Bachelor's thesis

Information and Communications Technology

2024

Jaakko Haavisto

Designing implementable tools

– the importance of easy and fast implementation in the metaverse era

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

Bachelor’s Thesis | Abstract

Turku University of Applied Sciences

Information and Communications Technology

March 2024 | 34 Pages

Jaakko Haavisto

Designing implementable tools

- the importance of easy and fast implementation in the metaverse era

The aim of this thesis was to investigate existing approaches to developer tool design and the impact these approaches have on the ease and speed of implementation.

The commissioner of this thesis, Futuristic Interactive Technologies research group at Turku University of Applied Sciences, requested that a Unity Engine tool for use in future metaverse platform development was developed as part of this thesis. In relation to this, this thesis explores the importance of properly designed developer tools in the realm of metaverse platform development and the viability of Unity Engine as a platform for metaverse platform development in comparison to Unreal Engine and Godot.

Existing material on tool design was researched and used to guide the design during the tool's development. Design principles derived from the material were then used to conduct a small study on the impact of specific design elements on the implementation of a tool.

The results of the study support the hypothesis that developers find consciously designed tools easier and faster to implement, though further studies are needed for more concrete and conclusive results.

Keywords:

virtual reality, Unity Engine, software development, software design, metaverse

Opinnäytetyö (AMK) | Tiivistelmä

Turun ammattikorkeakoulu

Tieto- ja viestintätekniikka

2024 | 34 sivua

Jaakko Haavisto

Implementoitavien työkalujen suunnittelu

- nopean ja helpon implementaation tärkeys metaverse-ajalla

Tämän opinnäytetyön tavoitteena oli tutkia olemassa olevia kehittäjätyökalujen suunnittelutapoja ja näiden vaikutusta työkalun implementoinnin nopeuteen ja helppouteen.

Opinnäytetyön tavoitteena oli myös kehittää työkalu Unity Engine -pelimoottorille, joka tulisi käyttöön tulevissa metaverse-alusta projekteissa. Tästä syystä opinnäytetyössä tutkittiin työkalujen suunnittelun vaikutusta metaverse-alustojen kehityksessä. Opinnäytetyössä selvitettiin myös Unity -pelimoottorin sopivuutta metaverse-alustojen kehittämiseen suhteessa Unreal Engine ja Godot -pelimoottoreihin.

Työkalun suunnittelussa ja kehityksessä käytettiin hyväksi kirjallisuuslähteistä löydettyjä suunnittelumalleja. Näiden suunnittelumallien tehokkuutta tutkittiin antamalla ryhmälle insinöörejä kysely suunnitteluelementtien vaikutuksesta työkalun implementaatioon.

Tutkimuksen tulokset kannattavat hypoteesia, että kehittäjien näkökulmasta implementaatioon keskittyen suunnitellut työkalut ovat merkittävästi helpompia ja nopeampia implementoida, mutta lisätutkimuksia vaaditaan tarkempien ja varmempien tulosten saamiseksi.

Asiasanat:

virtuaalitodellisuus, Unity, ohjelmistotuotanto, ohjelmistosuunnittelu, metaverse

Contents

Images

Tables

List of Abbreviations and Terms

2D	Two-Dimensional
C++	A programming language
C#	A programming language
DOTS	Data-Oriented Technology Stack
FIT	Futuristic Interactive Technologies
GDScript	A programming language
GUI	Graphical User Interface
XR	Extended Reality, catch-all for Augmented Reality, Virtual Reality and Mixed Reality

1 Introduction

Maintainability and ease of implementation are no doubt positive attributes for any piece of software, but they become increasingly important when the aim is to create tools that are to be functioning pieces of a much larger, unknown system. General-use tools developed outside the environment must account for problems that more project-specific software can solve as they appear. When implementing tools like these, developers must spend time understanding the tool and how it fits into the existing infrastructure of the project. Therefore, if it is possible to design tools to minimize the time spent on implementation without sacrificing functionality, it should be seen as an important part of tool development. This thesis's aim was to determine the impact of implementation-conscious design on the ease and speed of tool implementation.

The topic of the impact of implementation-conscious design was chosen, as there is a lack of material available on design principles focusing on tool development. This lack of material can be attributed to lack of demand for academic research on the topic, but increasing global interest in metaverse platform development makes this topic highly relevant. As metaverse platforms benefit from being able to add new tools as needed, it is important that these tools can be seamlessly implemented without the need for large system changes. The thesis hypothesizes that being conscious of the impact of tool design on implementation quality is especially important in metaverse development.

The thesis aims to test the impact of implementation-conscious design on implementation quality by using available material on tool design and finding a set of design principles to follow. These principles are then followed to develop a tool for procedurally instantiating content in a Unity project using the Unity Addressable Asset System, with a specific focus on easy implementation of the tool into any future project. As the tool developed is partially intended for use in possible metaverse platform projects, the theoretical part of this thesis explores the current state of metaverse platforms and their use-cases in addition to a brief comparison of game engines and an evaluation of Unity Engine from the perspective of metaverse platform development.

The tool developed was created for the Futuristic Interactive Technologies research group at Turku University of Applied Sciences, which commissioned this thesis. The tool was commissioned as a way for metaverse users to quickly request an arbitrary number of assets as they need them and have them appear instantiated and placed sensibly. The practical part of this thesis describes the design principles used in developing the tool and how they were implemented in the final product. Finally, the design principles are evaluated based on a small group of interviewed experts.

2 Metaverse, the Internet of Experiences

As businesses become increasingly globalized and more people start to work from home, the demand for platforms that enable virtual collaboration between co-workers and businesses is on the rise. Companies like Microsoft, Nvidia and Meta have made significant investments to develop metaverse platforms in response to this demand. These platforms aspire to recreate real world places and objects, to create a "permanent, immersive mixed-reality world where people and people and people and objects can synchronously interact, collaborate, and live beyond the limitations of time and space, using avatars and the immersion-supporting devices, platforms, and infrastructures" (Lee & Kim 2022, 3–4). This two-way link between the real world and the virtual world is the defining characteristic of the metaverse and the feature that allows metaverses to provide unique benefits to businesses.

To capitalize on these advantages, companies like BMW and AB InBev, parts of the financial industry and many educational environments have partnered with metaverse platforms to create "digital twins" of their factories or workspaces to enable virtual collaboration (George 2021; Karkaria 2023; Chen 2023, 9; Hussain 2023, 2–3). These companies believe that being able to test new things in a virtual space before any real-world commitment will speed up production steps like planning and testing while reducing the risk of errors.

Virtual collaboration in the metaverse lets people exchange information in an immersive way, replacing traditional slideshow presentations with physical simulations and examples. Collaborators can freely interact with and manipulate the virtual world, rapidly sharing and expanding on ideas. Any problems found while in the metaverse can be quickly addressed and the changes immediately presented to any collaborators. Even large-scale projects can be planned and simulated thoroughly, eliminating mistakes early without incurring the high costs in time and resources caused by real world iteration.

However, it is essential to recognize that a metaverse, at its core, only serves as a platform that can support additional content and tools to facilitate collaboration. These tools must be built for increasingly specific purposes as more vague and broad tools fail to meet the ease-of-use and feature demands of new users. By building simple and maintainable tools early, these tools can be modified, iterated upon, and reused later to create tools for specific use cases more easily. By simplifying and speeding up the creation, implementation, and maintenance of these tools, the widespread adoption of metaverses can also be accelerated.

3 Unity as a Game Engine for Metaverse Development

While many of the larger metaverse platforms run on completely proprietary software, smaller studios looking to participate in the metaverse market might look towards commercially available game engines to jumpstart their development. This chapter will mainly focus on Unity, as it is the chosen engine of the commissioner of the tool developed, but a short comparison of other game engines will be included for context. The engines are mostly compared using data provided by the developers of the engines and a summary of the comparison can be seen in table 1.

3.1 Comparison of game engines

There are countless viable options for game engine choice, but this comparison specifically examines Unity, Unreal Engine and Godot. These engines have a wide appeal and low specificity, but their differences still make each engine the most desirable choice for different developers.

Unity markets itself as a game engine for mobile game development and for multiplatform projects (Unity 2024b). It does not boast the ability to produce the state-of-the-art graphics of more industry-standard options like Unreal Engine, while still advertising its built-in technology more than free, community focused game engines like Godot (Unreal Engine 2024a; Godot 2024b). While it is not the focus of its marketing, the size of Unity’s asset library compared to other engines hints at its community being larger than its competitors, a conjecture supported by it being the game engine with the largest userbase (Unity 2024d; Unreal Engine 2024c; Godot 2024d; SlashData 2022). It is overwhelmingly the engine of choice for top mobile developers, with over 70% of the top one thousand mobile games being made with Unity (Unity 2024a).

Unreal Engine posits itself as the game engine to use for teams who want to take advantage of state-of-the-art technology and to minimize the performance overhead of their projects (Unreal Engine 2024a). It is preferred over Unity by major studios with high budgets, with the total game revenue of computer games developed with Unreal being over double that of Unity (Milenovic 2023).

Table 1. Comparison of game engines.

	Unity	Unreal Engine	Godot
Pricing	Free with paid professional versions, percentage of revenue, runtime fee (Unity 2024e; Unity 2024f)	Percentage of revenue (Unreal Engine 2024d)	Free (Godot 2024e)
Programming languages	C#	C++	C++, C# and GDScript
Asset library size	93019 assets (Unity 2024d)	42658 assets (Unreal Engine 2024c)	1674 assets (Godot 2024d)
Advertising focus	Multiplatform, Mobile development (Unity 2024b)	High Graphical fidelity, Performance (Unreal Engine 2024a)	Community, Modularity, Simplicity (Godot 2024b)

Godot is open-source and thus completely free but while its usage with independent developers is on the rise, it has yet to see significant use by major studios (Milenovic 2023; Godot 2024e). The programming language unique to Godot, GDScript, has high performance overhead compared to C# but is designed to be easy to learn and use (Godot 2024a; GDScript 2024). Godot also supports C++ and C#, making using GDScript completely optional.

3.2 Pricing

For professional use, Unity is generally more affordable as a development platform than Unreal Engine. The main exception to this is that, as of the writing of this report, Unity has a runtime fee which penalizes developing software that makes low revenue per install. This means that for a small team developing an openly distributed free platform with optional purchases, the combined price of Unity software license and the runtime fee might add up to a much larger fraction of their revenue than the 5% revenue split of Unreal Engine (Unity 2024f; Unity 2024e; Unreal Engine 2024d).

This directly shapes which types of metaverse platforms are feasible to develop with Unity, as free-to-play virtual worlds intended as social platforms will likely incur a large runtime fee, but more specialized platforms intended only for a small-to-medium userbase are very affordable.

3.3 Build target diversity

The amount of popular desktop, console and mobile platforms have over time reduced to a handful of options each. Currently these options are Windows, MacOS and Linux for desktops, Xbox, PlayStation and Switch for consoles and iOS and Android for mobile devices. While these platforms will change over time and thus require continuous support from the developer of each game engine, all game engines compared support the newest version of all these platforms (Unity 2024c; Unreal Engine 2024b; Godot 2024c).

Conversely, there are many XR (extended reality) platforms in use, with new ones still being developed. Unity offers a wide range of supported XR platforms, with their own framework specifically built for XR development (Lexis et al. 2022).

Metaverse developers that want to leverage XR technology must consider which XR platforms their product will be used with. If the metaverse platform is openly distributed, their userbase might be using very varied XR platforms, as many XR hardware manufacturers have their own platform that must be specifically accounted for when developing a metaverse platform. The wide range of platforms supported by Unity makes it an attractive choice for developers who are not developing with specific hardware in mind.

3.4 Performance

The only programming language compatible with Unity is C#, a high-level language designed with simplicity and generality in mind (ECMA 2006, 21). While this choice of language makes Unity more approachable for less savvy programmers, the language lacks the performance and capability for fine-grain optimization that languages like C++ have (Ogala et al. 2020, 11–14).

Unity is developing a new architecture called DOTS (Data-Oriented Technology Stack), which has been tested to reduce frame times by up to 50 times, but the architecture is not yet ready for production (Antich, 2023). These performance gains apply mostly to projects with many parallelizable actions, which may be relevant to large, openly distributed metaverse platforms.

Depending on the kind of metaverse platform being developed, the relatively inferior performance of Unity may be a dealbreaker. If the platform requires high graphical fidelity, extensive simulation or is intended for low powered hardware, developers might look towards a different engine. Metaverse platforms developed with Unity will likely have to opt for a simpler presentation, something that the largest metaverse platforms already do. For simpler, well optimized platforms like these, Unity will effectively have equal performance to other game engines.

3.5 Asset Store

By counting the assets in each asset category on the websites of all three asset libraries compared, the amounts of assets provided in table 1 can be found. It is apparent from these numbers that Unity has the widest array of ready-made assets and plugins available out of all game engines compared. This can be attributed to the Unity Asset Store being older than the asset distribution websites of the other engines. Everything on the Unity Asset Store can be used in used in any number of projects after a single payment, so development teams who have used Unity in the past have extra incentive to continue using Unity for their future projects.

The Unity Asset store contains multiple solutions for easy-to-use and high-performance multiplayer game development. This makes developing a metaverse platform with Unity more straightforward and frees up development time for other tasks. Other assets available on the store can be used as placeholders to quickly reach a minimum viable product, or to test out new functionality. Using assets in this way helps developers focus on the unique aspects of their platform first, gradually replacing assets with self-developed content as necessary.

3.6 Addressables

The Unity Addressable Asset system is an asset management system which allows loading assets in a more dynamic manner. The system allows for smaller update sizes, better content packaging workflow and better support for live content delivery (Palmer 2019). The system also supports loading assets from the web, simplifying user submitted content distribution and remote asset storage.

While all these benefits can be relevant for metaverse platform development, having support for delivering live content updates at runtime is an especially great tool for digital twin metaverse platforms. This allows multiple people to collaborate on modifying an asset and to see it update in real time, for developers to update existing assets without the need for distributing an updated version of the platform and for users to upload and modify their own assets uploaded to the web.

4 Striving for Quick and Easy Implementation

4.1 Design Motivation

Building a fully outsourced tool comes with a set of challenges and considerations. The tool will have to fit into a largely unknown workflow and be able to be implemented into a project that the developer of the tool has no access to, often with the understanding that the client will not be able to contact the developer with questions afterwards. It is therefore vitally important to design tools to both intuitively and through direct guidance minimize both the friction during implementation and the problems the client will have with the tool. Anticipating questions posed by developers and providing the information required to answer those questions is an important part of tool design (LaToza & Myers 2011, 2).

Through good and open communication with the client both before and during development, developers can ascertain their tools will not only function as required, but also fit well into their environment. However, this type of communication is not always possible, causing developers to have to rely on other means of guaranteeing the quality of their tools.

4.2 Tool Structure

Structuring a tool to have a clear delineation between what the developers using the tool should and should not interface with helps make the tool more readable by cutting down on the amount of information the developers will have to digest (Scalabrino et al. 2016, 10; Tashtoush et al. 2023, 25). Depending on the complexity of the tool, it can also be useful to have added delineations between different levels of custom implementations.

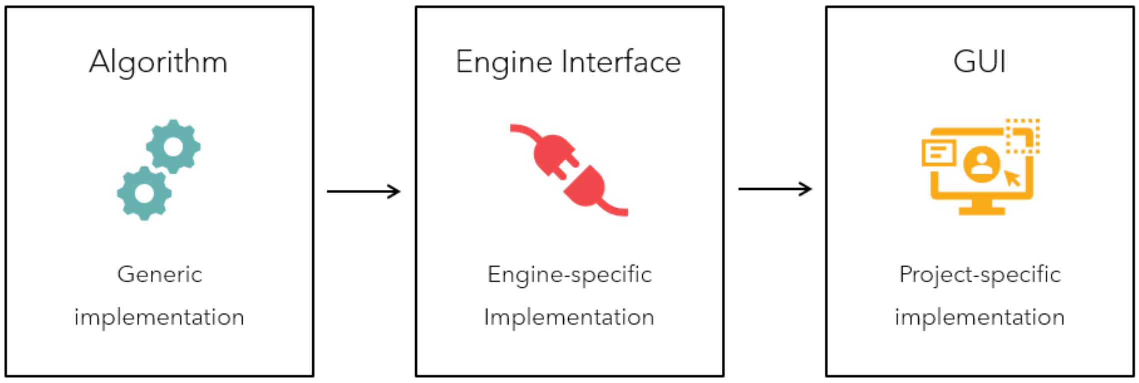
Algorithm
Generic implementation
Engine Interface
Engine-specific Implementation
GUI
Project-specific implementation

Image 1. An example of class separation by implementation depth.

As an example, a tool can be divided into a GUI, an interface class and a background work class as seen in image 1. For basic implementation, using the GUI might be sufficient, very straightforward and require no knowledge of programming or how the tool works. For more advanced implementation, the interface class lets developers with some programming knowledge and understanding of the tool implement it in a more customized way, while still hiding the logic they do not need to access or understand in the background work class.

4.3 Code Formatting

Readable code not only helps tool developers make less mistakes during development but is especially important when the developers know it will have to be read and understood by other people. Following code format guidelines and naming conventions will reduce misunderstandings and general confusion the client will have to deal with during implementation (Scalabrino et al. 2016, 10).

When building an outsourced tool, tool developers must be especially conscious of naming conventions. Names the developers of the tool are used to and understand might be completely foreign and opaque to their target developers. The solution is to sacrifice brevity for clarity, name even short-lived variables descriptively and avoid inline declarations (Scalabrino et al. 2018, 21).

Adding comments to code is the most obvious way to clarify its purpose and convey information to the reader, but often well written code does not need comments and over-commenting ends up making the code slower to read and harder to parse (Spinellis 2003, 4; Scalabrino et al. 2018, 21). However, comments briefly explaining complex or important parts of code or warnings against modifications can not only speed up the implementation process, but also potentially save the client from making mistakes the tool developers have already dealt with.

4.4 Supplemental material

For more complex tools, supplemental materials provide a way to communicate with the client indirectly. Through examples and explanations, supplemental materials can help clear up confusion about a tool and direct its proper usage (Garousi et al. 2015, 16–17). The types of supplemental material a tool should use depend on many factors, such as platform, complexity, and the client. There are, however, some supplemental materials which can be added to almost any tool.

Technical documentation is seen as an important practice for most software development. It allows the tool’s creator to highlight important parts of the tool and explain its functionality in an intuitive way. While properly implementing any tool that uses the aforementioned design guidelines is possible without technical documentation, the client would have to read through code to understand it. Technical documentation can provide a step-by-step process for implementation, answer potential questions, isolate, and explain important parts of code and help troubleshoot common issues.

Implementation examples are supplemental materials that let the client see what a possible implementation could look like. These can range from full examples, where the tool is used as a part of a small project, often alongside other tools, to small examples that only demonstrate a possible implementation of part of the tool. These examples can ease implementation by letting the client reverse engineer the example and see how to build their own, or even use parts of the example in their own implementation.

4.5 Questionnaire and results

A group of 3 engineers from the FIT (Futuristic Interactive Technologies) research group at Turku University of Applied Sciences were invited to respond to a short questionnaire about the impact of the design elements described in this chapter on the speed and ease of implementing a tool. All engineers had experience in videogame programming and Unity Game Engine. The goal of the questionnaire was to gauge the opinions and their homogeneity of engineers with experience in implementing outsourced tools.

The elements were divided into 4 structures: class separation, code formatting, documentation, and examples. The engineers were briefed on the meaning of each structure, after which they rated each structure on a scale of 1 to 7, with grade 1 meaning the structure is useless for implementation and grade 7 meaning implementation is impossible without the structure.

The engineers mostly agreed on each structure, with the “examples” structure being an outlier with a grade variance of 4.32 compared to the average grade variance of 1.5 and an average grade variance of 0.56 when ignoring the “examples” outlier.

Table 2. Interview data.

	Class separation	Code formatting	Documentation	Examples
Engineer 1	5	4	6	3
Engineer 2	6	4	5	7
Engineer 3	7	5	6	4

All structures received average grades above the median available grade of 4, with the “class separation” structure being rated highest with an average grade of 6, emphasizing the importance of a clear separation of different implementation levels. As seen in table 2, the “documentation” structure was rated only slightly below the “class separation” structure with an average grade of 5.67 and the “code formatting” structure was rated lowest with an average grade of 4.32. As the grade variance of the “Examples” structure was so high, its impact on implementation is less clear from the results, but it should be noted that two of the engineers rated it as the least important structure, while one engineer rated it as being a requirement for a successful implementation.

While the small size of the study prevents it from providing any conclusive evidence, the high average grades support the hypothesis that conscious attention paid to these specific design elements helps developers implement tools faster and easier.

5 Development and Structure of the Tool

5.1 Purpose of the tool

For this thesis, a tool was commissioned by the FIT research group for use in various future projects. The goal of the tool was to use the Unity Adressables system to add objects into the game world in a specific manner. Players can choose the objects and their amounts, and those objects will appear in the game world. This is only restricted by the area in which the objects appear in, if the objects cannot fit, they will not appear. The purpose of the tool therefore is to take the selected objects and try to find a way to fit them all inside the area, no matter their size or shape.

5.2 Design Reasoning

The tool is designed to be as easy to understand and implement as possible, with potential problems either intuitive to solve, or anticipated in the design process and a solution provided in the supplemental material. The tool is also designed to be easily implementable into as wide of a range of different projects as possible.

The separation of the base class and the area class serves two essential functions. The clear separation between an interfacing class and a functional class intuitively tells the developers which parts of the code they are meant to access. The separation between code that needs to access Unity namespaces and code that does not increases modularity and allows parts of the code to be used in a wider range of projects.

An additional separation layer could be added by isolating every public method of area class into its own class, but since the same separation is implicit in the class between public and private methods, it was not deemed necessary.

5.3 Structure of the tool

The tool primarily consists of two classes: a class that generates a solution for packing rectangular objects into a rectangular area, referred henceforth as "base class", and a class that interfaces between the base class and Unity, referred henceforth as "area class". The base class uses a binary tree 2D (two-dimensional) bin-packing algorithm with additional conditionals to check for solutions where some or all the objects are rotated by 90 degrees (Skiena 2008, 595–597). The class runs entirely asynchronously. The area class is attached as a Unity component to a prefab, which is an empty GameObject and is used for the position, rotation, and size of the area in which to spawn the objects. Any number of these objects can exist and function simultaneously in a scene.

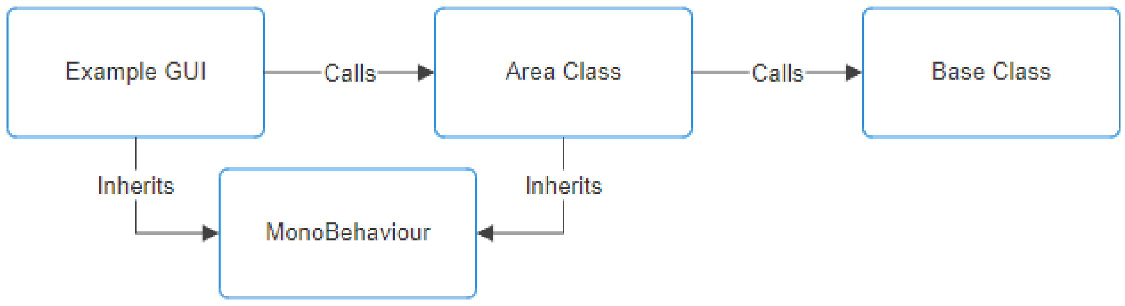

Image 2. Class structure of the tool.

While the tool does not primarily include any interface between the area class and the user, an example interface class for this purpose is included as supplemental material. This class uses a simple mouse and keyboard GUI to pass user input data to the area class. Additionally, a prefab is supplied that uses this class and default Unity user interface elements to build a full mouse and keyboard GUI (Graphical User Interface) that works without modification when dropped into any scene.

The hierarchical structure of the tool can be seen in image 2, with each class only calling the class more generic than itself. The base class not inheriting the Unity MonoBehaviour class required for all Unity integration makes it the most generic, only dealing with abstract data passed to it when called by the area class.

A sample Unity project that implements every aspect of the tool is provided as supplemental material. This project consists of a simple scene only including the aforementioned example GUI and area prefab.

Technical documentation precisely describing the tool and all its parts is included as supplemental material. This documentation also includes a step-by-step guide for quick implementation.

5.4 Functional description

Base class

The base class uses a binary tree 2D bin-packing algorithm, modified to account for more possible solutions (Skiena 2008, 595–597). The algorithm has been modified in two ways:

1. Whenever an object is recognized to not fit in a proposed position, the algorithm rotates the object 90 degrees around the y-axis and tries again.

2. Whenever the algorithm completes but could not find a solution, it will rotate every object 90 degrees around the y-axis and tries again. The second attempt cannot trigger this behavior again.

These modifications work in tandem to vastly increase the number of arrangements considered. The unmodified algorithm tries to place every object in its original orientation, and therefore can often fail to find a solution when using objects with a significant difference between height and width. The first modification solves this problem by checking two different orientations. Since every object is rectangular, this eliminates all possible rotations with 90-degree increments. This modified algorithm still prefers placing objects in their original orientations and can therefore lock itself out of a possible solution. The second modification solves this problem by running a second check that prefers rotated orientations the first time the algorithm determines a solution could not be found.

As seen in the top-right section of image 3, with available space in both horizontal and vertical directions, the red objects stay vertical and the yellow objects horizontal. When the horizontal area is restricted, as seen in the top-left section of image 3, some of the yellow objects rotate to fill the space more efficiently. In the bottom section of image 3, because the red objects are placed first and the algorithm prefers to place the objects in their original orientations, the vertical size of the area has to be restricted to be less than the height of one red object for the objects to rotate.

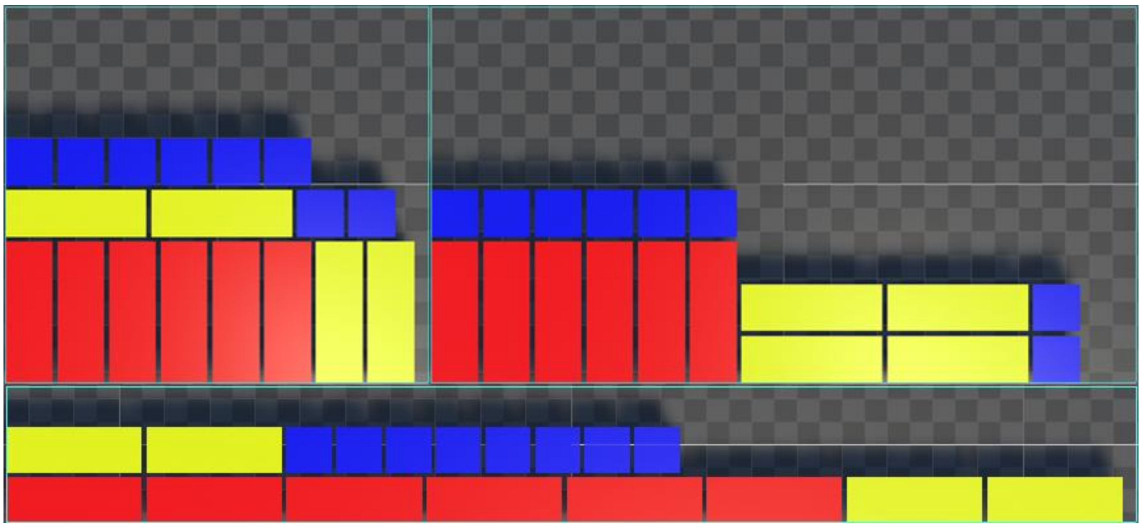

Image 3. Example of different solutions caused by different area restrictions.

Since the full algorithm is computationally expensive, it is important to use less resource intensive checks before initiating it to stop execution early when a solution clearly cannot be found. The checks included in the base class stop execution if the combined area of the objects is greater than the area of the spawn area, if the height of any of the objects is larger than the height of the spawn area, or if the base class is called with parameters which would cause errors in the algorithm.

The algorithm outputs an array of 2D coordinates, with (0,0) being the bottom left corner of the spawn area and assumes each object's pivot is also at its bottom left corner.

The algorithm was programmed to run asynchronously to minimize its impact on game performance.

Area class

The area class is an interface between the base class and Unity. It provides multiple functions that allow the base class to seamlessly be used with Unity Addressables. Its functions include:

1. Loading addressables.

2. Calculating the bounds of objects.

3. Transforming the 2D coordinates calculated by the base class into three-dimensional coordinates.

4. Instantiating objects.

5. Moving objects to align to a new corner of the spawn area without reinstantiation.

6. Visualization of the spawn area in Unity editor.

The area class is mainly accessed through a single method which accepts information consisting of the objects to spawn, the amount of each object to spawn, the amount of space to leave between each object and the corner of the spawn area to align the objects towards. It then executes the first four of the functions listed above, in order. Other methods exist to guarantee functionality, help improve performance or to provide useful information.

Transforming the 2D coordinates calculated by the base class into three-dimensional coordinates is the most involved part of the area class. The coordinates of each object need to be partially or fully negated if the objects are not to be aligned to the bottom-left corner of the spawn area. They then must be offset by the position of the spawn area, the position of the object’s pivot and finally both rotated and moved to match the rotation of the spawn area. All these calculations are different depending on the chosen corner of the spawn area to align the objects towards.

To account for the bottom-left corner alignment of the base class coordinates, the area class pre-emptively calculates vectors from world (0,0,0) to each of the four bottom corners of the spawn area. It also calculates vectors from the four bottom corners of each object to its pivot and stores them for future use. Once these vectors are calculated, they can be used to calculate offsets for any combination of chosen alignment corner and rotation of the object.

The area class contains a method for changing the alignment corner without executing the base class algorithm again by calculating new positions for the objects and directly changing their positions to match. Using this method instead of the main method whenever possible helps with performance, since it doesn't execute the expensive base class algorithm, nor does it destroy and reinstantiate the existing objects.

5.5 Supplemental material

Example GUI

The example GUI provides a simple mouse and keyboard interface for the user and interacts with the area class. It consists of a C# class attached as a component to a Unity prefab, which contains all the necessary elements of the GUI.

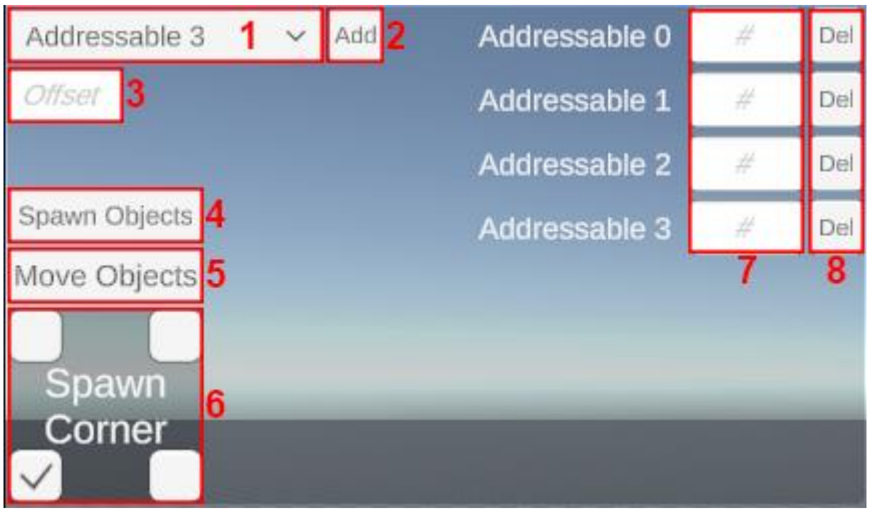

Image 4. Layout of the Example GUI with elements highlighted for clarity.

As seen in image 4, the interface includes:

1. A dropdown menu listing available addressables
2. A button to add the selected addressable to the list
3. A field for specifying the offset between each addressable
4. A button to spawn the selected addressables in specified quantities
5. A button to re-align existing objects without respawning the objects
6. A set of radio buttons to select the alignment corner of the objects
7. Fields for specifying the amount of each selected object
8. Buttons for removing a selected object from the list

The example GUI has no functionality for selecting different spawn areas, so it can only be used for situations where a scene contains only a single spawn area.

Sample Unity project

The sample Unity project consists of a simple scene, only containing a floor, a single spawn area, the example GUI, and four simple prefabs configured as addressables.

No additional setup is needed for the project to work, and all the elements of the tool are on display and ready for experimentation.

Technical documentation

An eight-page technical documentation is provided with the tool.

The documentation contains:

1. A brief description of the tool and it's components

2. A list of notable issues with the tool

3. A step-by-step guide for simple implementation

4. An in-depth description of each class and their methods

6 Conclusion

The objective of this thesis was to study the importance of designing tools with ease and speed of implementation in mind, focusing on development for metaverse platforms. A tool commissioned by the FIT research group at the Turku University of Applied Sciences was developed with a focus on implementation design and used as an example in a small study to gauge the views of developers on the importance of specific design elements.

As the tool was commissioned for Unity development with plans of implementation into future metaverse projects, the thesis examined the game engine from the perspective of metaverse platform development in comparison to Unreal Engine and Godot. This examination showed Unity to have many beneficial aspects for the purpose, with the downsides relative to other examined game engines being situational and manageable.

A small study on the importance of specific design elements was conducted by querying developers with experience in programming. Due to the small sample size of the study the results are inconclusive, but support the hypothesis that implementation-conscious design, specifically design using class separation, readable code and additional materials help speed up implementation.

The thesis could have been improved by researching and documenting more approaches to tool design before the beginning of development and conducting a larger scale study to compare the impact of these approaches to the implementation process of the tool. A more conclusive study examining the implementation quality of the tool developed was considered, but because of the time required to implement even an easily implementable tool, such a study was out of the scope of this thesis. The results of the study conducted in this thesis only reveal the importance of the design elements in theory and cannot be used to draw conclusions on their effectiveness in practical applications. In addition, without any specific study on the relative importance of implementation quality in the development of metaverse platforms, the original hypothesis of implementation-focused design being more important for this purpose can only be supported by written material.

Future research on the topic could benefit from interviewing developers who have released developer tools on platforms like the Unity Asset Store. As the tools released on platforms like these are exactly the kind of general-use tools this thesis is focused on, their developers could have important insight into useful design practices and their impact on implementation.

Overall, the thesis finds implementation-conscious design an important part of a properly developed tool, with theoretical benefits for ease and speed of implementation. The hypothesis that implementation-focused design has elevated importance in metaverse platform design was not confirmed by the thesis.

References

Antich, A. “Unity DOTS / ECS Performance: Amazing” [online]. 2023. Consulted 23.01.2024. https://medium.com/superstringtheory/unity-dots-ecs-performance-amazing-5a62fece23d4

Chen, Z. Metaverse and Stock Market—A Study Based on Fama-French Model. In: Proceedings of the 2022 3rd International Conference on E-Commerce and Internet Technology (ECIT 2022). Atlantis Press International BV; 2023:725-734. doi:10.2991/978-94-6463-005-3_74

ECMA. 2006. C# Language Specification (4th ed.). 2006. www.ecma-international.org

Garousi, G., Garousi-Yusifoğlu, V., Ruhe, G., Zhi, J., Moussavi, M., Smith, B. Usage and usefulness of technical software documentation: An industrial case study. Information and Software Technology. 2015;57:664-682. doi:10.1016/j.infsof.2014.08.003

GDScript. “GDScript” [online]. 2024. Consulted 7.2.2024. https://gdscript.com/

George, S. “Converging the physical and digital with digital twins, mixed reality, and metaverse apps” [online]. 2021. Consulted 10.12.2023. https://azure.microsoft.com/en-us/blog/converging-the-physical-and-digital-with-digital-twins-mixed-reality-and-metaverse-apps/

Godot. “Performance of C# in Godot” [online]. 2024a. Consulted 7.2.2024. https://docs.godotengine.org/en/stable/tutorials/scripting/c_sharp/c_sharp_basics.html#performance-of-c-in-godot

Godot. “Features” [online]. 2024b. Consulted 5.2.2024. https://godotengine.org/features/

Godot. “List of features” [online]. 2024c. Consulted 5.2.2024. https://docs.godotengine.org/en/stable/about/list_of_features.html#platforms

Godot. “Godot Marketplace” [online]. 2024d. Consulted 5.2.2024. https://godotmarketplace.com/product-category/all/

Godot. “License” [online]. 2024e. Consulted 5.2.2024. https://godotengine.org/license

Hussain, S. Metaverse for education – Virtual or real? Frontiers in Education. 2023;8. doi:10.3389/feduc.2023.1177429

Karkaria, U. BMW to Build Factories Faster Virtually: Nvidia’s Omniverse Cuts Costs, Increases Flexibility.; 2023. https://www.proquest.com/trade-journals/bmw-build-factories-faster-virtually/

LaToza, T. D., Myers B. A. Designing useful tools for developers. In: Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools. ACM; 2011:45-50. doi:10.1145/2089155.2089166

Lee, U. K., Kim, H. UTAUT in Metaverse: An “Ifland” Case. Journal of Theoretical and Applied Electronic Commerce Research. 2022;17(2):613-635. doi:10.3390/jtaer17020032

Lexis, T. A., Flynn, S., Ruddell, D., Brown, D., McDonald, W. ”Games Focus: Extending your reality with XR” [online]. 2022. Consulted 30.01.2024. https://blog.unity.com/engine-platform/games-focus-extending-your-reality-with-xr

Milenovic, S. “Exploring the PC Game Engine Landscape” [online]. 2023. Consulted 7.2.2024. https://www.gamedeveloper.com/game-platforms/exploring-the-pc-game-engine-landscape

Ogala, J., Ogala, B., Onyarin, J. COMPARATIVE ANALYSIS OF C, C++, C# AND JAVA PROGRAMMING LANGUAGES. Global Scientific Journal. 2020;8(5):1899-1913. https://www.researchgate.net/publication/358368843

Palmer, S. “Addressable Asset System” [online]. 2019. Consulted 30.01.2024. https://blog.unity.com/games/addressable-asset-system

Scalabrino, S., Linares-Vásquez, M., Oliveto, R., Poshyvanyk, D. A comprehensive model for code readability. In: Journal of Software: Evolution and Process. Vol 30. John Wiley and Sons Ltd; 2018. doi:10.1002/smr.1958

Scalabrino, S., Linares-Vasquez, M., Poshyvanyk, D., Oliveto, R. Improving code readability models with textual features. In: 2016 IEEE 24th International Conference on Program Comprehension (ICPC). IEEE; 2016:1-10. doi:10.1109/ICPC.2016.7503707

Skiena, S. S. The Algorithm Design Manual: Second Edition. Springer London; 2008. doi:10.1007/978-1-84800-070-4

SlashData. “Did you know that 60% of game developers use game engines?” [online]. 2022. Consulted 7.2.2024. https://www.slashdata.co/post/did-you-know-that-60-of-game-developers-use-game-engines

Spinellis D. Reading, Writing, and Code. Queue. 2003;1(7):84-89. doi:10.1145/957717.957782

Tashtoush, Y., Abu-El-Rub, N., Darwish, O., Al-Eidi, S., Darweesh, D., Karajeh, O. A Notional Understanding of the Relationship between Code Readability and Software Complexity. Information (Switzerland). 2023;14(2). doi:10.3390/info14020081

Unity. “Games” [online]. 2024a. Consulted 7.2.2024. https://unity.com/games

Unity. “Unity Engine” [online]. 2024b. Consulted 5.2.2024. https://unity.com/products/unity-engine

Unity. “Platform Development” [online]. 2024c. Consulted 5.2.2024. https://docs.unity3d.com/Manual/PlatformSpecific.html

Unity. “Unity Asset Store” [online]. 2024d. Consulted 5.2.2024. https://assetstore.unity.com/

Unity. “Plans and Pricing” [online]. 2024e. Consulted 30.1.2024. https://unity.com/pricing

Unity. “Runtime Fee Estimator” [online]. 2024f. Consulted 30.1.2024. https://unity.com/runtime-fee-estimator

Unreal Engine. “Unreal Engine 5” [online]. 2024a. Consulted 5.2.2024. https://www.unrealengine.com/en-US/unreal-engine-5

Unreal Engine. “Features” [online]. 2024b. Consulted 5.2.2024. https://www.unrealengine.com/en-US/features

Unreal Engine. “Unreal Engine Marketplace” [online]. 2024c. Consulted 5.2.2024. https://www.unrealengine.com/marketplace

Unreal Engine. “License” [online]. 2024d. Consulted 5.2.2024. https://www.unrealengine.com/en-US/license