	Shields	Personnel	Total Wealth	Wealth Per Second
Pisces	ORDELL	2000	5000	312	25000	15
Draco	URSA	1700	3500	210	16700	10
Hydra	ORDELL	3200	2000	186	12000	30
Aquila	URSA	900	1200	64	3400	6
Gemini	TAURUS	2000	5000	285	22000	12
Puppis	URSA	4000	2500	612	32250	8
Ara	ORDELL	500	4200	36	3000	55
Scorpius	CASTOR	1300	2150	172	17000	7
Pegasus	URSA	2300	2000	98	10500	9
Lyra	TAURUS	750	1800	65	8700	3
Virgo	TAURUS	1700	2400	211	29000	10
Fornax	CASTOR	2500	1800	256	32500	12
Eridanus	URSA	580	1560	132	19800	15
Antlia	ORDELL	6000	690		1755	3
Vela	URSA	4560	2400	550	45000	5
Leo	CASTOR	3110	1560	410	36000	4
TOTAL		37100	39760	3599	314605	204

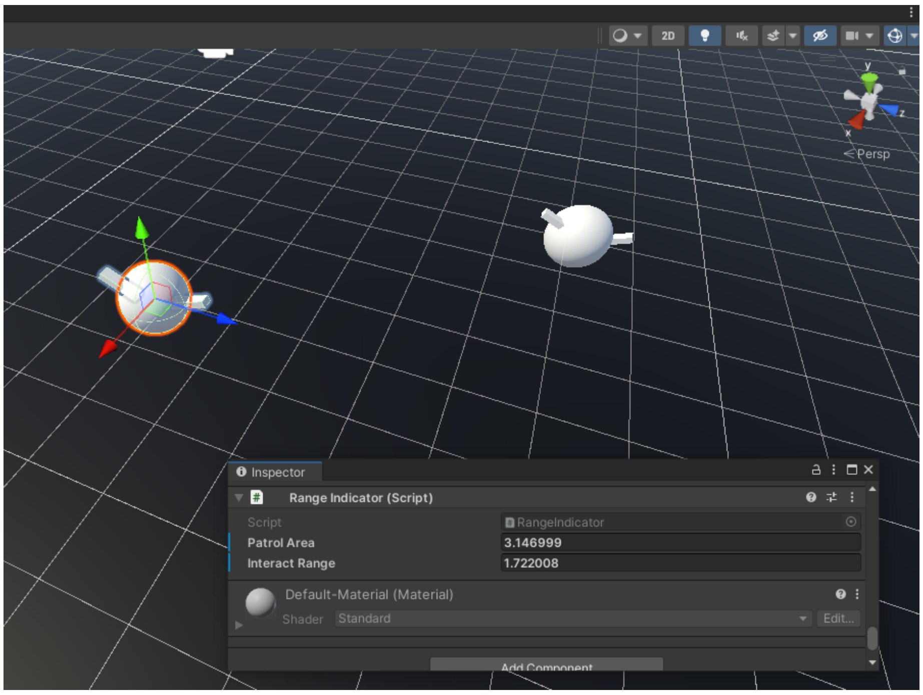
FIGURE 14. Custom editor window to view multiple project’s scriptable objects (Gauffin. H. 2022e.)

The editor window script also converts the scriptable objects to serialized objects when they are drawn as it allows any changes made to the values to be automatically saved to the scriptable object asset and changes can be reverted with the Unity Editor’s own undo option. Other possible implementation option could be to not automatically save changes but handle the saving by a separate button. The tool could also be modified to support for creating new station data scriptable objects directly from the tool.

Alternative to creating this editor window tool would be to read the data from a text file or external spreadsheet application to the scriptable objects but this would also require the creation of additional parsers and keeping the data editing inside Unity allows for further customized drawing of the data and improvement of the tool. The basic implementation of the tool can be used to find and draw information from other assets than scriptable object as well. A modified version of the tool could for example find and display all the 2D art assets within the project and display relevant information regarding them.

5.2.3 Scene view additions to illustrate ranges

One challenge when trying to balance and fine tune the in-game values for a project such as the Aurora Pirates is that by default the different values contained in the game objects are not illustrated in the editor by default. For example, the values that determine the ranges for the station's patrolling area and interact range is hard to gauge and change just from the default editor view which can be seen in Figure 15.



Range Indicator (Script)

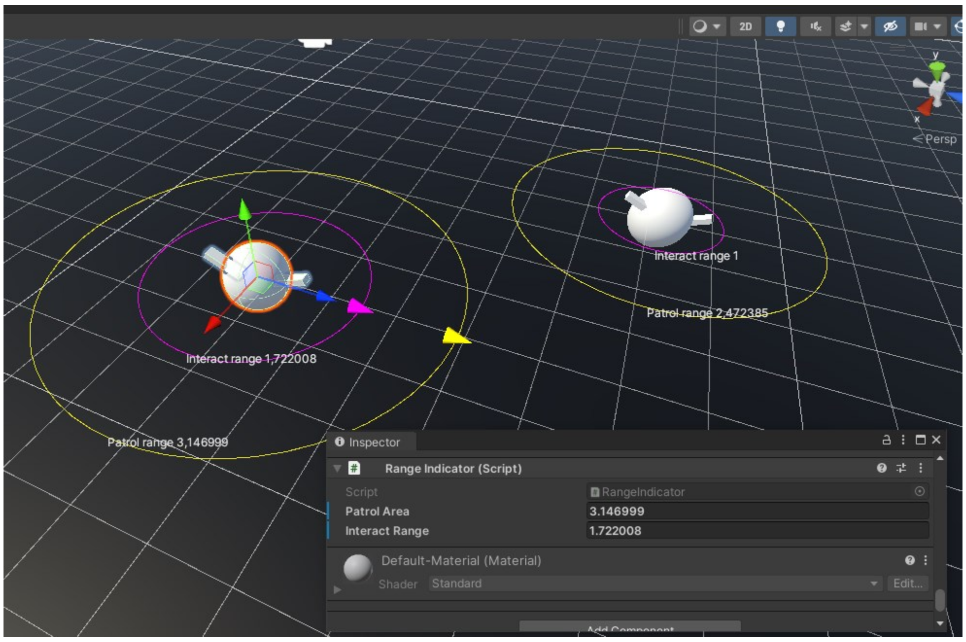
Script	RangeIndicator
Patrol Area	3.146999
Interact Range	1.722008

Default-Material (Material)

Shader Standard

FIGURE 15. Unity default view of station patrol and interact range (Gauffin. H. 2022f.)

To illustrate and change these ranges better, a custom addition to the scene view was needed. This was achieved by creating a custom editor for the range indicator script that would draw these ranges in the 2D plane that the game functions in and a handle to allow the values to be changed similarly to the default move object handles. The result of the custom editor for the range indicator for the scene view illustrates these ranges with wire discs, value labels and allows the selected object's values to be changed using the corresponding handles as can be seen in Figure 16.



Range Indicator (Script)

Script	RangeIndicator
Patrol Area	3.146999
Interact Range	1.722008

Default-Material (Material)

Shader Standard Edit...

FIGURE 16. Unity custom editor view of station patrol and interact range (Gauffin. H. 2022g.)

The handles and the wire discs to represent the ranges were first drawn to the scene via the custom editor script by utilising the Editor class's OnSceneGUI method which can be used to draw additions to the scene view. There was however an issue as the Editor and Handles classes by default only draws the relevant information to the scene when the object is selected in the scene view and the need was to illustrate these values for all the range indicator scripts at the same time.

To achieve the drawing of these ranges even when the object is not selected, the use of the Gizmos was considered since they are useful for drawing the graphical representation for either the one selected object or for all the objects. However, since the functionality had already been created for the custom editor script using the Handles class and the Gizmos do not offer similar fast wire disc drawing out of the box, it was decided to create an additional MonoBehaviour script attached to a single game object in the scene. This additional script finds all the range indicators in the scene and draws the wire disc representation in that object's OnDrawGizmos method using the Handles class. Since this is a script in the game scene and utilises the Editor class, it is necessary to use the conditional compilation directive to only run the code in the Editor or otherwise the game cannot build correctly since the editor tools are not available there. The actual handle drawing and value change was still left to the range indicator custom editor script when the object is selected.

5.3 Key takeaways

5.3.1 Key takeaways of the created tools

While the tools created for the project are rather small, they provide great value for the data visibility and management of the project going forward. This shows that even small changes and increase in productivity early on can bring great accumulated benefits over the course of a game project development cycle. The development took time and required several iterations before the wanted functionality was achieved but the time spent on them was not excessive. The experience of understanding the editor tool creation allows for much faster tool creation in the future and can constantly give new ideas how certain things could be done better.

Based on the tool creation, it might have been easier to create a completely new custom inspector view for the serializable class that was used for the loot table changes. The custom property drawer that utilises the EditorGUI methods was much harder to use due to having to calculate the different field heights and widths manually rather than the EditorGUILayout which contained the pre-built layout options for laying out the different fields for the scriptable object custom editor window.

The tools can be further expanded upon and utilised in other parts of the project or other future projects. The next steps would also include making the implementation of the tools more generic to support other use cases and projects as well, as the current implementation is heavily tied to the specific problems that were solved in this project.

5.3.2 Key takeaways of additional tool creation in general

Based on the research and the case project, several recommendations can be made regarding additional tool creation for the Unity Editor. The first thing to consider and weigh when considering the creation or acquisition of tools for a game project is to consider whether the tool is needed. Whether the tool is created by the developers themselves or acquired from somewhere else, they require development and integration time on top of possible monetary costs. In case of for example automating certain functions through a tool, there is no incentive to create the tool if the cost is greater than manually doing those functions. Therefore, it is good to consider how often and for what purpose the tool is going to be used and whether it can be repurposed for other things or future projects which increases the value of the tool.

Especially for new starting developers, creating a game project requires a lot of knowledge of different domains and the tool creation aspect might not be of the highest priority when learning about game development. In the end the players of the completed game project are likely to not ever know how much time, time that could potentially have been used to create new or enhance existing gameplay systems that the players interact with.

What also needs to be considered is if the required tool could be acquired from an outside source. There exist several free and paid tools in the different marketplaces for the game engines that can be utilised by developers in their game projects. This however also brings certain challenges when trying to integrate the tool to work on a particular project. More time might be spent on fixing issues with an acquired tool than how long it would have taken if the tool was created by the developers themselves. In addition, support for the tool might not always be available or documentation may be lacking if the tool is acquired from somewhere else.

Even with these concerns and considerations, additional tools clearly offer value for both the tool creators and their users. The correct tools can speed up the development process and reduce errors in game projects. The creators of these tools can tap into additional revenue source by publishing their tools for others to use and can learn from user feedback how to improve them. The number of tools available in the Unity marketplace and the amount of developer’s who use them is a testament to how beneficial additional tools can be. The creation of additional tools for the Unity game engine requires time investment but it is well worth it.

6 CONCLUSIONS

Through the research and example case project of additional tool creation for the Unity game engine, the result of the thesis provides the commissioner of the thesis and other stakeholders reasons and recommendations on why and how the additional tools can assist developers in game projects and game production education.

The additional tools can help greatly in game projects if the need and resource concerns are taken properly into account. Depending on the situation, it might be better to forego the tool creation or look for accessible third-party options, but this requires careful consideration, and it is not without its own challenges. The focus on additional tool creation for the Unity game engine should not be the first things a new starting developers concentrate on but should still be kept in mind as they gain experience with the Unity game engine due to its benefits.

As a result of the thesis, several smaller additional tools were created for the case project which can be utilised in other game projects or in the commissioner's game production education. They can also give ideas for further development of said tools and in which other cases they might be applicable. The thesis also served as a basis for a separate document which was compiled and submitted to the commissioner of the thesis regarding the additional tool creation for Unity game engine.

In retrospect, the better defining and narrowing of the thesis to only a single bigger tool or knowledge area could have been better as the time concerns prevented the execution of all the wanted features and proper analytical comparison to third party tool usage in practice. This could have led to a better structuring of the thesis. Still, the thesis provided the author much needed and valuable personal experience on the subject matter and provided the commissioner recommendations and guidance according to the objective.

.

REFERENCES

Arendt, S. 2008. Showdown: Developers Argue Pros And Cons Of Middleware. Wired. Accessed 9 November 2022.
https://www.wired.com/2008/02/las-vegas-1/

Barclays. 2021. Funding and Finance: Making the most of middleware. Barclays Bank UK PLC. Accessed 9 November 2022.
https://games.barclays/resource-hub/games/funding-and-finance/making-the-most-of-middleware/

Blackman, S. 2014. Unity for Absolute Beginners. Apress. Accessed: 31 October 2022. Requires access.
https://learning.oreilly.com/library/view/unity-for-absolute/9781430267782/

Blow, J. 2004. Game Development: Harder Than You Think. ACM Digital Library. Accessed: 7 November 2022.
https://doi.org/10.1145/971564.971590

Gauffin, H. 2022a. Conditional compilation example. Thesis author.

Gauffin, H. 2022b. Default Unity inspector view of the loot table class. Thesis author.

Gauffin, H. 2022c. Custom property drawer for the loot table class. Thesis author.

Gauffin, H. 2022d. Unity default inspector view of project's scriptable object. Thesis author.

Gauffin, H. 2022e. Custom editor window to view multiple project's scriptable objects. Thesis author.

Gauffin, H. 2022f. Unity default view of station patrol and interact range
Thesis author.

Gauffin, H. 2022g. Unity custom editor view of station patrol and interact range. Thesis author.

Gregory, J. 2018. Game Engine Architecture, Third Edition, 3rd Edition. A K Peters/CRC Press. Accessed: 7 November 2022. Requires access:
https://learning.oreilly.com/library/view/game-engine-architecture/9781351974271/

Hougaard, K. 2013. Funding Indie Games with the Asset Store. Unity Technologies. Accessed 11 November 2022.
https://blog.unity.com/games/funding-indie-games-asset-store

Kok, B. 2021. Beginning Unity Editor Scripting: Create and Publish Your Game Tools. E-book. 1st ed. Hong Kong, China: Apress. Accessed: 9 October 2022. Requires access.
https://learning.oreilly.com/library/view/beginning-unity-editor/9781484271674/

Martin, J. 2020. Game Design Tips and Trends: What is a game Engine? University of Silicon Valley. Accessed 7 November 2022.
https://usv.edu/blog/what-is-a-game-engine/

Tadres, A. 2015. Extending Unity with Editor Scripting: Put Unity to Use for Your Video Games by Creating Your Own Custom Tools with Editor Scripting. [N.p.]: Packt Publishing (Community Experience Distilled). Accessed: 2 October 2022. Requires access.
https://search-ebscohost-com.libproxy.tuni.fi/login.aspx?direct=true&AuthType=cookie,ip,uid&db=e000xww&AN=1069146&site=ehost-live&scope=site

Unity Technologies. 2022a. Learn Unity: Explore the Unity Editor. Accessed: 31 October 2022.
https://learn.unity.com/tutorial/explore-the-unity-editor-1?uv=2021.3

Unity Technologies. 2022b. Manual: Unity Architecture. Accessed: 2 November 2022.
https://docs.unity3d.com/Manual/unity-architecture.html

Unity Technologies. 2022c. Manual: Conditional compilation. Accessed: 2 November 2022.
https://docs.unity3d.com/Manual/unity-architecture.html

Unity Technologies. 2022d. Manual: Assembly Definition Files. Accessed: 2 November 2022.
https://docs.unity3d.com/Manual/ScriptCompilationAssemblyDefinitionFiles.html

Unity Technologies. 2022e. Manual: Script Serialization. Accessed: 3 November 2022.
https://docs.unity3d.com/Manual/script-Serialization.html

Unity Technologies. 2022f. Manual: Scriptable Object. Accessed: 3 November 2022.
https://docs.unity3d.com/Manual/class-ScriptableObject.html

Unity Technologies. 2022g. Manual: Custom Editors. Accessed: 4 November 2022.
https://docs.unity3d.com/Manual/editor-CustomEditors.html

Unity Technologies. 2022h. Unity Asset Store: Start publishing on the Asset Store. Accessed: 11 November 2022.
https://assetstore.unity.com/publishing/publish-and-sell-assets

Unity Technologies. 2022i. Unity Asset Store: Tools. Accessed 11 November 2022.
https://assetstore.unity.com/tools

Unity Technologies. 2022j. Manual: Editor Windows. Accessed 14 November 2022.
https://docs.unity3d.com/Manual/editor-EditorWindows.html

Unity Technologies. 2022k. Manual: Property Drawers. Accessed 14 November 2022.
https://docs.unity3d.com/Manual/editor-PropertyDrawers.html

Unity Technologies. 2022l. Manual: Gizmos Menu. Accessed 15 November 2022.
https://docs.unity3d.com/Manual/GizmosMenu.html

Unity Technologies. 2022m. Manual: Important Classes - Gizmos & Handles. Accessed 16 November 2022.
https://docs.unity3d.com/Manual/GizmosAndHandles.html

Williams, A. 2017. History of Digital Games. Routledge. Accessed: 7 November 2022. Requires access.
https://learning.oreilly.com/library/view/history-of-digital/9781317503804/