

Zurich University of the Arts

Game Design

Orientation C



The Unity Glue

Principles and strategies for a maintainable Unity project structure.

Author:
Goran Saric
May 22, 2017

Mentors:
René Bauer
Mela Kocher

Abstract

Today game developers can find hundreds of tutorials on the internet covering specific topics for the popular game engine “Unity”. However, there is a lack of corresponding literature to explain how to integrate all of these different topics together into one big, scalable, and maintainable project.

Moreover, Unity’s application program interface was not designed to follow a specific workflow: game developers have the freedom to work in various ways. I personally think this open setting is especially good for fast prototyping production processes, but when it comes to larger projects, where multiple people have to work together productively, the overly flexible engine can easily lead developers to organization problems and bugs.

To address this problem, my master’s thesis focuses on project structures within Unity 5.6. In order to first get an overview of existing practices, I interviewed individual developers and game studios. By analysing different methods and approaches for organizing assets and source code, I developed a reliable workflow for my current Unity game “*FAR: Lone Sails*” [1]. The results of my research are a “cookbook” that helps to create a maintainable project structure for teams working on a collaborative basis in Unity.

Contents

1 Introduction	5
1.1 Thesis structure	5
1.2 Context	6
1.2.1 My background	6
1.2.2 FAR: Lone Sails	7
1.2.3 Why Unity?	10
1.3 Research question	10
1.4 Assumptions	10
1.5 Target audience	10
1.6 Definitions	11
2 Research	13
2.1 Review of the literature	13
2.2 Not the Unity way	15
2.3 Interviews	16
2.3.1 Survey	16
2.3.2 Results	17
3 Basic knowledge	21
3.1 Vocabulary	21
3.2 Editor	23
3.2.1 Assets and objects	23
3.2.2 Prefabs	24
3.2.3 Multi scene editing	25
3.3 Scene management	26
3.3.1 Unity scenes	26
3.3.2 Asynchronous scene loading	27
4 Organize your project	29
4.1 Editor content	29
4.1.1 Naming	29
4.1.2 Folder hierarchy	30
4.1.3 Multi scene setup	34
4.2 Scene content	37
4.2.1 Naming and hierarchy	37
4.2.2 Managers	39

Scene manager	42
Save game manager	44
Input manager	47
Debug manager	48
4.2.3 Level design management	51
Closed ecosystem	51
Animations	51
Events	53
4.3 Ten organization rules	55
5 Additional knowledge	57
5.1 Editor extensions	57
5.1.1 Keyboard shortcuts	58
5.1.2 Scene view helpers	60
5.2 Version control	61
5.2.1 Scene and prefab handling	62
5.2.2 Git LFS	62
5.3 Performance	63
5.4 Unit tests	64
6 Conclusion	65
6.1 Summary	65
6.2 Evaluation	66
6.3 Personal reflection	68
6.4 Outlook	69
7 Credits	70
8 Bibliography	71
9 Ludography	76
10 List of tables	76
11 List of figures	77
12 Licence	80

1 Introduction

1.1 Thesis structure

This thesis is set up in three main segments. In the beginning I write a little bit about my background and how I've come to be involved with the game project I'm currently working on. This explanation includes my motivations and reasons for writing this paper. I'm also clarifying definitions and questions in this section; For whom is this paper? How do I interpret the word "*structure*" and in which context is it used? My intent is to use the same technical language as the reader and to clarify any abstract and widely open definitions.

The second section contains more details about my scientific approach while finding and researching methodologies within Unity. First, I give the reader a slight overview of existing literature and research in this field. I explain how I talked to different individual developers and studios who were using Unity as their main tool, and how I tried to figure out their views in order to evaluate their experiences and hints. I also analyzed parts of the Unity documentation and existing tutorials on the internet.

In the last section I start with basic knowledge about Unity. I am not talking about a beginner's basics, but rather, basics I needed to know when I implemented my project strategies and structures. I'm definitely not explaining how to use the buttons of the editor; even if you are an experienced Unity developer, it makes sense to establish the basics as a starting place. Further chapters are based on this specific knowledge, and I also define some terminology related to the Unity editor at the beginning. After the basics, I write about a concrete project organization within the engine. This includes naming, asset management, scene handling, scene content, level design workflows, and programming related topics. Most chapters thus contain a first part with general thoughts and research about the certain topic. The second part shows the approach or implementation I used for my current game project. At the end there is a summary with future use cases and a personal conclusion of my findings and strategies.

1.2 Context

1.2.1 My background

Hi fellow readers! I'm happy that you are reading my master's thesis and I would like to give you a little summary of myself before you dive deeper into the paper. My name is Goran Saric, I am 30 years old, and this thesis is my final paper for my MA in game design at the Zurich University of the Arts in Switzerland. I have a background as a software engineer, and after travelling a year in Africa by motorbike, I decided to start a Bachelor in Game Design.

The skills I learned during my bachelor covered a lot of different game design topics. I learned the basics of art, programming, modeling, texturing, animating, sound sampling, game design, level design, and some storytelling. In the end I became a game design jack-of-all-trades, capable of working on any project involved with a game or in its related fields. All students in the BA program were free to choose any game engine they wanted to work with, and since Unity became popular among indie developers, most students of my class year chose it. We had to develop a fair amount of both small and bigger prototypes during my three years of study, and we even achieved a reasonable level of polish with our final bachelor projects.

After I finished my BA two years ago, I was motivated to finally work on a bigger project and to finalize and release a game. Together with five of my student colleagues, I founded a game design collective called "*Mr. Whale's Game Service*" [1].



Mr. Whale's Game Service. From left: Martina Hugentobler, Don Schmocker, Melanie Vetterli, Goran Saric, Christian Schmidhalter, Dominik Haas. [f1]

Our goal was to shine with a combined portfolio and to have a strong and skilled backbone built from different talents. After a couple of our prototypes were nominated and won prizes at relevant game events, we got the chance to pitch a few of our projects to investors and publishers. Our game "*DERU*" [2]

managed to get funding from the public Swiss foundation Pro Helvetia. With the early prototype of FAR: Lone Sails we managed to get a publisher on board. Since we ended up with two running projects simultaneously, we split our collective into two teams. At the same time I decided to start a Master Degree in Game Design and to use my findings as the technical lead of FAR: Lone Sails as a basis for my thesis. If you want to know more about my current and past activities, please visit my website: <http://www.goga.ch>.

1.2.2 FAR: Lone Sails

The original prototype called “FAR” was created by Don Schmocker [2] in the summer of 2015 as his final bachelor work. Don Schmocker is a member of our Mr. Whale’s Game Service collective, and we decided to take the project into our portfolio. FAR was very special in the artistic way it told a story about a girl’s journey across a dried out seabed. In the story, a harbor-based civilization is suffering from a tide phenomenon, an event which drags the sea level further and further away from the coast. The player controls a lonely little girl who was left alone with nothing but a big, mysterious vehicle. Together with the vehicle, the girl tries to find both the coastal edge and civilization. The gameplay is about the management and control of the vehicle during difficult weather situations and access denying obstacles. We call the genre a vehicle adventure, since it has a linear act, a story plot, and the player is mostly involved with managing a vehicle.



Screenshot of FAR: Lone Sails. [f2]

FAR won several prizes and even got an honorable mention at the IGF Student Award 2016. Since the project gained a lot of attention, several publishers have shown interest. Early in 2016 Don signed a contract with our current publisher “Mixtvision” [3]. In March 2016, I joined the project as a technical leader. My initial task involved an estimation about the usage of physics in the gameplay. Afterwards I built up the source code from scratch, since Don’s old prototype lacked in structure and extensibility. The rest of my technical lead related tasks formed the basis for this thesis paper. The following table can help one to get a better idea of some of the technical challenges facing the game [t1]:

Gameplay related facts:	Technical requirements:
<ul style="list-style-type: none"> • Singleplayer • Linear gameplay • 2-3h of gameplay • Physically based vehicle and player movement 	<ul style="list-style-type: none"> • Keyboard & gamepad input • Visual scripting tools for level designers • Save game system • No loading bars, seamless content loading • Runs on midrange GPU [4] laptops

In June 2016 Microsoft featured FAR in their indie portfolio at E3 in Los Angeles. [5]



Snippets from the FAR trailer during the Xbox E3 press conference. [f3]

We received a lot of positive press coverage from the event, and people visiting our booth gave us valuable feedback. One large problem we encountered was that nobody could find information about the game on the internet because of its short name. We decided together with our publisher to add the subtitle “*Lone Sails*” in order to fulfill SEO [6] related needs.

In January 2017, the new prototype for FAR: Lone Sails got nominated for the Student Award at the IGF. Since then two alumni friends from the Zurich University of the Arts bachelor program have joined our team part time and are still working with us on the game.



FAR: Lone Sails team at IGF 17. From left: Goran Saric, Martina Hugentobler, Don Schmocker [f4]

FAR: Lone Sails should be released on Steam in November 2017 and on Xbox One and Playstation 4 in the first quarter of 2018. [7]

1.2.3 Why Unity?

In 2005 an unknown game engine appeared on the horizon. It was called “Unity 3d” and it soon became one of the most popular game engines for indie developers worldwide. Since its earliest days, Unity enticed developers with its simplicity, its entity based structure, and its support for the UnityScript and C-Sharp programming languages. Unity also established a wide range of support for a variety of popular target platforms. Unity thus promised to minimize the struggle of deploying a game on multiple platforms. I believe that this is still the main reason a lot of developers prefer Unity over other engines. As a developer, I personally enjoy working with Unity because of its scene editor. The controls are intuitive, and once you learn all of the important shortcuts, you can create abstract worlds within hours.

1.3 Research question

Considering the above current state of affairs for FAR: Lone Sails, the question I arrived at for my thesis is this: How can indie game developer teams achieve a scalable and maintainable project structure within Unity 5.6?

1.4 Assumptions

- Determining basic project structure rules will improve the overall quality and efficiency of the production workflow.
- Early logic planning for seamless content loading will help to sort and to form the final game content bundles.
- Additional self-implemented debugging tools will help to avoid heavily time consuming bug determining processes.

1.5 Target audience

First and foremost, this paper is strongly technically oriented. People who do not know anything about game development and programming in general will have a hard time in understanding some chapters. Since I am writing about best practise methods for a concrete game engine, people who use Unity will profit the most. Therefore, this paper is targeting intermediate and experienced Unity developers, i.e. people involved in structuring and programming related tasks. It is definitely not written for Unity beginners and artists, even though they might

find helpful information that helps them to understand their technical lead's point of view.

Unity is special in its unique feature implementation and is undeniably different from other engines, but it is of course still a game engine. It therefore still produces some comparable, cross-engine issues to deal with. This is why developers working with other engines will also likely find chapters and topics that overlap with their particular development environment.

As mentioned in the chapters before, I found myself working on a game with a publisher contract after I had finished my bachelor degree. I was confronted with questions I had never had to answer when I was developing small prototypes and projects for my studies. I found myself worried about whether I could see the big picture of the project's technological scope, which was heavily relying on me to be implemented. I hope this thesis can help other developers in similar positions to get ideas for problems that can occur with their projects, or to find a little guidance on questions which they still haven't answered on their own yet.

1.6 Definitions

The following list contains definitions that will appear throughout the thesis. [\[t2\]](#)

Definition 1:

Project structure

A project structure is a mesh based connection between logical and physical elements within Unity. Physical elements in Unity are things like textures, and logical actors are things like programming code. The project structure defines ways for how these different elements can interact with each other.

Definition 2:

Workflow

A workflow describes a logical order and sequence of work tasks. For example, "*I have to transform item A into item B*". Every sequence in between A and B that leads to end state B describes a workflow.

Definition 3:

Design pattern

By design pattern I mean a reusable structure of solutions intended for a certain issue or problem. In this thesis the term is used mostly to address specific software design patterns.

Definition 4:

Gameplay

The gameplay describes the connection between the player and the game. This contains all abilities, rules, and restrictions which apply to both the game world and the player.

Definition 5:

Side-scroller

A side-scroller game is a video game where the player's camera is restricted to movement occurring mainly on the x axis (sideways). Most often, the screen movement is from left to right (or less commonly, right to left). Side-scrollers can be made with both 2d and 3d games.

Definition 6:

Unity editor / engine

When I write about the Unity editor, I mean the concrete application where developers can orchestrate their assets and programming code. By talking about the Unity engine, I mean the software as a whole. FAR: Lone Sails was built within the Unity editor and it runs with the Unity engine.

Definition 7:

Framework

"Framework" describes a set of tools which can be used and extended by developers for their own purposes. I mostly use the term when discussing visual tools within Unity or programming libraries.

Definition 8:

Indie games / developers

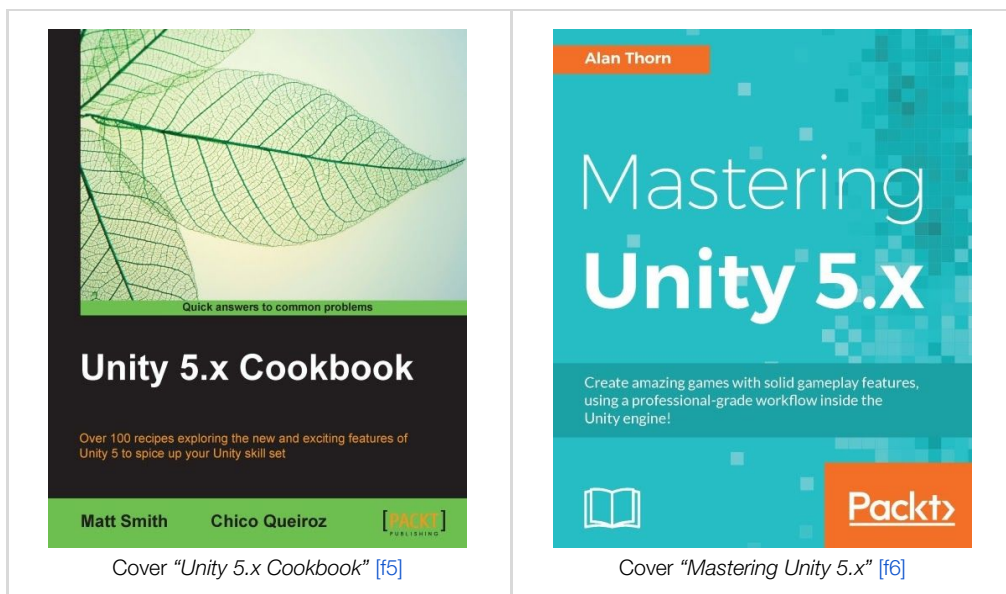
Indie games are video games created by individuals or small teams, usually without significant financial support from outside sources. [8] In my opinion not every small studio is an "indie", especially once they have launched a financial hit and are able to cover the development costs for their next title.

2 Research

This chapter contains several approaches I used to research project structures within Unity. I first went through popular literature and internet sources in order to analyze existing Unity production strategies. This section is then followed with a second sub chapter that recaps a past meeting between myself and local developers, wherein we discussed ways to establish workflows not following the causal Unity way. The chapter ends with interviews I did with individual developers and studios with my corresponding conclusion of the results. Additional research is included within separate chapters in order to stay coherent with the context and to maintain the readability of the document.

2.1 Review of the literature

Since Unity has become one of the most popular game engines for indie developers, literature can be found on the subject all across the internet and in bookstores. For my research I went through the most up-to-date and popular books I could find online. Specifically, I took a closer look at “*Unity 5.x Cookbook*” [9] by Matt Smith and Chico Queiroz and, and “*Mastering Unity 5.x*” [10] by Alan Thorn.



As mentioned already in my abstract, most literature I found covered a “*from beginner to expert*” approach, or else went far deeper into specific game development related topics. The same applied to these two popular books.

Even in other books, I was unable to find any chapter about project structures and asset organizing related fields. Greg Lukosek writes about folder hierarchies in the project window in his book *“Unity 5: Learning C# by Developing Games”* and Lukosek gives examples of good and bad practices. [11] However, the chapter is just a small part of the book and doesn’t offer sufficient explanation of these complex ideas.

The most helpful information I could find were discussions on blog posts and on Unity’s official forums. [12] Piotr Korzuszek, a developer from a group called *“The knights of Unity”* wrote in one of his blog posts:

“I saw a person on Quora the other day, asking how programmers are able to write projects that consist of over 10,000 lines of code. When software gets bigger, it is more difficult to maintain and that’s a fact. So here’s the thing – if you don’t keep your project organized, you’re going to have a hard time to keep the pace. Later on, you will find yourself wasting time with a messy project instead of adding new features. This is also true regarding any Unity Project.” [13]

In his blog post, he also mentions seven ways to keep a Unity project clean. In another blog post by Herman Tulleken, he gives a specific example for hierarchy structures in a scene. [14]

Several developers discussed best practice strategies for folder structures on a forum thread between 2010 and 2017. [15] In general all agreed that it made sense to structure assets by type or by context / relevance. One developer with the nickname *“Eric5h5”* mentioned that the structure always depends on the project one is working on. I definitely agree with this opinion. If your content is very level or scenery related, it probably makes sense to separate all assets within this context. If not, it is easier to align assets by type.

All these recommendations are mostly based on the personal experiences of the authors. This is because there is no one right scientific way to work with Unity, as each game comes with its own needs. What helps most is to know the different methods possible within Unity, and to learn from the practices existing developers have established.

2.2 Not the Unity way

“The Unity game engine is an incredible powerful and productive tool for game development. However, for more complex game production than prototyping the tutorials and some other core concepts can be error prone, not performant and hard to manage.

In this meetup session, we want to discuss the benefits of code-centric development in general and, more specifically, the model-view-controller pattern and dependency injection.

This is the perfect opportunity if you have suggestions and experience in the topic or if you are eager to improve your Unity skills in that regard.” [16]

This little introduction prefaced an email invitation I’d received in November 2016. The Zurich based game studio Gbanga had invited local game developers to have a talk about different approaches when considering code related design patterns within Unity. During the event, Gbanga’s lead programmer Robin Bornschein presented their workflows and strategies for a model-view-controller (MVC) [17] and a dependency injection [18] framework. It was interesting to see that a lot of the present developers agreed that one should avoid working with Unity’s MonoBehaviour scripts as often as possible. Reasons given surrounded the old structure which MonoBehaviour scripts are built on, and the limitations that inherently come with them. Those in attendance agreed that it is hard to establish a good way to code proper object-oriented programming (OOP) [19] in Unity. All of the reasons to avoid Unity’s component based pattern I heard that evening initially seemed legitimate.

Yet after thinking twice about the different approaches, I realized that none of the present developers utilized the scene editor for their level crafting. Their games were mostly procedurally generated or UI-centric experiences. Their levels were created within text files or other self-written level editor tools. This was the point where I realized how heavily FAR: Lone Sails depends on the features of Unity’s scene editor. My team has to juggle with meshes, lights, and other components all the time in order to stage a scene and to make it look beautiful. To keep the runtime performance of the game in an acceptable range, we could not discount scene editor features like light-baking or occlusion culling.

Of course some of these above mentioned features can be implemented with procedurally generated content, but the workflow and effort required for using

this method grows with it. It would take us far too long to code a specific level-editor for our needs that could offer even a part the flexibility of the scene editor. In the beginning of FAR: Lone Sail's production we were not sure how deep customization on specific level parts would effect things, and if the level designers would be able to set elements autonomously. There are definitely a lot of advantages if a team decides to develop their own level editor, but the costs and the time efforts are too high for most small indie studios. The scene editor forms a strong backbone of Unity and is likely a reason why even big companies have started to build games in Unity.

I thus knew I had to live with the scene editor. I also had to find ways to give our studio's coders, artists, and level designers a smart workflow and pattern within the scene editor that would keep the project both maintainable and scalable at the same time.

2.3 Interviews

In order to get an overview of the strategies and principles being used by other studios, I set up a survey containing organization-centric Unity questions. The main goal was to figure out if there were additional ways of organization that I hadn't considered during my research processes, and to gather a lot of different scenarios to cover these bases.

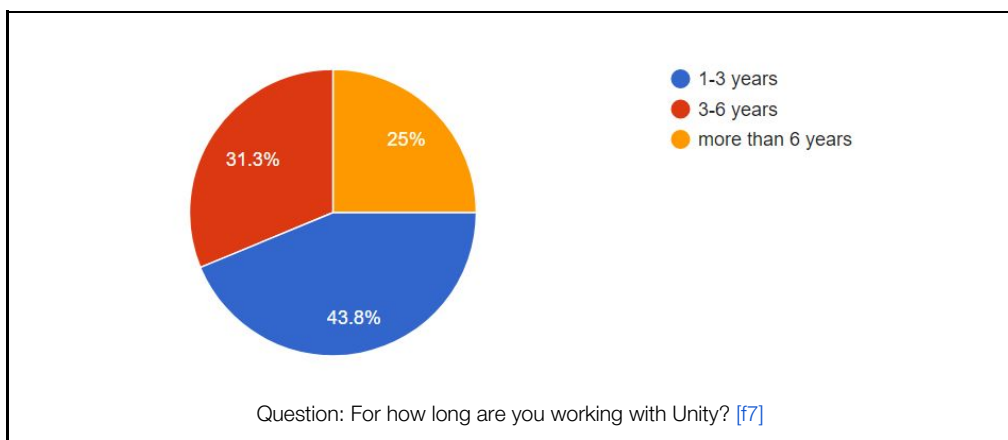
2.3.1 Survey

I used an interactive Google form with an introduction and several guided questions for my survey. I sent the survey specifically to the Sand Sailor Studio [20] from Romania and to the studio Blindflug [21] from Switzerland. The team behind Sand Sailor Studios is currently working on a 3d side-scroller game called "*Black The Fall*". [15] Blindflug is currently on Steam Early Access with their title "*Airheart*". [16] I chose them since I knew the two projects well. The team from the Sand Sailor Studios were twice our booth neighbours, once at E3 2016 and once at Gamescom in 2016. To also get some answers from projects and developers I did not know, I also posted the link of the survey on the social media sites Reddit, Facebook, and Twitter.

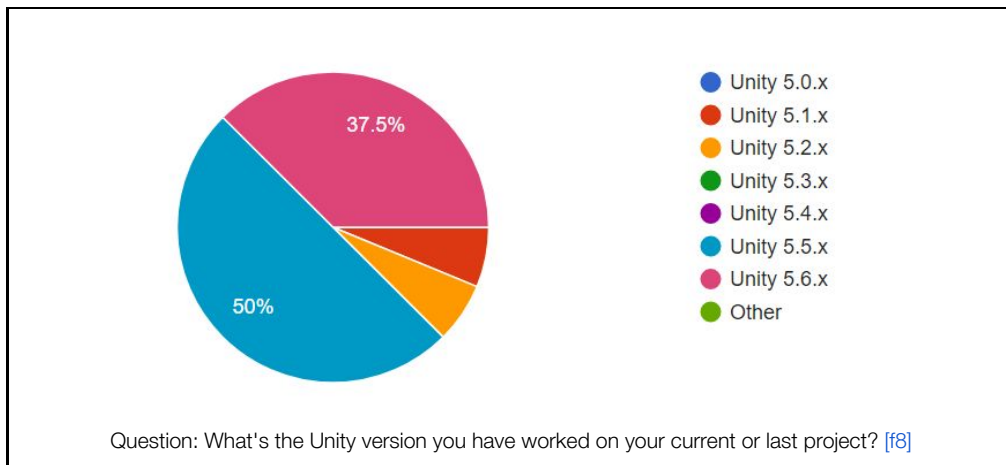
2.3.2 Results

In total, 26 developers filled out the survey. Most of the questions were answerable by text fields so that the participants had the possibility to explain their responses in greater detail. Besides technical questions, I also gathered information about their company positions and the development cycle of their current project. This part of the chapter contains a summary and a conclusion of my findings. The complete survey with all answers and charts can be found as an appendix at the end of this paper.

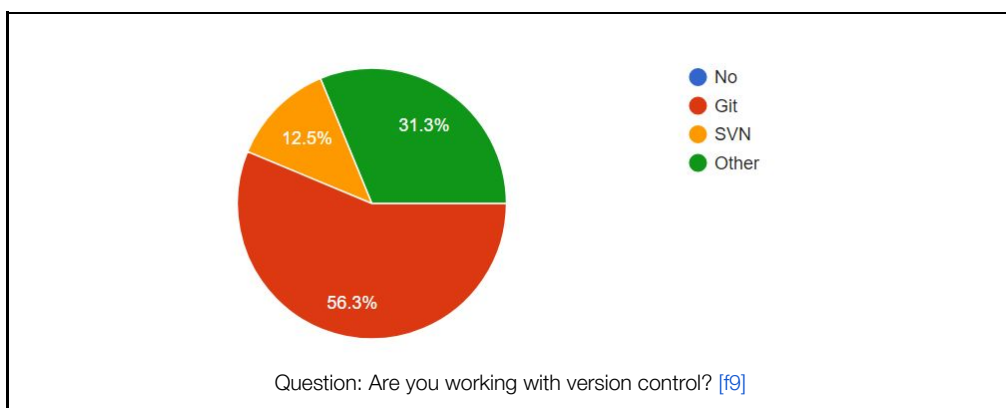
The majority of the questioned participants worked in small teams and had an average experience of three years working with Unity.



All participants had already switched to Unity 5 and half of them were already working with Unity 5.5 or one of its newer versions. This was definitely an indicator for me that some Unity developers are trying to catch up with the development cycle of the engine. One reason for this behaviour could be that it takes rather less time to update your project to the newest Unity version in small steps instead of big ones. Most small indie teams tended to avoid updating their Unity version once they had started their game production. This approach can help to prevent fatal and unexpected behaviour changes, enforced through engine updates, which are not well documented or aren't intended by Unity.



56.3% of all participants used Git as their version control system. Some participants who worked alone or in teams of two mentioned that they did versionized backups by themselves on hard drives.

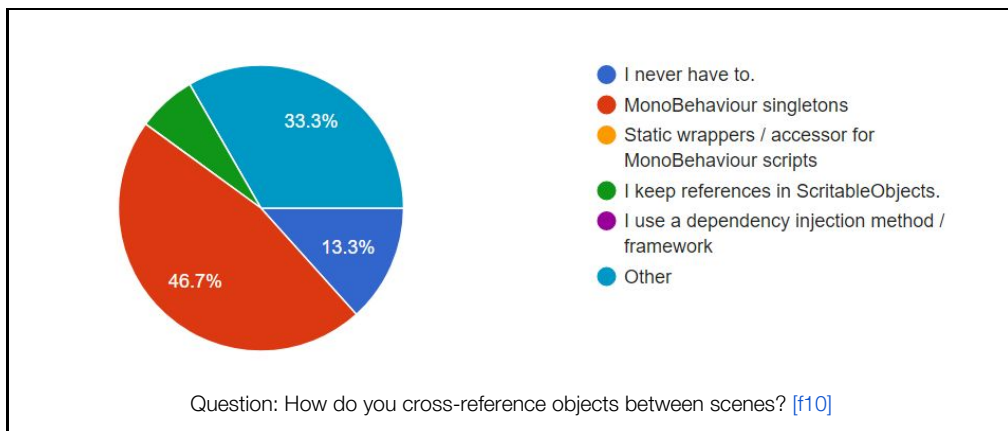


Almost all participants structured their assets by using fixed naming conventions. A majority used the so called “*lower camel case*” or “*pascal case*” [22] patterns to name their files and folders, where hierarchies are sorted by the type of object or by relevance. That means that some developers separated their content according to their level structures and others according to the type of a certain object. A more detailed explanation is described in the chapter 4.1 Editor content.

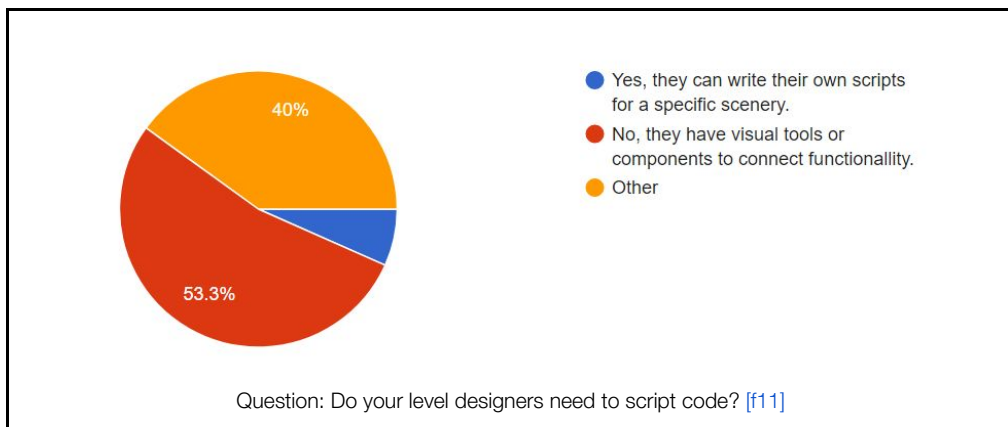
Two third of all participants already used Unity’s multiple scene editing features, which were introduced already with Unity 5.3. This statistic was surprising because the feature was barely documented for a long time and most developers couldn’t share a detailed experience due to its novelty. Additionally, 80% answered they loaded scenes asynchronously. By loading new content

asynchronously, developers need to separate their content by a certain logic and to guarantee object references between scenes.

46.7% answered that they established these cross-references through the singleton design pattern. An explanation of the pattern is described in the chapter [4.2.2 Managers](#).



In order to figure out how teams organize their discipline workflows, I asked the participants if their level designers needed to script any code. Interestingly a majority let their level designers work with visual scripting tools. 40% (represented by "Other" in the chart below) of the participants mentioned that they work with a mix between the first two options.



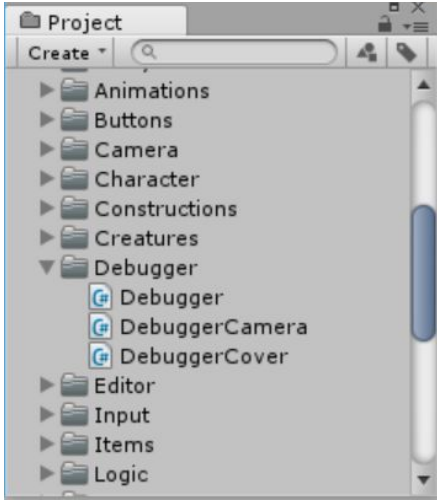
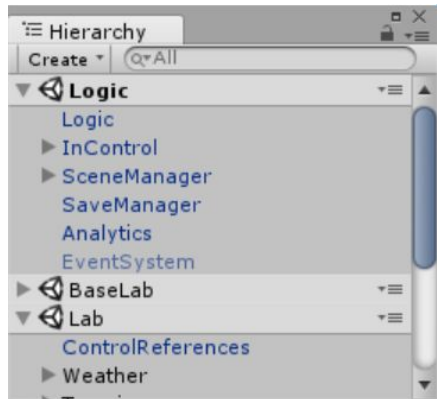
The results of this survey helped me to get an overview of the best practices being used by other developers so that I could use these findings to form my own conclusions and project implementations for FAR: Lone Sails. All further chapters are influenced by these results and rely on the information I gathered through this survey.

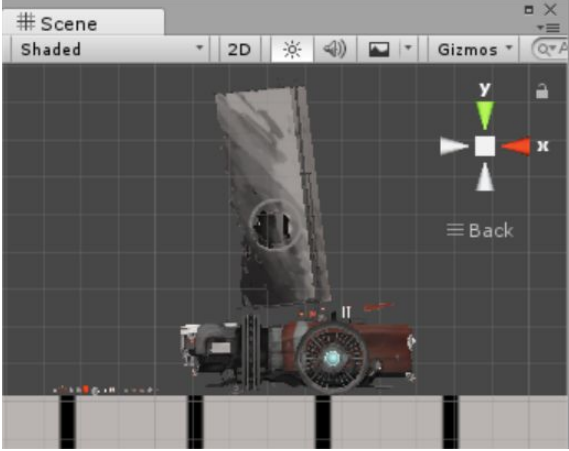
3 Basic knowledge

This chapter begins with a definition table for Unity specific vocabulary. The purpose is to clarify meanings of words like “*GameObject*”, since they can on their own be interpreted in various ways. If you already know these terms, it might still be helpful to read this section in order to understand how I used them in their particular contexts. The covered topics are related to editor-specific knowledge I needed to know when I implemented my strategies and structures for FAR: Lone Sails.

3.1 Vocabulary

The following definitions (explained through both graphics and text) are the meanings of technical terms I use throughout my thesis [t3]:

Project view / window [f12]	 <p>The screenshot shows the Unity Project view window. At the top, there is a 'Create' dropdown menu and a search bar. Below this, a list of folders is displayed with expandable arrows. The folders are: Animations, Buttons, Camera, Character, Constructions, Creatures, Debugger (expanded), Editor, Input, Items, and Logic. Under the 'Debugger' folder, three sub-items are visible: Debugger, DebuggerCamera, and DebuggerCover.</p>
Hierarchy view / window [f13]	 <p>The screenshot shows the Unity Hierarchy view window. At the top, there is a 'Create' dropdown menu and a search bar containing 'All'. Below this, a list of scene objects is displayed with expandable arrows. The objects are: Logic (expanded), InControl, SceneManager, SaveManager, Analytics, EventSystem, BaseLab, Lab (expanded), ControlReferences, and Weather. Under the 'Logic' object, several sub-objects are visible, including InControl, SceneManager, SaveManager, Analytics, and EventSystem. Under the 'Lab' object, ControlReferences and Weather are visible.</p>

<p>Scene view / window [f14]</p>	
<p>Asset</p>	<p>Describes a physical file in the project window. Folders are not considered as an asset.</p>
<p>Editor content</p>	<p>Editor contents are always assets.</p>
<p>Scene</p>	<p>A scene is a specific asset ending with the file extension *.unity.</p>
<p>Scene content</p>	<p>Everything that can be selected within the hierarchy window.</p>
<p>Scene editor</p>	<p>This is a synonym for the scene window and its functionalities.</p>
<p>GameObject</p>	<p>An object in the hierarchy window.</p>
<p>Prefab asset</p>	<p>A prefab asset is a GameObject template stored in the project window.</p>
<p>Prefab instance</p>	<p>A prefab instance is a GameObject in the hierarchy window that is linked with a prefab asset.</p>
<p>Blank GameObject</p>	<p>A GameObject in the hierarchy window which is not linked to a prefab asset.</p>
<p>Transform</p>	<p>Describes Unity's default transform component, which is on every GameObject. It keeps the 3d coordinates and information about a GameObjects's rotation and size.</p>
<p>Collider</p>	<p>Describes Unity's default collider component, which can be attached on GameObjects in order to calculate physical collisions. It can have various geometrical forms and sizes.</p>

3.2 Editor

Since the scene editor is one of the most important utilities for an indie team I will next be describing some important parts of it in this chapter. In order to understand the following topics, it is necessary to already have a fundamental knowledge about the basic functionalities of the Unity editor.

3.2.1 Assets and objects

To get an organized project structure in Unity one must understand how the engine defines assets. The easiest explanation is that everything that is stored as a separate file on the harddrive in the project folder is an asset - materials, 3d meshes, sprites, audio files, scripts, prefabs, scenes, etc. Folders do not count as an asset by definition.

It is important to know this because developers of the Unity community often talk about “*objects*” on forums and other social networks. An asset is also treated as an object in Unity, but the engine differs between two different types of objects: Inside a Unity scene, developers work with so called “*GameObjects*”. GameObjects are never stored in a specific file for itself, rather they are always part of a scene or a prefab. Imagine you have a scene with three GameObjects inside. The GameObjects will be listed as an array in the scene file. Assets on the other hand always belong to their specific hard drive file as mentioned above.

So let’s break down some other types of assets in order to clarify their definitions [\[t4\]](#):

Assets include:	These types are not assets:
<ul style="list-style-type: none">● All C-Sharp files● ScriptableObjects● Terrain data files● Audio files, 3d models, Textures, Sprites, etc.	<ul style="list-style-type: none">● GameObjects● Components● Transforms● Rigidbodies● Colliders● Light settings (they are stored in the scene asset!)

3.2.2 Prefabs

Prefabs may test (or destroy) your love for Unity. The idea of having predefined GameObject templates is trivial and is already in every game designer's mind. Their functionality will improve your workflow for placing similar GameObjects in the scene editor, or instantiating them on runtime. However, Unity is not supportive of nested prefabs right now, and this is probably the most in-demand feature from the Unity community. Nested prefabs would allow developers to connect prefabs hierarchically between each other. Imagine you had a template within a template and when you changed the subtemplate, the changes would apply on all subtemplates. This is unfortunately impossible right now.

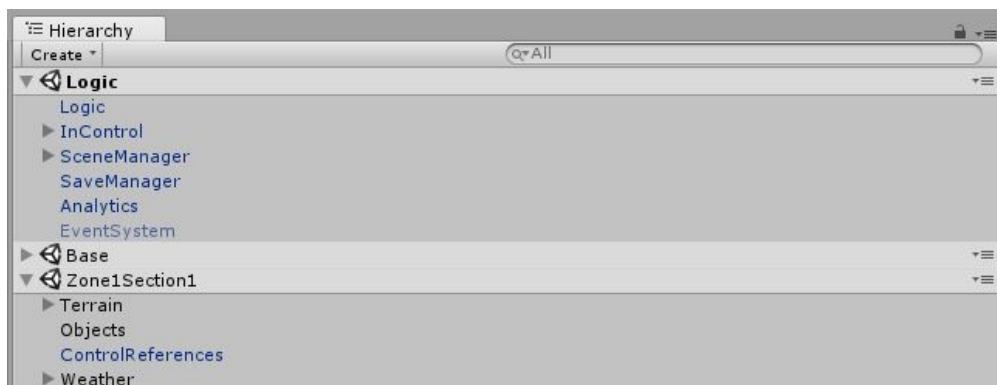
In good news, the Unity development team this year finally announced the feature as being on their future development roadmap. [\[23\]](#) There are third-party tools on Unity's asset store currently offering nested prefab functionality, but I personally have never tried them due to negative consumer feedback and performance throwbacks. There are developers who have managed to ship a finished product even without nested prefabs, but I have also met some developers trying on their own to implement self-made complex nested prefab functionalities. If you ever need a complex prefab structure in your project, it will probably be easiest to solve the problem by duplicating and populating multiple instances of prefabs manually. Just be aware of the default behaviour of your prefab when you revert them. That means: manipulate, populate, apply, and revert data from prefabs. You should know how these things will behave in your environment before your team starts the production of your game.

Another important piece of information some developers don't know is that if you create a prefab of a GameObject, all serialized data of each attached component will move from the scene asset to the prefab asset. The scene asset itself will only hold a reference to the newly created prefab and will store only unapplied values of the prefab in the scene asset. As soon as the unapplied values get applied, they also move to the prefab asset. To know this structural behaviour can help to find problems involved with prefabs and save game systems.

3.2.3 Multi scene editing

Since Unity 5.3 it is possible to have multiple scenes loaded in the scene view of the editor. [24] This was one of the most amazing new 5.3 features for me, as it was one I had always wanted while working with past versions of Unity. It also has changed the way I structure my projects. I can now drag multiple scenes into the hierarchy window and all GameObjects of these scenes will be shown in the scene view, where I can separate my content within multiple scenes. Likely the biggest advantage of this feature is that it allows developers to create multiple scenes overlapping each other at the same position of the 3d world. By doing this, developers can create a scene file for each discipline.

For example, one could create a scene for audio placements, one for visuals, and another one for logics. This allows a team to work together in different scenes at the same position in a level without ending up having unsolvable merge conflicts between scenes with version control. Another advantage is that you can load and unload these scenes additively in the background. Memory resources can be reduced if you only load content the player can actually see. In FAR: Lone Sails I established a scene system that loads new chapters and unloads old ones seamlessly, so that I could keep the memory footprint small, and so the game never had to blend in any loading bar during the gameplay. This can be great for creating an immersive experience, and open world games are usually using this sort of technique.



Example: multiple scene editing in the hierarchy window. [f15]

As mighty and handy as multi scene settings might now sound, the technique has one big disadvantage; you can not cross reference GameObjects and components between scenes. However, there are workarounds to establish cross scene references. I will describe the system I implemented for FAR: Lone Sails later in chapter 4.2.2.Managers. Another disadvantage is that Unity also

only allows you to have one “active” scene. The active scene is always highlighted bold in the hierarchy window and the light settings of this scene will always be applied to the game. Because of this there are no native possibilities to make a smooth transition between scene light settings, and you have to handle this yourself if you ever need to switch active scenes.

The multi scene features of Unity seem to be pretty limited at present as one often has to wrap up things oneself in order to get things working. There are options of using third-party tools on the asset store, tools which are doing these tasks at a more advanced level. I recently watched a GDC talk from the makers of “Firewatch” (2016) [13] that dealt with this subject. [25] Campo Santo, the company behind the game, used a tool called “SECTR” to achieve seamless content loading. [26] It might be worth taking a look at both Unity’s multi scene editing feature and third-party tools before you decide to use asynchronous scene loading. Take time to figure out how advanced it has to be for your needs.

3.3 Scene management

3.3.1 Unity scenes

In this sub chapter I will describe how Unity is managing scenes internally. This knowledge can help to both structure your scene assets and to also give hints which are useful for a save game implementation.

Each scene in Unity is represented in the project view as an asset and is saved on the harddrive as a *.unity file. If you synchronize your project over a version control system, you have likely set Unity’s asset serialization mode to “Force Text”, since binary serialized files aren’t supported by most version control systems. [27] If so, your scene files will contain a “Yet Another Markup Language” (YAML) structured text file with all of its content in it. [28]

```
m_sceneInLightmap: 0.0250
m_LightmapParameters: {fileID: 15203, guid: 0000000000000000f000000000000000, type: 0}
--- !u!4 &1659073185 stripped
Transform:
  m_PrefabParentObject: {fileID: 4000010444805044, guid: 44fd9c4bb019843baba8905af7d50114,
    type: 2}
  m_PrefabInternal: {fileID: 245013662}
--- !u!1001 &1685756248
Prefab:
  m_ObjectHideFlags: 0
  serializedVersion: 2
  m_Modification:
```

Example: YAML structured text content of a *.unity file. [f16]

Every `GameObject` inside a scene is represented in the scene file with a unique id and with all its components attached to the `GameObject`. Prefabs and other assets are linked with a so called “*Guid*” [29]. The *Guid* represents a globally unique identifier. Unity guarantees that these identifiers are always unique and that the engine is referencing assets with them. Even the scene file itself has a unique *Guid*. The *Guid* of each asset is written in a meta file, which automatically gets created for every asset. The meta files are always stored at the same location on the harddrive along with the asset and have the same naming with the exception of a `*.meta` file extension.

In FAR: Lone Sails we encountered a problem where copy pasting existing scene files sometimes didn't generate new meta files for the new scene. After synchronizing those copy pasted scenes via version control, we figured out that references to the meta files of these scene files were broken on other computers. It took us quite a while to solve this problem but our knowledge about the internal asset handling of Unity helped us to understand the problem and to find solutions for it. To get an even a deeper understanding of Unity's serialization procedure, I recommend reading their developer blog post called “*Assets, Objects and serialization*”. [30]

3.3.2 Asynchronous scene loading

Unity allows for the loading and unloading of content asynchronously. [31] This is a fantastic feature if you have big worlds and levels and only a small amount of memory. With asynchronous scene loading we can achieve a seamless game experience for players without any loading bars, but it also forces us to split up the content into multiple scenes and to accordingly structure the way we work. There are still some problems with this which I want to explain shortly in the next two sub chapters.

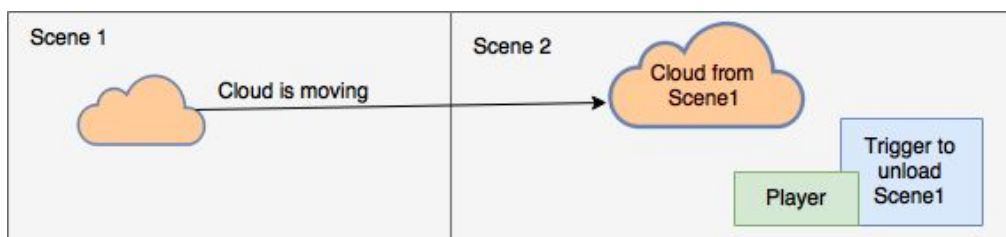
Hiccups / frame rate drops:

If asynchronous loading is happening, Unity utilizes a running background thread to get the content. This works only if the target platform supports multithreaded applications. In a best case, no visual hiccups or frame rate drops happen when new scenes get loaded additively. But if the new scene contains `MonoBehaviour` script, it will call the “*Awake*” and then the “*Start*” method upon its initialization. If a lot of new loaded `MonoBehaviour` scripts are doing heavy calculations during the initialization process, there may be a noticeable hiccup when the scene gets loaded. One way to avoid this is to spread the calculations over multiple frames during initialization, in order to optimize the calculations or to reduce the amount of scripts in that scene.

Visability

Another difficult challenge is to establish a system where the player will never see old assets disappearing or new content popping up during gameplay. This is especially hard for 3d games where players can manoeuvre the camera quickly and towards any direction. In FAR: Lone Sails it was not such an issue for us since the game is a 3d side-scroller. We did encounter problems with moving objects like clouds. If a cloud moved out of its scene borders together with the player into a new scene, and then the unload sequence happened, the cloud just disappeared. In this case we moved the trigger further in order to give the cloud time to disappear before unloading its scene. It is also possible to move a GameObject to another scene with the method:

“SceneManager.MoveGameObjectToScene”.



Example of the cloud example described above. [f17]

It might help to find smart strategies and ideas like this by watching the GDC talk from Campo Santo I mentioned in chapter [3.2.3 Multi scene editing](#).

4 Organize your project

This chapter shows ways to organize your editor and scene content. I first describe here how I implemented my strategies for FAR: Lone Sails based on both my research and my personal experiences. The chapter also offers concrete code snippet examples in order for the reader to better understand the chapter theory, and so that they can also copy the code for their own testing purposes. At the end of this section I define ten possible workflow rules useful to keep a solid project structure alive.

4.1 Editor content

This chapter covers content within the project view window. Scene specific content will follow in the next chapter.

4.1.1 Naming

Naming conventions within a team can be terribly hard to define. Often, team members have different naming habits which they already internalized a long time ago. Some have worked in companies and heard that there is only one “*right*” way of naming. In reality, there is no right or wrong way. One thing most developers would agree upon is to stay consistent with the chosen naming pattern.

The naming convention for FAR: Lone Sails:

I decided to name every asset in the project window according to the pascal case pattern [22]. Our studio also names folders in the pascal case style, but we do allow for spaces between words since that feels more like the nature of folders for most of us. One big advantage of using pascal case is that we also apply it to our C-Sharp scripts in order to follow Microsoft’s naming conventions. [32] In this way we can follow the same pattern for different fields. However, we have encountered a disadvantage by naming assets in pascal case style. When it comes to version control, merge conflicts can arise if somebody commits an asset in lower-case letters and changes it afterwards to capital letters, or vice versa. If you name your assets with capital letters, make sure that your version control is not set to ignore case sensitive changes.

4.1.2 Folder hierarchy

By the summer of 2016 our game development team had grown to be four members, and we clearly had to better structure our editor content in the project window for the sake of organization. All of the team members had different ways of organizing assets and naming in Unity (and like with naming, there is no single right, perfect solution. It all depends on your game and your needs.) Our folder hierarchy accordingly changed quite a lot during the production phase as we needed to first find conventions that made sense to all of us.

My research showed that developers tend to build their folder hierarchies based on asset types, or based on context / relevance. We had previously first tried to separate content by levels, until we realized that we had more assets that would be used in multiple levels, versus assets that show up only in a certain level. The same applied for prefabs and other assets. Because of this we decided like the other developers to build our hierarchy based on asset types. The research had also indicated that a lot of developers started folder names with a number instead of alphabetization, as this can help break the default alphabetic order of folders by placing a number in front of it.

After developing FAR: Lone Sails for one year, we found the following folder structure to be an improvement, if not perfect:

Folder hierarchy in the project view window:

1. Content
 - a. Prefabs
 - b. Models
 - c. Textures
 - d. Sprites
 - e. Materials
 - f. Terrain
 - g. Fonts
 - h. Shaders
 - i. Sounds
 - i. Effects
 - ii. Music
 - iii. Mixers
2. Data Content
3. Gizmos
4. Lib
 - a. InControl
 - b. Google Analytics
 - c. Etc.
5. Scenes
6. Scene Setups
7. Scripts
8. Standard Assets

I will now briefly explain some thoughts about our folder hierarchies, but I will not go into detail for folders like “*Fonts*”, since they are fairly self-descriptive:

1.a. Prefabs

All prefabs are stored in this folder. Sometimes it makes sense to separate them inside the folder again. This might arrive if you have prefabs that only handle logic issues and are not part of the in-game world, for example.

1.c. Textures

The place for all texture atlases. If a texture is represented in only one 3d model, we keep the texture in the same folder together with the model. Otherwise we will always have to search for the right texture for a specific model, and vice versa.

1.e. Materials

Here we store the shared materials of all texture atlases and of our dummy prototype obstacles.

2. Data Content

All ScriptableObject assets are placed here. The intention is to separate the audio-visual content from data content. This is an old habit of mine and takes account of the model-view-controller programming design pattern. [17] The pattern basically suggests to keep data, control, and visuals separate in favour of easier iteration and extension possibilities.

4. Lib

Lib stands for library and all third-party libraries are stored here. Usually, when people buy or download assets from Unity’s asset store, the installer places all new assets into the root folder. This can get messy, especially if one is using a lot of third-party tools. Since Unity 5, it is quite simple to move these folders into subfolders. Just remember, if you have to update a library, move it back to the root, finish the update procedure, and then you can easily move it again back to your desired subfolder.

6. Scene Setups

Scene setups are files that define a multiple scene view hierarchy. I will explain them in the next chapter [4.1.3 Multi scene setup](#).

7. Scripts

Scripts are probably the hardest part when it comes to folder hierarchies. I have certainly changed the folder layout in my projects countless times and still been unhappy with the solution. My final structure follows more or less my class diagram. Anyone who has not made a class diagram for his or her code should at least try to create an overview of it. I use a simple tool called Draw.io to create my diagrams. [\[33\]](#) It is free and it works in-browser.

Once you have an overview of your code, it's much easier to create a folder structure out of it. Imagine you have an abstract class for a vehicle and that there is another abstract class for the health of a vehicle:

"Vehicle.cs" and *"VehicleHealth.cs"*.

Since I ended up with two files having the word Vehicle in the beginning of the name, I created a folder called Vehicle and moved the files there. Let's say the VehicleHealth.cs class has even more sub versions:

"VehicleHealthLow.cs" and *"VehicleHealthHigh.cs"*.

Now again, I would create the subfolder VehicleHealth inside the Vehicle folder. This helps to determine by game logic where your scripts could be stored. So all scripts which are logically connected to the abstract class Vehicle, will be inside this folder. Using this kind of naming convention for scripts does bring the disadvantage of long class names. That means you could easily end up with a class name like *"VehicleModuleHealthLow.cs"*. But in general the C-Sharp conventions suggest you to name your classes clearly, so while it might sound odd in the beginning, it will help in the long run.

I definitely suggest for every developer to have a standard structure, even if you are working by yourself. You won't be able to find a perfect solution and you should certainly not try to do that. But instead find a solution your team can live with, and even more importantly, one where you can easily find things within the structure.

4.1.3 Multi scene setup

In this chapter I will describe how to organize your scenes and how you can save multi scene hierarchies into an asset. In the previous chapter you likely noticed that there are two specific scene folders in our project window hierarchy. One is called “*Scenes*” and the other “*Scene Setups*”. In the folder *Scenes* the usual *.unity scene files are stored. In *Scene Setups* we store assets with scene hierarchy constellations. Normally, when you set up multiple scenes in Unity and you quit and restart the editor, your last opened scene hierarchy will be loaded again. But sometimes you need to have different hierarchy constellations. For example, imagine you have multiple parts in your game and you don't always want to load all your scenes at once in the editor. Instead you just want to have some specific scenes you want to work on. Rather than manually dragging your desired scenes into the hierarchy window all the time, you can save a specific constellation into an asset and load it later again.

Unity doesn't currently offer a built-in functionality for this purpose, but it is possible to extend the editor in order to get it working. This is the way that I implemented it for FAR: Lone Sails:

1. First of all, since this is an editor only function, all of the following scripts need to be stored in a folder called “*Editor*” somewhere in your project window.
2. The first script is a ScriptableObject, let's call it “*SceneCollection.cs*”:

```
using UnityEngine;
using UnityEditor.SceneManagement;

public class SceneCollection : ScriptableObject {

    public SceneSetup[] setup;

    public void SaveSetup() {
        setup = EditorSceneManager.GetSceneManagerSetup();
    }

    public void LoadSetup() {
        EditorSceneManager.RestoreSceneManagerSetup(setup);
    }
}
```

SceneCollection.cs code snippet. [\[f18\]](#)

Since we need to be able to save an asset with data in the project window, it is pretty handy to use a `ScriptableObject` for this purpose. If you don't know what `ScriptableObjects` are, have a look in the Unity manual. [34] The script basically only serializes the loaded scene hierarchy setup from the `"EditorSceneManager"` instance provided by Unity.

3. Now we need an editor script that allows us to create an asset from a `ScriptableObject`. Let's call this one `"SceneCollectionCreator.cs"`:

```
using UnityEditor;
using System.IO;

public class SceneCollectionCreator {

    [MenuItem("Assets/Create/SceneCollection")]
    public static void CreateCollection() {

        SceneCollection asset = ScriptableObject.CreateInstance<SceneCollection>();

        string path = AssetDatabase.GetAssetPath(Selection.activeObject);
        if (path == "") {
            path = "Assets";
        } else if (Path.GetExtension(path) != "") {
            path =
path.Replace(Path.GetFileName(AssetDatabase.GetAssetPath(Selection.activeObject)),
"");
        }

        string assetPathAndName = AssetDatabase.GenerateUniqueAssetPath(path +
"/NewSceneCollection.asset");

        AssetDatabase.CreateAsset(asset, assetPathAndName);
        AssetDatabase.SaveAssets();
        AssetDatabase.Refresh();
        EditorUtility.FocusProjectWindow();
        Selection.activeObject = asset;
    }
}
```

SceneCollectionCreator.cs code snippet. [f19]

This script will add a menu point in the `"Create"` menu of the project window. The menu will be extended with an option called `"SceneCollection"`. By clicking on this option, the script will automatically create a `SceneCollection` asset in the selected folder.

4. In the last step, we will write a custom inspector for the SceneCollection asset. It will only contain two buttons that will allow us to save the current hierarchy and to load the selected collection. Let's call this script "SceneCollectionInspector.cs":

```
using UnityEngine;
using UnityEditor;

[CustomEditor(typeof(SceneCollection))]
public class SceneCollectionInspector : Editor {

    private SceneCollection collection;

    public override void OnInspectorGUI() {

        collection = (SceneCollection)target;

        if (GUILayout.Button("Save current hierarchy")) {

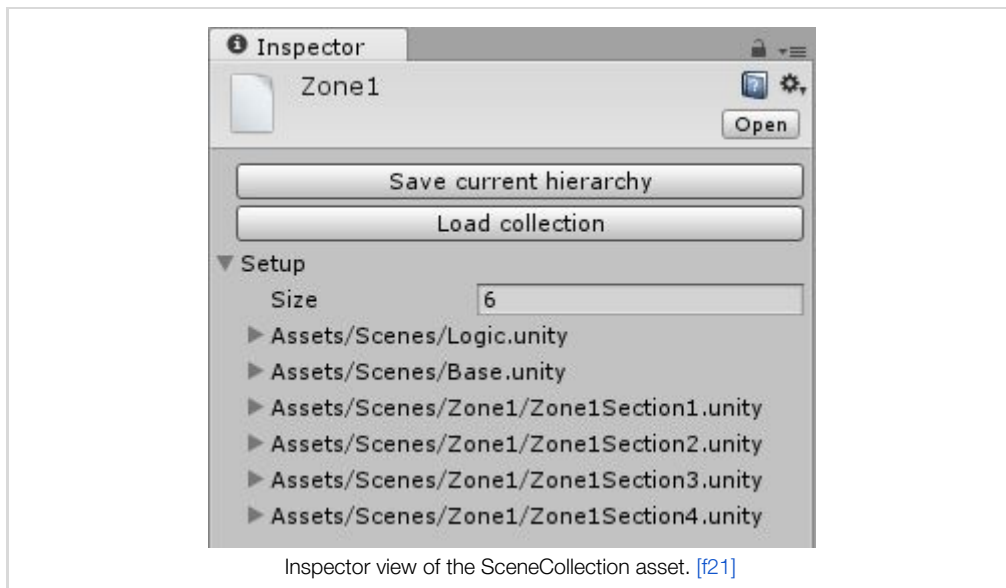
            if(EditorUtility.DisplayDialog("Save new hierarchy", "Do you want to
override this hierarchy collection?", "Yes", "Nope")) {
                collection.SaveSetup();
                AssetDatabase.Refresh();
                EditorUtility.SetDirty(collection);
                AssetDatabase.SaveAssets();
            }
        }

        if (GUILayout.Button("Load collection")) {
            collection.LoadSetup();
        }

        // show default inspector property editor without script referenz
        serializedObject.Update();
        DrawPropertiesExcluding(serializedObject, new string[] { "m_Script" });
        serializedObject.ApplyModifiedProperties();
    }
}
```

SceneCollectionInspector.cs code snippet. [f20]

Now when we select a SceneCollection asset, the inspector will look like this:



Additionally, you will be able to see the serialized setup array which contains the scenes that are stored within this collection. **Important:** if you haven't saved your current open scenes and you hit the "Load collection" button, you will lose all your unsaved changes!

4.2 Scene content

This chapter covers strategies to organize content within a scene; essentially everything that can be modified and manipulated in the hierarchy window of Unity. This also includes methods and workflow structures for level designers. Furthermore, strategies and potential implementations for basic logic managers are described within this chapter.

4.2.1 Naming and hierarchy

In this chapter I highlight the possible ways to organize GameObjects. The methods to create hierarchy structures in the hierarchy view are admittedly pretty limited. We can separate GameObjects by putting them into different scenes and we can parent GameObjects to achieve a folder behaving structure. Based on my research interviews, I determined that most developers use this approach of empty uniform scaled GameObjects which act as folders. It's important to mention that a uniform scaling of 1 on all axes will help you to minimize misbehaviour on your child GameObjects and will help to reduce performance overhead while manipulating big hierarchies.

In the responses for my research interviews I found a variety of different naming conventions used for GameObjects. Again, there is no one right or wrong approach and a team should discuss and decide what makes the most sense for their development process. Most developers mentioned in the survey that they place important root GameObjects on top of their hierarchies. Some indicated that they described states on GameObjects when they represented a part of the level which still needs to be activated.

For FAR: Lone Sails I defined the following rules with my team:

1. Each GameObject is written in pascal case style if one word is not enough to describe the meaning or the function.
2. Folders are always empty GameObjects and scaled to 1 on all axes.
3. MeshRenderer and Colliders are always on separate GameObjects which share the same parent GameObject.
4. If a GameObject represents one function or visual, like a mesh or an animation, the name of the GameObjects start with its function name.

For example:

VehicleDoor - AnimationMoveDoor - ColliderDoor - MeshDoor
--

5. If more than two objects of the same kind are present in a folder, they should be sub parented again.

For example:

Actual state:	Target state:
VehicleDoor - ColliderDoor1 - ColliderDoor2 - ColliderDoor3 - MeshDoor	VehicleDoor - Colliders - ColliderDoor1 - ColliderDoor2 - ColliderDoor3 - MeshDoor

4.2.2 Managers

In this chapter I will describe how to implement so-called managers. The terminology does not really indicate what one would normally mean with the word “*manager*”. However, most people in the Unity community would probably describe it as a centralized controller in the form of a MonoBehaviour on an empty GameObject. A common scenario where a manager is often used is an input manager; A single instance of a manager class is handling all the inputs from the player and passing the information to a receiver or listener. Since developers tend to have a manager for unique functions within the game, they also tend to apply the singleton pattern to their manager classes.

“In software engineering, the singleton pattern is a software design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system.” [35]

While singletons can make sense in some cases, they can also cause a lot of trouble. By design singleton instances are globally accessible in a code base. Every object can access it from everywhere, and this can lead to difficult debugging procedures. After developing prototypes for almost five years in Unity, I never really managed to get totally rid of singletons. It sometimes makes sense to use them, and in the case of an input manager where one probably never wants to have multiple instances of it, it can help to structure your code. The problem is that developers also tend to overuse singletons and managers in general. This is mostly coupled with a wrong or non-existing understanding of OOP principles. [36]

Still, our team needs managers. I use them mostly for handling fundamental game logic like scene loading, saving, and loading game states, etc. I basically place all of my managers in a scene called “*Logic*”. This scene needs to be always loaded since all the game handling is happening through the managers. The question is, how can we access these managers from other scenes, since we cannot reference GameObjects over different scenes in the hierarchy window through public inspector fields? What if we need the input manager in a MonoBehaviour script, just to check if the player pressed the jump button? We could create a singleton for each manager and directly access the current instance in scripts. But what if we need another instance of a manager? Or what if we want to extend a manager?

I solved this problem with a static instance wrapper. This means a static class called *“API.cs”* is looking out for the right instances and linking them with static fields inside the class. I used this name because it acts like an internal application program interface (API) [37] for the core elements of the game. Consider if we have an input manager. If I want to access the input manager in a level script I can get it simply by calling:

“API.InputManager.IsJumping()”.

The accessor *InputManager* will represent a clean *MonoBehaviour* instance from the Logic scene. The static API class will handle the linking and the access by itself, so that the input manager does not need to have any intelligence for being accessible. We can achieve this behaviour with the following example script on the next page:

```

using UnityEngine;

public static class API {

    private static T FindSingleInstance<T> () where T : Object {

        if (Application.isEditor) {
            T[] result = GameObject.FindObjectsOfType(typeof(T)) as T[];

            if (result.Length == 0) {
                throw new System.Exception("API: can't find module " + typeof(T) +
                " in the scene!");
            }

            if (result.Length > 1) {
                throw new System.Exception("API: there is more than one " +
                typeof(T) + " in the scene!");
            }

            if (result[0] is T) {
                return result[0];
            } else {
                throw new System.Exception("API: there is a type mismatch with " +
                typeof(T) + "!");
            }
        } else {
            return GameObject.FindObjectOfType(typeof(T)) as T;
        }
    }

    private static InputManager _inputManagerInstance;

    public static InputManager InputManager {
        get {
            if (_inputManagerInstance == null ||
            ReferenceEquals(_inputManagerInstance, null)) {
                _inputManagerInstance = FindSingleInstance<InputManager>();
            }
            return _inputManagerInstance;
        }
    }

    public static bool PrewarmReferences() {

        if (InputManager) {
            return true;
        }

        return false;
    }
}

```

API.cs code snippet. [\[f22\]](#)

First we have a generic static method called *“FindSingleInstance<T>”* that will help us to find an instance of type T and to guarantee that there is only one of it in all the loaded scenes. We will only throw an exception in edit mode, since we can't see them in builds anyway. Then we have a private static instance of the input manager called *“_inputManagerInstance”*. A public static property called *InputManager* will check to see if our private static instance of the input manager is valid and exists, and if not, it will check, find, and apply it to the instance. Now when we call *API.InputManager.IsJumping()* our static API class will lazy load [38] the input manager from the Logic scene. If lazy loading is not an option due to hiccups during the gameplay or other reasons, one can also pre warm all references as soon as the game is loaded with the static function *“PrewarmReferences”*. Any additional manager can be placed in the API and if there is a new version of a manager, we can create a new *“APIv2.cs”* class or just add a new accessor to the existing API model.

Another way to handle dependencies across scenes more professionally are dependency injection frameworks. [18] Most dependency injection frameworks work with setup containers, where developers can configure the connection between single instances. This can help to keep an overview of links between instances and to help decouple single classes from each other. A very popular framework among developers is Zenject. [39] It works with most platforms Unity is supporting and there is a great community around the project.

Most dependency injection frameworks require an initial time investment to be understood. If you decide to work with one of them, be sure that your team is also understanding the way instances get linked together. These frameworks often make use of C-Sharp's reflection library, which can create a little scripting overhead and cause problems during an ahead-of-time (AOT) [40] compilation. Zenject for example supports Unity's IL2CPP [41] source to source compiler with some exceptions.

Scene manager

A scene manager handles all loadings and unloadings of scenes. This can happen asynchronously if the game needs to load additional level content, or synchronously if you load the game from a main menu screen for example. For FAR: Lone Sails I implemented a simple scene manager logic. It is basically a *MonoBehaviour* script attached to an empty *GameObject* in the *“Logic”* scene with a static access wrapper, as I described in the previous chapter.

In order to work as a core element of the game, the scene manager script assumes that the Logic scene is always loaded. In my scene organization, I separated the content that will always be present in the game in a second scene called “Base”. In the case of FAR: Lone Sails, the character and the vehicle are on the screen constantly. The rest of the content I split into zones and sections. Zones define a specific level theme and are only abstract limiters. Each section of a zone is 1km long in Unity measurement and is a single Unity scene file. A zone can contain between four to eight sections. We named them “Zone1Section1”, “Zone1Section2”, and so on. The scene manager loads at the beginning of the game the Base and Zone1Section1 scene.

The following diagram can help to explain the present layers of the system:

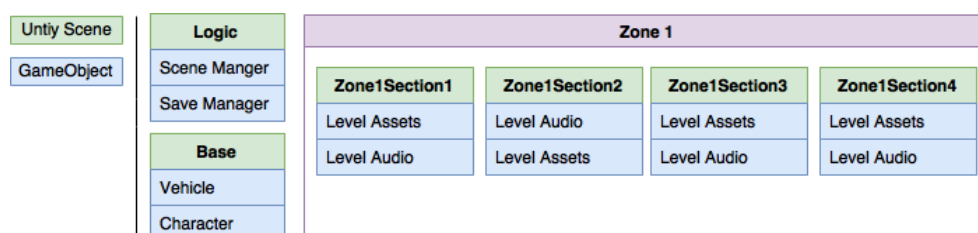


Diagram of the scene structure in FAR: Lone Sails.

In each section we placed character triggers which load the following section and unload the old section asynchronously. By doing it this way, we managed to only have two scenes loaded at the same time. With this technique we were able to reduce the memory footprint to the minimum considering asset and texture sizes. Having clustered level chunks in the form of scenes also brings advantages for a team working with version control. I will describe more about these advantages in chapter [5.2.1 Scene and prefab handling](#).

When loading scenes asynchronously a couple of problems can occur. First of all, you should have an overview for which of your scene background elements are getting loaded and unloaded. We had a problem where some parts of the decoration were being placed far in the background and when that specific scene got loaded, the decoration popped up out of nothing. To avoid this, it's important to know where to place your scene loading triggers and to check if objects will pop up or disappear when you load or unload scenes. Another tip is to have a smart trigger system. If you have a single point of entrance on the trigger, the player could abuse your scene loading system by walking quickly back and forth at that specific point. Depending on the elements you have in your scenes, the background loading thread can take time. So you definitely want to avoid having the player doing this, either by purpose or on accident.

Save game manager

The save game manager is the logical instance that handles all requests to load existing save game points, or requests to create new save game points during the gameplay. The manager knows exactly which states and objects need to be saved and how to recreate or to load them again.

To accomplish something like a save game feature it is necessary to understand the serialization structures of C-Sharp and Unity. Most developers underestimate the complexity of save game systems and try to implement them in late states of their production. I definitely suggest to get at least an idea about possible ways to save your game state as early as possible. All save game systems require serializable data sets of your logical objects, and it can affect your code base massively if you have to refactor everything to be able to implement the needed structures.

In the beginning, I assumed that I could just somehow serialize the whole scene and then load it again, and that it would be something like a memory dump of the actually running game written down in a file or a database. The problem is, that the memory footprint is pretty high on most games and it would take too much space and time to save down these multiple hundreds of megabytes or even gigabytes. Even if I would have liked to save my game this way, Unity does not offer any function to do this and it is also not possible to achieve this with blank C-Sharp. (When I think about this naive and rather foolish idea of saving an entire memory dump, I have to question all of the principals I learned during my software development apprenticeship time!)

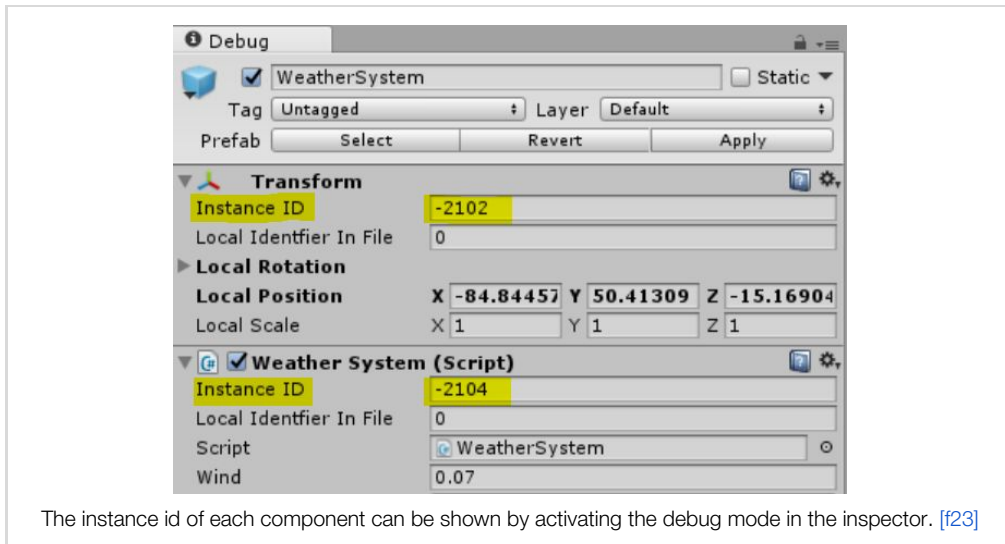
One right way to handle a save game system is to first check which states really need to be saved. Afterwards, programmers need to serialize these variables or data sets into a binary or text format and to save the information as a file or store it into a database. Most developers in the Unity community and third-party tools tend to use a JSON-Serializer [\[42\]](#) to achieve this. The advantage of the JSON format is that you can read and manipulate the values inside the serialized textfiles. By contrast C-Sharp's BinaryFormatter writes machine code which is not readable by a human.

For FAR: Lone Sails I used the open source library FlatBuffers from Google. The library is extremely fast, has almost no overhead, and is optimized for mobile devices. It brings several advantages over the classical approaches, like forward and backward compatibility, and it can access data without packing or unpacking it. It was in fact more than robust enough for our needs (I needed a

technical challenge and I was curious how it worked!) Since we will be releasing the game on Steam and consoles, we had no need for FlatBuffer's performance features. The implementation is much more complicated than with popular JSON-Serializers, and I would not recommend people to use FlatBuffers if they don't need its performance features.

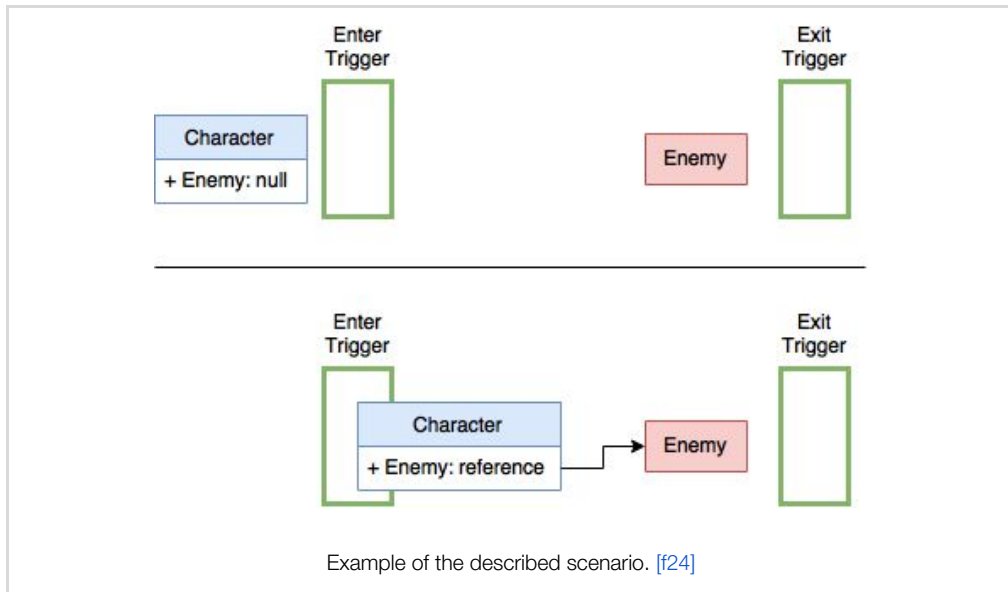
Beside the concrete serialization of the data, it is important to think about the logical functions you have to provide to your scripts which need to save states. Every object of your game that can be saved and loaded needs to get its proper state when you feed it with a saved state. Imagine you have a 30 second long animation on your character and you allow the player to save the game during the gameplay manually. You will need to save the state of the animation and to reset it to the exact position when the player loads it again. You could of course just ignore the fact that the player had this animation running during the save procedure and put it to a default idle animation, but you have to consider whether this choice could disturb the player or the gameplay. The same problem occurs with running coroutines during the save procedure, for example. These are only two scenarios which highlight what make save game features so complicated.

Another big issue comes with identifiers of GameObjects. When you load back your saved game data, you need to know to which GameObject in the scene the data belongs or which prefab you need to instantiate and populate with the given data. Unity itself tags each component (which includes MonoBehaviour scripts) with a unique instance id. The id can get called with the method "*GetInstanceID*" [43] of each class that derives from the base class "*Unity.Object*". The engine only guarantees that each id is unique, but not that it is persistent. That means, that if the scene gets reloaded during runtime, all the ids change.



If you want to tag a component persistently, you can mark it with a MonoBehaviour script that contains a serialized Guid [29] field. This solution works only with blank GameObjects and does not work with prefab instances. The reason is that serialized fields on prefab instances will always get overwritten by the prefab's asset value if the prefab gets reverted by a level designer. So you need to handle save game states differently between blank GameObjects and prefab instances. One might think it is easier to just have prefabs of all the GameObjects that need to be saved, but the problem with new instantiated prefabs is that the save logic also needs to manage all references to other GameObjects it holds on a MonoBehaviour script.

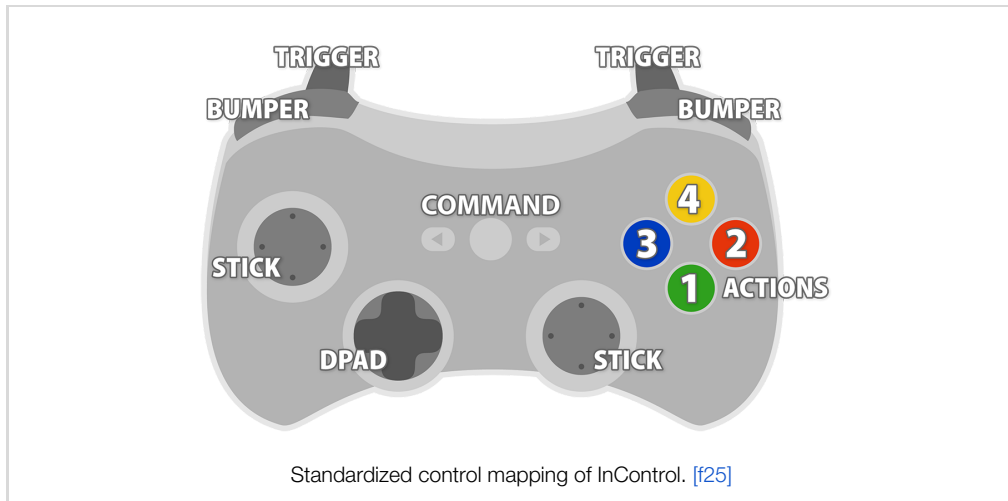
For example, imagine you have a character and an enemy in a scene. They are both prefab instances with different MonoBehaviour scripts, which control the behaviour and the save game handling. Additionally, there is an entrance and an exit trigger in the scene. If the character runs into the entrance trigger, the script starts to hold a reference to the enemy instance in a private variable until the character reaches the exit trigger. If the player now creates a save game before the character reaches the exit trigger, you need to take care of that reference to the enemy instance. If you don't, the character will not know about the enemy when that specific save game gets loaded. It makes it even more complicated if the reference object is a prefab that needs to be instantiated as well during the loading process.



This above example of course also contains an extremely bad trigger handling design and there are better ways to solve this problem, but the same issue occurs when you are working with serialized UnityEvents [44] and you crosslink other instances of the scene over the inspector. I definitely recommend to read Lucas Meijer’s Blog entry “*Serialization in Unity*” in order to understand how Unity instances and their serialization processes work internally. [45]

Input manager

An input manager should help you to handle and to control the flow of all player inputs. The idea is to have a centralized point where all input related scripts can check if a player input happened. Unity has a built in input API which covers most basic features for a proper input manager. [46] For FAR: Lone Sails we used the third-party library “*InControl*” by Gallant Games. [47] InControl extends the existing input API of Unity with standardized device mappings. This means that it offers a defined model of buttons and triggers which are used by most controllers, and developers can use this model as a reference point. The library maps these references automatically and you don’t have to worry if an Xbox or a Playstation controller is being used to play your game.



An input manager often comes along with the command design programming pattern. The command design pattern creates objects that can be applied to actors or other objects in order to perform an action. A popular group of software developers called “Gang of four” described the pattern as a way to “encapsulate a request as an object, thereby letting users parameterize clients with different requests, queue or log requests, and support undoable operations.” [48]

If you are more interested in the concrete implementation of the pattern, read Bob Nystrom’s helpful chapter about the usage of the pattern in combination with player inputs in his book “Game Programming Patterns”. [49]

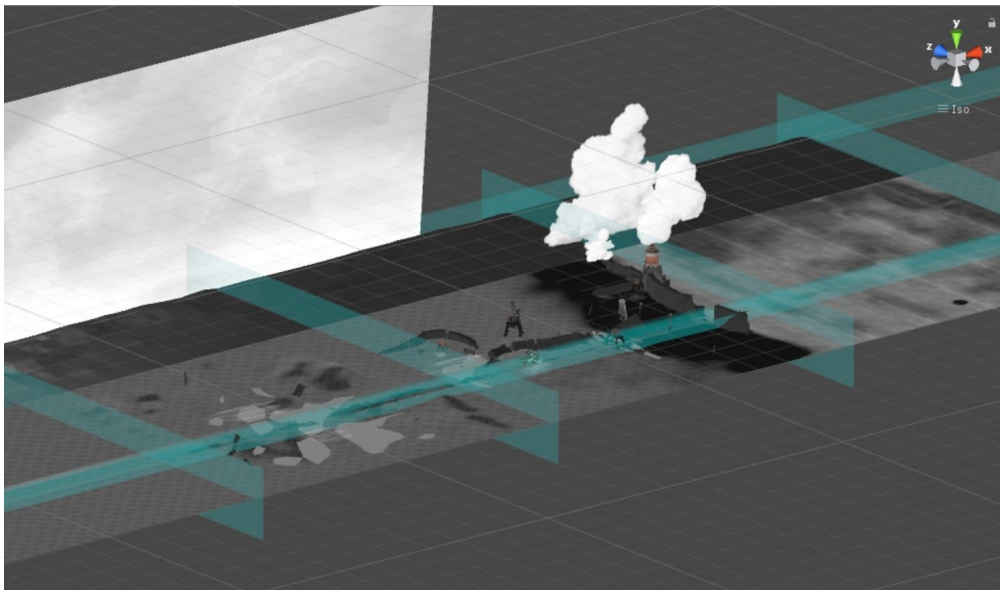
Debug manager

Every developer knows the importance of having the ability to debug his or her game. In this chapter I would like to mention the debug possibilities available within the scene view, as well as some methods we used to analyze misbehaviours and bugs in FAR: Lone Sails.

For our game I implemented all debugging features into a centralized manager. By having this feature centralized, it is easier to deactivate or remove debugging features for final builds, or to remove them if you don’t want them to interfere with the game temporarily. Before I started to develop our internal debug manager, I asked my team how they work in the editor and the scene view. This was important to do since level designers especially face troubles when it comes to detecting misbehaviors. To know how your team is working can give you a clue about the tools needed to improve their workflow. This will in turn will save time for the whole production.

One level designer had to test specific parts of the game world and to do so he needed to move the main vehicle of the game, the character, and the camera to a certain position. Since these three GameObjects are root objects in the scene hierarchy, he had to mark all the GameObjects before he could then finally move them. This is not a particularly big task, but he had to do it all the time, and he became frustrated from having to select the GameObjects over and over again. In response, I wrote a keyboard shortcut function to move these three GameObjects together to the current position of the mouse cursor. All objects were additionally positioned to the value 0 on the z-axis, since this was a defined game rule anyway. This simple keyboard shortcut helped to reduce the time it took to select the GameObjects and reduced an error-prone behaviour, where the z-coordinates could be moved accidentally away from 0 during the movement process in the editor.

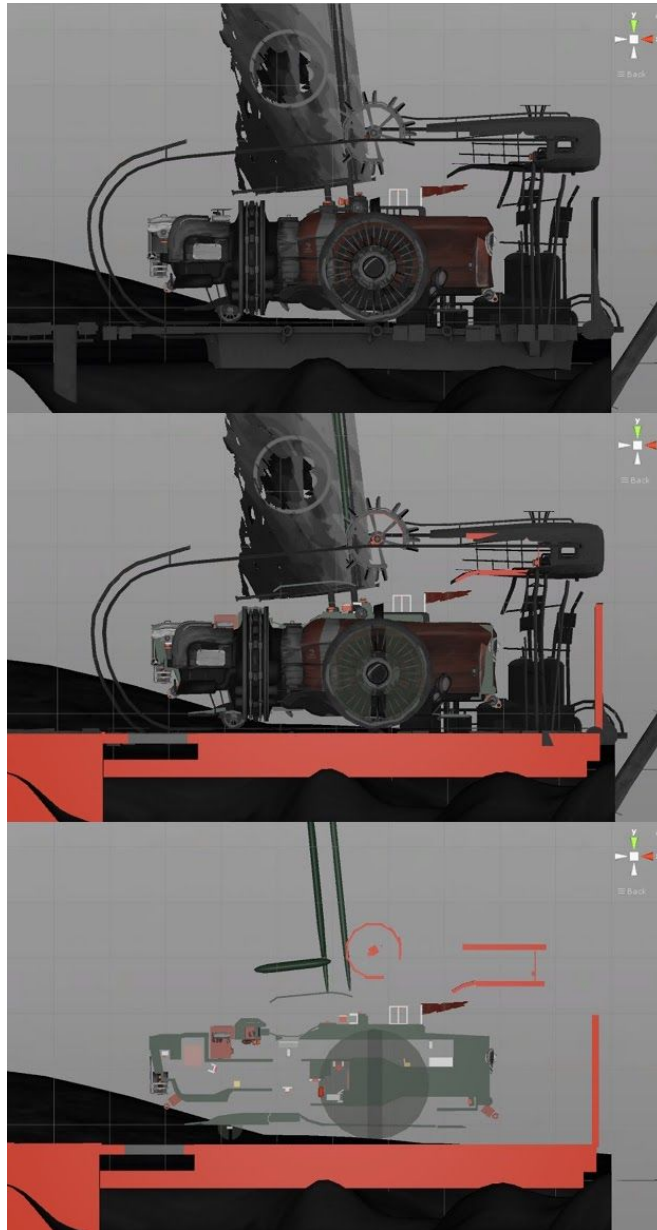
Another keyboard shortcut I made showed the section borders of the scenes in the game. This helped my developers to figure out if a GameObject should be placed in the previous or next scene, and it helped them to know where to place the triggers to load and unload scenes.



Debugger: keyboard shortcut that helps to visualize the borders of each scene / section. [\[f26\]](#)

The debug manager also contains marker functions and visual mode changers for the scene view. Our game uses markers to visualize directions or hits of raycasts, for example. Since our own written raycasts are pretty abstract programming code, these simple visualizations can help us to find incorrectly

calculated directions or positions in the 3d world. In FAR: Lone Sails we avoided working with MeshColliders completely due to their negative performance. Instead we placed primitive Colliders within our meshes. The visual mode changer feature allows the level designers to switch between mesh, Collider, and mix mode within the editor. This can help them to place Colliders among the meshes and help them to detect where exactly the characters or objects will collide, without being distracted by meshes. I describe the implementation of these debug features in chapter [5.1 Editor extensions](#).



Visual changer feature: 1. picture: mesh mode, 2. picture: mix mode, 3. picture: collider mode. [f27]

4.2.3 Level design management

This section is about tools for keeping a maintainable level design workflow. Some questions it deals with are - How can you achieve a generic logic for your level designers without writing a script for each individual element of your game? Are there visual scripting tools I can use for this purpose? I would like to give some answers to these questions, and to present the way we fulfilled the needs of our level designers while working on FAR: Lone Sails.

Closed ecosystem

If programmers want to encapsulate their core elements in a game, they should definitely do it. If your core gameplay is based on the movement of a main character, the character scripts should not contain any relevant public variable or method that can disturb movement. If you have critical parts of your core game elements, protect them. There are several ways to do this, but a good communication streamline between team members is still the best way to prevent fatal code accessing critical core elements.

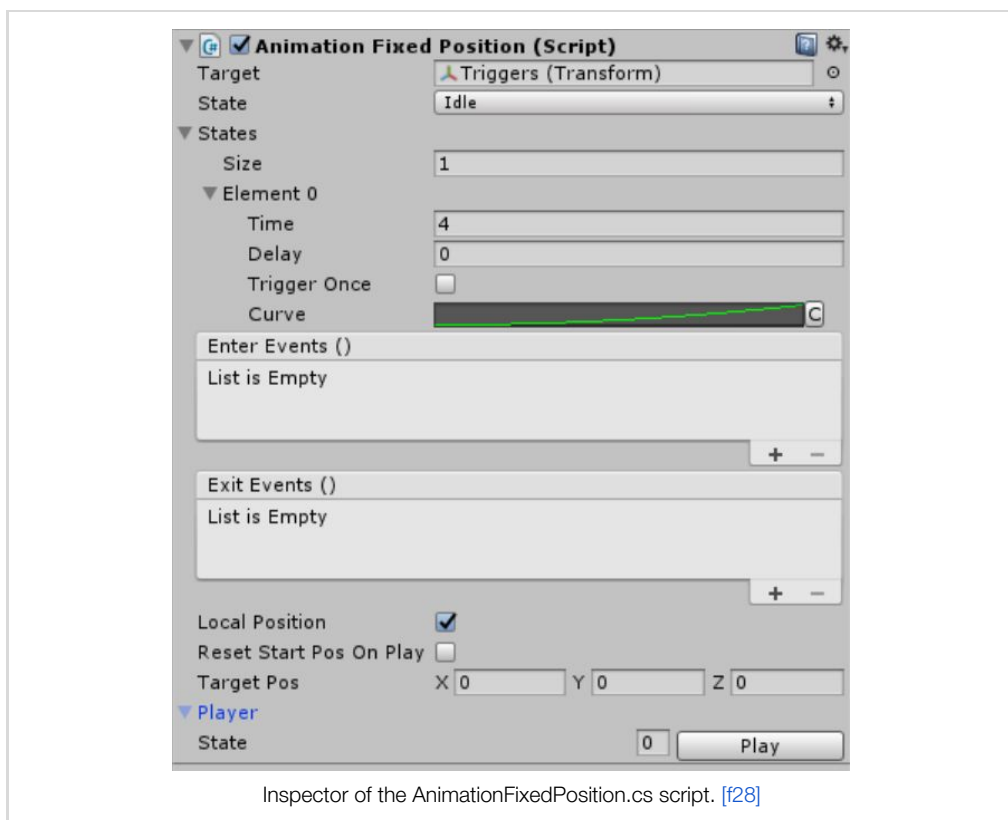
In FAR: Lone Sails I built an extra internal API for level designers. Each core element of the game has its own control class that wraps up level design related functions. For example, we have a script called *“Vehicle”* covering all the logic for the main vehicle in the game. This class offers a method called *“ReleaseBrakes”* which allows us to release the brakes of the vehicle. Instead of referencing the vehicle directly in a level design script and then calling the method, level designers first have to attach a MonoBehaviour script called *“VehicleControl”*. This specific control class offers the same method and forwards the request to the concrete Vehicle instance. The benefit of this extra layer is that lead programmers can set up specific access rules for level design calls. Additionally, the control classes may implement dependency injection methods. In this way level designers can attach the control class to a prefab without being worried that references will get lost on other prefab instances. Methods from the control classes can also be called from UnityEvents and do not need direct referencing to the target object with this kind of bypass trick.

Animations

Animations in Unity can be achieved in multiple ways. One way is to import keyframe or bone animations within a 3d model. Another way to animate GameObjects can be achieved with Unity’s built-in keyframe animation system by manipulating physics over scripts. There are also third-party libraries which allow a developer to interpolate between positions, rotations, and sizes of

Transforms and other components with easing methods. The most popular ones among the Unity community are iTween and DoTween.

At this point it may be pertinent to mention Unity's AnimationCurve class, which can be exposed in the inspector and displayed with a custom property drawer. This property drawer lets developers modify a function curve easily within a bezier curve editor. The AnimationCurve object can then, for example, be used to define the interpolation curve of a lerp function within a coroutine. For FAR: Lone Sails I programmed my own visual animation system that can manipulate Transforms and other components. Essentially, level designers can take a specific animation class and define states and customize settings. These MonoBehaviour scripts also provide public methods to start, to stop or to transit into a specific defined state.



The figure above shows the AnimationFixedPosition component of my animation system. The first property gives the possibility to address a target Transform which should become animated. Then the initial state of the animation can be set between "Idle" and "Running". The next field is an array of possible states for the animation.

Each state of the component can be defined by the following settings:

- *“Time”*: The time it takes to perform the animation.
- *“Delay”*: If the animation should be delayed for a certain time.
- *“Trigger Once”*: An option if the state should only be triggerable once.
- *“Curve”*: An AnimationCurve which forms the interpolation of the lerp.
- *“Enter Event”*: UnityEvent for actions that will be invoked when the animation starts.
- *“Exit Event”*: UnityEvent for actions that will be invoked when the animation ends.

The section after the states form the settings of the animation in general:

- *“Local Position”*: An option to define if the target should be manipulated in local or world space.
- *“Reset Start Pos On Play”*: An option to define if the start position of the animation should be reset every time the animation gets played.
- *“Target Pos”*: The desired target position.

It’s possible to test the animation within the scene view by choosing first the state array number in the *“Player”* section and then hitting the *“Play”* button.

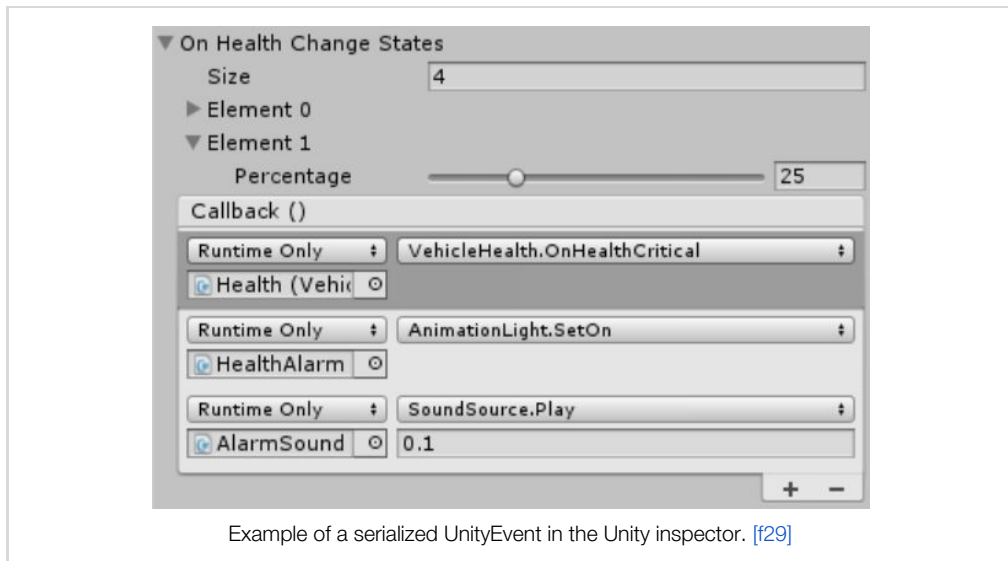
Besides this simple animation which moves a Transform to a fixed position, I also implemented a lot of other helpful components such as animating a Transform’s rotation, or fading the colors of a material, etc. The animation system offers an *“AnimationBase”* class with all of the important main functions. New components with additional functionality can easily be implemented by deriving from that base class. The complete source code with all components of my animation system is available on this Git repository:

<https://github.com/woistjadefox/AnimationSystem>.

Events

Level designers should have the possibility to configure actions based on events. Drawing from my research interviews I determined that most developers use classic C-Sharp events to decouple actions from events. Unity’s own event system that was introduced with Unity 5 was also widely used by the participants of the interview. [44] If level designers are allowed to write their own scripts, they need a way to somehow hook actions into events. The benefit of Unity’s event system is that it offers a class called *“UnityEvent”* which can be serialized and exposed in the inspector. Level designers can then define actions

in the inspector and attach any GameObject to them. By attaching a GameObject, publicly exposed methods of any components corresponding to this GameObject will be selectable in the inspector. There is a limitation of parameter types that are supported by the UnityEvent class.



This fantastic feature was the reason why I decided to work with UnityEvents. In combination with our own implemented animation system, our level designers could configure complex behaviours without writing a single line of code. This can definitely help to avoid bad scripting code from inexperienced programmers which affects stability and performance. The negative side of Unity events are the overhead and the garbage it creates during its creation and invoking process. If you plan to use them often in your game, it is worth reading Jackson Dunstan's article about the performance and overhead comparison between C-Sharp and Unity events. [\[50\]](#)

Another way to give level designers visual scripting possibilities is to use the third-party tool Playmaker. [\[51\]](#) During my bachelor studies when I first heard of Playmaker, I was skeptical and didn't want to give it a try. It was a huge library on which you became dependent once you started to work with it. Since this time Playmaker has become a highly popular visual scripting tool, and even the developers of the popular indie title INSIDE [\[14\]](#) used it. Last year one of our professors gave us a little introduction to it, mainly for the sake of students who didn't know how to code. I was fascinated about the abilities of the tool and I would definitely consider trying it for my next project. So if you don't want to or you don't have the time to create your own visual scripting tools, have a look at it.

4.3 Ten organization rules

As an additional measure for my FAR: Lone Sails team, I defined 10 workflow rules to help keep our project structure organized. These rules are not cut in stone and it depends in each particular case whether it makes sense to apply them or not. For the team it was still helpful to have some written guidelines, and hopefully they may also be useful for other developers. [\[t5\]](#)

1. Don't call any asset or GameObject "test". If you need to place a temporary file, make it clear within the naming; i.e., "tempColliderDoor".
2. Don't put any asset in Unity's root folder.
3. Set all empty GameObjects with logical functions (managers) to position 0 / 0 / 0.
4. Don't scale and rotate empty GameObjects if not necessary. Set the rotation on all axes to 0 and the scale to 1.
5. If you have at least three GameObjects of the same type on a hierarchical folder level, parent them to an additional empty GameObject which describes the type.
6. Try to separate logic and visuals. I.e, keep Colliders and meshes in different GameObjects.
7. Try to separate static and dynamic GameObjects within hierarchies. Mark all your static GameObjects with the "static" flag in the inspector.
8. If you made changes in a scene, be 100% sure nobody else has uncommitted changes before you commit your work to the version control repository. If somebody still does it, they have to pay for the other person's coffee.
9. Everybody stays on the current engine version until the tech lead tells them to update to a different version.
10. If you are working within a prefab, keep all logic and reference within this prefab. It should run out of the box when one instantiates a new one from the project window.

5 Additional knowledge

5.1 Editor extensions

Unity has a specific editor API which offers to extend the editor with additional functions, behaviours, and windows. [52] It is also possible to write custom inspectors and property drawers for components. This can be extremely helpful if you want to offer level designers functional and understandable tools. However, a certain amount of effort is necessary in order to understand the editor's GUI layout system. And of course, the danger exists that you might invest a lot of time building a perfect system or tool, instead of building a working game. I often found myself in situations where I wanted to code a generic editor feature that fits all possible scenarios, and I know other developers who fell into this trap too. Try to find a solution for your specific needs. You can still develop an amazing generic editor tool for Unity's asset store after you have released your game.

For FAR: Lone Sails I wrote several custom inspectors and property drawers. These were more or less just visually improving array arrangements or value manipulations, and are not worth me having a closer look at them here. A feature that did help the team a lot were keyboard shortcuts and scene view helper functions. I will explain these two features better in the next two sub chapters.

5.1.1 Keyboard shortcuts

In order to work efficiently in the scene view, I implemented a keyboard shortcut system that triggered a function based on a pressed key in the editor. Some of these functions are only working in non-playmode and others work in both modes. This table is a little overview of the provided functions [t6]:

Function	Key	In PlayMode	In SceneView
Refill Energy	F1	x	
Switch Camera Distance	F6		x
Teleport Vehicle/Character/Camera	F3		x
Update Weather at Position	F4	x	x
Show / Hide Level-Zone-Helper	F5	x	x
Focus Scene Camera Switch	F8	x	x
Show / Hide Front Covers	F9	x	x
Switch Visual Mode	F10	x	x
Select Vehicle Back Modules	F11	x	x
Select Vehicle Front Modules	F12	x	x
Reset Vehicle & Player	I	x	

Most of the functions help to improve visibility within the scene view. I have already described some of them in the chapter [4.2.2 Managers](#) about the debugging manager .

If you want to implement a keyboard shortcut system for play and non-playmode, read through the following steps:

Create a new file called *“EditorKeyboardShortcuts.cs”* and place it in a folder named *“Editor”*. It doesn't matter where this folder is placed within Unity's asset folder. Paste this code snippet into the script file:

```
using UnityEngine;
using UnityEditor;
using System.Reflection;

[InitializeOnLoad]
public class EditorKeyboardShortcuts {

    static EditorKeyboardShortcuts() {
        // apply global eventHandler
        EditorApplication.CallbackFunction function = () =>
        OnGlobalEventHandler(Event.current);
        FieldInfo info = typeof(EditorApplication).GetField("globalEventHandler",
        BindingFlags.Static | BindingFlags.Instance | BindingFlags.NonPublic);
        EditorApplication.CallbackFunction functions =
        (EditorApplication.CallbackFunction)info.GetValue(null);
        functions += function;
        info.SetValue(null, (object)functions);
    }

    public static void OnGlobalEventHandler(Event e) {

        if (Event.current.keyCode == KeyCode.F1 && Event.current.type ==
        EventType.KeyDown) {
            // do here whatever you want to do
            Debug.Log("F1 was pressed!");
        }
    }
}
```

EditorKeyboardShortcuts.cs code snippet. [\[30\]](#)

As you probably have noticed, the script first gets the *“globalEventHandler”* field of the *“EditorApplication”* class via reflection. Afterwards we do some reflection magic to register our *“CallbackFunction”* delegate we defined before. This hack is necessary to be able to hook into the internal editor callback handlers. There are other ways to capture pressed keys in the editor, but this method worked quite well for me.

After our little hack, our static method *“OnGlobalEventHandler”* will be called when a global event happens. The rest is pretty self explaining. In this example I check if the F1 key was pressed down and if so, the message *“F1 was pressed!”* will be sent to the console.

5.1.2 Scene view helpers

If you want to highlight something in the scene view, you can make use of Unity's debugging draw functions. The class *Unity.Debug* offers a method to draw a line *DrawLine* and one to draw a ray *DrawRay*. They can be helpful if you need to visualize and to debug raycasts from your scripts. The class *Unity.Gizmos* offers a variety of drawing methods with which you can also draw primitive meshes and shapes. The gizmo methods can only be called in a defined function called *OnDrawGizmosSelected* within a *MonoBehaviour* script.

One of the most useful helper features was a self implemented visual mode changer. By separating primitive Colliders and MeshRenderers on different GameObjects and by tagging them with Unity's tag system, it was quite easy to code a toggle function to display or hide one of them. This helped us to set primitive Colliders among meshes since we avoided to use MeshColliders at all. To achieve this kind of a toggle, you can use this method:

```
public void ShowAllMeshes(bool state) {  
  
    Renderer[] allRenderer = Resources.FindObjectsOfTypeAll<Renderer>();  
  
    for (int i = 0; i < allRenderer.Length; i++) {  
        if (allRenderer[i].CompareTag("TagName")) {  
            allRenderer[i].enabled = state;  
        }  
    }  
}
```

Example method to toggle visuals by tag. [\[f31\]](#)

The script will go through all loaded components of the type *Renderer* with the specific tag *TagName* and set the boolean that was defined in the parameter *state*. This method is best coupled with a keyboard shortcut to display only meshes and another shortcut to enable only Colliders in the scene view.

5.2 Version control

“Using a version control system makes it easier for a user/multiple users to manage their code. It is a repository of files with monitored access, which in the case of Unity, will be all the files associated with a Unity project. With version control it is possible to follow every change to the source along with information on who made the change, why they made it and what they changed/added. This makes it easy to revert back to an earlier version of the code or to compare differences in versions. It also becomes easier to locate when a bug first occurred along with what code might have caused it.” [53]

This is a perfect summary of the benefits of using a version control system for your project. I recommend every team who is not already working with one to organize its integration immediately. I can not imagine all the code losses we would have gone through without using version control. As my research interview showed, 90% of the questioned developers work with a version control. I was actually surprised that it was not 100%. Version control immensely helps to improve your professional workflow. The reason some developers still avoid them are the entry barriers which do exist. It is not a simple task to understand existing version control systems, but it is definitely one worth digging into before you start a bigger project.

In FAR: Lone Sails we worked with Git. Git is one of the more popular version control systems among its competitors. It became very famous through the online platform github.com. Git also offers up-to-date features like branching [54] and it has become an industry standard over the past few years. Git can be used with a self-managed server or with cloud stored solutions offered by different providers. Since we already use the tool “SourceTree” to handle our git workflows visually, we also decided to store our repository on its self-owned provider called “Bitbucket”. When working with version control, it is important to change some default settings in the Unity editor and to make other preparations. If you need help to implement such an environment, there is a well written Unity manual about how to set up your Unity project with SourceTree with Bitbucket on the internet. [55]

There are other paid version control tools available which can handle Unity’s workflow needs easier than Git. Plastic SCM is the most popular one next to Unity’s own cloud based collaborative version control, which is currently in beta. In my opinion there are two disadvantages of paid version control services. Next to the periodical costs, your team will also depend on an online service. These

tools also have a smaller online community, and developers often depend on the surrounding support team for a tool just as much as the tool itself.

5.2.1 Scene and prefab handling

Version control and Unity work well together, with one exception. When people work on the same scene or prefab, it is not possible to merge the changes in a semantically correct way. Unity offers a tool called “*UnityYAMLMerge*” to solve this problem. [56] It can be integrated into existing version control systems with a little extra effort. I personally don’t know anybody who has ever used it, nor have I ever read any positive feedback about the tool.

According to discussions on the Unity forum, most developers just try to avoid merge conflicts between scenes and prefabs. Our team also follows this paradigm and we managed to integrate it well into our workflows. Sometimes it can really become a bottleneck for production if a team can not work on the same scene. One possible solution is to split up the content of one scene and to create a separate scene for each discipline of your team. So instead of having one scene with audio, art, and logic, you could create three different scenes loaded at the same time. The different disciplines can then work on their specific field and only commit the changes of their own scene.

5.2.2 Git LFS

Version control might be a powerful tool for Unity projects, but the Git program itself was initially not intended to work with some of the file types Unity is incorporating. Since Git was originally developed to keep versions of classical software projects, it was only built to work with text files. In Unity we have all different types of files. The differential algorithm of Git ignores differential changes of binary files like textures, videos, or audio files. This means that every time you change one pixel of an existing texture, a new version of the complete texture needs to be committed to your repository. If your artists work with tight intervals, this can lead to huge repository sizes. Additionally, it will slow down the speed of your version control system. For these kinds of issues, developers can use an open source Git extension called “*Large File Storage*” (LFS). LFS stores large files on a remote storage instead of the concrete repository. If you work with huge binary assets, I recommend to read the documentation of Git LFS and to implement it within your repository. [57]

5.3 Performance

In order to have a smooth experience for your players, it is mandatory to obey some principles to keep your game's performance within the system resources of your target devices.

To measure the performance of games, it is a common approach to calculate the time a device takes to render 1 frame of the game. Based on this information we can also determine the frames per second. The higher this value is, the more fluid the picture, movement, and actions in game will be. When playing games, normally two processing units are working together to calculate a frame. Unity sends calculation tasks to the central processor unit (CPU) [58] and to the GPU [4]. Within the Unity profiler we can check how much time these two processing units take and this gives us indicators of where we can optimize something in order to keep the calculation time low. [59] If the CPU takes more time to calculate a frame than the GPU we are CPU bound. The other way around, GPU bound.

Since performance optimizations are a general topic in game development the field would go beyond the scope of my thesis. If you would like to know more specifically about performance optimizations within Unity, I recommend to read Unity's tutorial "*Performance Optimization*". [60] I recently also watched a Unite 2016 talk from the creators of the game INSIDE [4] which gave me very detailed extra information I had never heard before. [61]

5.4 Unit tests

Unit tests are a common programming task in established software development processes. Most software developers should do unit tests, but don't like to implement them, since their only function is to test existing behaviour. A unit test is often a simple task runner that tests methods of existing core elements of the software to prove if everything is still working the way it should. They are helpful to check the functionality of a software after new features have been implemented. A smart unit test workflow will reduce bugs and misbehaviour in the long term.

The problem is that Unity is not a classic software project, and these tests somehow need to interact with the editor. Since Unity 5.6 a new implementation of Unity's "*Test Runner*" got shipped which is coupled with the NUnit library. NUnit is an open source unit test library for C-Sharp. The documentation is not yet present in the official manual of Unity but I found an official draft from Unity in the form of an online Google document. [\[62\]](#)

Because of its novelty, I couldn't try the new Test Runner. Nor have I worked before with unit tests within Unity. But this would definitely be something I would like to implement in future workflows and I recommend to check out the new features if you are standing at the beginning of a bigger project.

6 Conclusion

This chapter provides a retrospective look at specific elements of my thesis. I first summarize my conclusions, evaluating whether my beginning assumptions were right. I then write a personal reflexion about my work, followed by an outlook towards future use cases.

6.1 Summary

This thesis indicates the essential tasks required to implement guiding principles and strategies for organizing a Unity project within a game developing team. It highlights various technical issues which are related to structure building processes and their possible solutions.

Interviews with individual developers and studios show how professionals from the industry structure their projects and workflows. From these interview results and subsequent literature review the thesis discussed which implementations were useful and which were not. It emphasized the general need to reflect and to question known structures in order to establish a more productive workflow within a team.

It was through a combination of my academic research and my personal experience from the one and a half year production time of FAR: Lone Sails that "*The Unity Glue*" was shaped. It developed into a cookbook of sorts for indie developers who want to attempt to start a bigger project in Unity. The thesis can be especially useful for game design alumni and other developers who have only worked on smaller projects and prototypes thus far.

6.2 Evaluation

In the introduction I settled the following research question:

How can indie game developer teams achieve a scalable and maintainable project structure within Unity 5.6?

Since every project and every game is different, it is of course not easy to give a generic answer to this question. A first step, independent of any structure, is to analyze and to define how the target game should look. This demands time before the actual production of the game starts for conceptualizing and determining crucial elements of the game. Most teams write a solid game design document (GDD) [63] for this purpose. For FAR: Lone Sails we used a complete wiki website which defined all important elements of the game. The wiki was also a base with which we could share the same vision with our publisher. As soon as the scope, the gameplay, and the potential game assets are clarified, a team can start to research possible suitable project structures.

This thesis shows several ways to solve structural questions and issues. It covers questions about naming conventions, folder hierarchies, and content separation. In addition, it offers further information about logic handling and explanations about how to implement managers. Specific level design workflows for teams are presented which describe how different disciplines can interact with each other in a productive way. This paper also includes tutorials on how to build one's own visual scripting tools for level designers. It shows ways for teams to debug their game with visual markers and keyboard shortcut systems.

In addition to the main research question, I made several assumptions:

Assumption 1: Determining basic project structure rules will improve the overall quality and efficiency of the production workflow.

This turned out to be true for our studio. All structures helped us to talk about the same “thing”. Members could more easily find what they were looking for; for example assets or positions in a specific scene or all over the game. After using time tracking tools within our project management software, we realized that our efficiency got better when we clarified project structure. We all feel like we are working in a more professional way and that everybody has gained greater confidence in their abilities.

Assumption 2: Early logic planning for seamless content loading will help to sort and to form the final game content bundles.

Our early decision to split the complete game into small level chunks in FAR: Lone Sails helped us to categorize concrete sections of the game and all their corresponding assets. We could define zones and, based on objects that were shown, we could also decide how to put together textures on atlases due to memory performances. Even if you are not planning to load content seamlessly, the outlined requirements will help you to categorize objects and abstract logic.

Assumption 3: Additional self-implemented debugging tools will help to avoid heavily time consuming bug determining processes.

In our team’s case this assumption turned out to be valid. We had a team member initially selecting three main GameObjects in order to debug a specific position, a process which took about 5-10 seconds. With a keyboard shortcut that moved these three objects to the current mouse position, we could do this task in 1 second. We had to move these main objects around quite often every day, and those minute increments of time added up. This is only one example of how important self-written, game related debugging tools were for us.

6.3 Personal reflection

In reflecting on my thesis, I can name a few things that worked well, and some that did not end up the way I had planned them. I always knew it would be hard to write a document about principles and best practice workflows, and I had suspected I set the scope to be way too wide right from the beginning. Finding out what I should take into the thesis and what I could leave out was difficult. Game development is a pretty big and complicated field and it covers many different tasks and disciplines. The same applies for a project structure within a game engine. In order to not become totally lost, I had to set my focus on our game FAR: Lone Sails, and I based the thesis's research and findings according to FAR's needs.

What frightened me the most was the possibility that I could unintentionally write about bad practices or things that a lot of developers would disagree with. I had initially thought that by writing this kind of document, one would automatically claim (explicitly or implicitly) to have the “one” objective solution. This was definitely never my goal and I still recognize that I am not an expert in the field of project structuring in Unity. I would however say that I have improved my abilities, and at a certain point in the writing process I figured out that I don't have to try and write the one absolute truth. I know that I am still a student and that it will still help others when I write about the experiences I had with my in-progress strategies.

What did not always work well was my research workflow. Based on the fact that I still had to work on a game with deadlines, I often tried to implement a component of the paper by adding existing experiences and with inefficient “*try and fail*” methods. I lost time figuring out things by myself instead of setting up a concrete research approach first. I especially wasted way too much important time with the logic managers. For the save game manager for example, I tried a new overly complex technology which we didn't even need for the game. This was obviously an act of escaping from my real work tasks into a technological novelty that interested me at that time. (Similarly, I also can't remember another time in my life when I had ever cleaned my apartment so often.)

I feel positive that I accomplished my goal of writing a useful document for game design alumni and other game developers. I believe they can use this thesis to conceptualize the scope of a complete indie game project, and to get some important development hints before they consider starting their own bigger game production. If I think back to a time 1.5 years ago, this thesis could have been helpful to me. This gives me confidence that this thesis contains relevance and value.

6.4 Outlook

This thesis writing process has motivated me to continue to question and to improve Unity project structures and workflows. My motivation for writing it was not about finding a specific, generic best practice approach which could be applied to all teams and all projects. It was more about finding concrete design conclusions and reflecting on Unity development in general. I explored why project structuring should be a task for every indie developer team, and how it is one that is ideally not orchestrated by a single team member. I hope this document will help other developers to reflect on their project structures and to prompt them to think about how they can improve their workflows.

As a first step towards sharing my ideas I have applied to be a speaker at the Unite '17 Europe in Amsterdam. My submission holds the title "*Multi Scene Editing in Unity for FAR: Lone Sails*", and I would like to talk about the specific workflow process we used to implement our seamless content loading system. The decision will be announced within the next two weeks at the time of writing. I am also interested in establishing contact with an official Unity team member who is involved in project structuring processes. I would appreciate the opportunity to exchange findings and to have a talk about possible improvements or advancements for the game engine.

I hope this thesis can be used as a base for further research, publications, or talks in game related (and other) fields. I invite everybody to reference this thesis for their own work and purposes, if they can benefit from its findings.

Besides this thesis outlook, I will be continuing to work on FAR: Lone Sails with my team in order to have the game ready to ship by November 2017.

7 Credits

I first want to thank my mother Andja Macot-Saric and my sister Grozdana Matic. They are my emotional backbone and they have always believed in every step I made on this planet. The second thank you goes to Don Schmocker. He has always been a unique and wonderful friend over all of the years we studied together. Thanks for the trust you have in me, and for all the good times in the past and in the future.

I additionally want to thank my teachers and mentors, René Bauer, Mela Kocher, Ulrich Götz, Beat Suter, Max Moswitzer, and Margarete Jahrmann for their help, advice and feedback during my master's degree. I especially want to thank Rebecca Goodine, who reviewed my thesis and corrected my German influenced English sentences and typos. This thesis would not be an enjoyable read without her, thank you so much!

An additional thank you goes out to all of my colleagues from our publisher Mixtvision. Their support and feedback was crucial for the production of FAR: Lone Sails. I also want to thank all my master's student colleagues, Christian Schmidhalter, Francine Rotzetter, Daniel Gonçalves, and Patrik Toth for the good times we shared together in the master's atelier and on our excursions.

At last I want to thank all of my friends who supported and suffered with me during the last one and a half year. Thanks for being with me.

8 Bibliography

[1]	Mr. Whale's Game Service. (n.d.). Retrieved January 18, 2017, from http://www.misterwhale.ch
[2]	Schmocker, D. (n.d.). Portfolio. Retrieved January 4, 2017, from http://www.donschmocker.ch
[3]	Mixtvision. (n.d.). Retrieved December 20, 2016, from http://www.mixtvision.de
[4]	Definition GPU. (n.d.). Retrieved December 11, 2016, from http://www.pcmag.com/encyclopedia/term/43886/gpu
[5]	Here's Microsoft's annual Xbox One indie montage - Xbox E3 2016. (2016, June 13). Retrieved April 20, 2017, from https://www.youtube.com/watch?v=9EHneQXksX8
[6]	SEO: The Beginner's Guide to Search Engine Optimization from Moz. (2014, March 04). Retrieved February 11, 2017, from https://moz.com/beginners-guide-to-seo
[7]	FAR: Lone Sails. (n.d.). Retrieved May 1, 2017, from http://store.steampowered.com/app/609320/FAR_Lone_Sails/
[8]	Dutton , F. (2012, April 18). What is Indie? Retrieved May 15, 2017, from http://www.eurogamer.net/articles/2012-04-16-what-is-indie
[9]	Queiroz, C., & Smith, M. (2015). Unity 5.x Cookbook: Over 100 recipes exploring the new and exciting features of Unity 5 to spice up your Unity skillset. Packt Publishing.
[10]	Thorn, A. (2017). Mastering Unity 5.x. Packt Publishing.
[11]	Lukosek, G. (2016). Unity 5: Learning C# by Developing Games. Packt Publishing
[12]	Unity Technologies. (n.d.). Unity Community. Retrieved November 17, 2016, from https://forum.unity3d.com
[13]	Korzuszek, P. (2016, August 23). 7 Ways to Keep Your Unity Project Organized. Retrieved January 13, 2017, from http://blog.theknightsofunity.com/7-ways-keep-unity-project-organized/

[14]	Tulleken, H. (2012, July 12). 50 Tips for Working with Unity (Best Practices). Retrieved June 11, 2016, from http://devmag.org.za/2012/07/12/50-tips-for-working-with-unity-best-practices/
[15]	Best practices - Folder structure. (2008). Retrieved November 10, 2016, from https://forum.unity3d.com/threads/best-practices-folder-structure.65381
[16]	Zurich Unity 3D User Group. (n.d.). Retrieved November 20, 2016, from https://www.meetup.com/Zurich-Unity-3D-User-Group/events/235712116/
[17]	Microsoft. (n.d.). Model-View-Controller. Retrieved May 20, 2017, from https://msdn.microsoft.com/en-us/library/ff649643.aspx
[18]	Microsoft. (n.d.). Dependency Injection. Retrieved May 16, 2016, from https://msdn.microsoft.com/en-us/library/dn178469(v=pandp.30).aspx
[19]	N. (2015, February 05). Introduction to Object Oriented Programming Concepts (OOP) and More. Retrieved August 11, 2016, from https://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep
[20]	Sand Sailor Studio. (n.d.). Retrieved October 3, 2016, from http://www.blackthefall.com/sand-sailor-studio.html
[21]	Blindflug Studios. (n.d.). Retrieved September 25, 2016, from http://www.blindflugstudios.com/
[22]	Camel Case. (n.d.). Retrieved June 9, 2016, from http://wiki.c2.com/?CamelCase
[23]	Unity Technologies. (n.d.). Roadmap Retrieved February 10, 2017, from https://unity3d.com/unity/roadmap
[24]	Unity Technologies. (n.d.). Unity 5.3 Release Notes. Retrieved February 11, 2017, from https://unity3d.com/unity/whats-new/unity-5.3
[25]	Making the World of Firewatch. (2016, May 20). Retrieved March 1, 2017, from https://www.youtube.com/watch?v=hTqmK1Zs_1I
[26]	SECTR on the Asset Store. (n.d.). Retrieved February 20, 2017, from https://www.assetstore.unity3d.com/en/#!/content/15356

[27]	Unity Technologies. (n.d.). Editor settings. Retrieved January 14, 2017, from https://docs.unity3d.com/Manual/class-EditorManager.html
[28]	Unity Technologies. (n.d.). Description of the Format. Retrieved March 19, 2017, from https://docs.unity3d.com/Manual/FormatDescription.html
[29]	Microsoft. (n.d.) Guid Structure. Retrieved February 1, 2017, from https://msdn.microsoft.com/en-us/library/system.guid(v=vs.110).aspx
[30]	Unity Technologies. (n.d.). Assets, Objects and serialization. Retrieved April 20, 2017, from https://unity3d.com/learn/tutorials/temas/best-practices/assets-objects-and-serialization
[31]	Unity Technologies. (n.d.). SceneManager. Retrieved April 2, 2017, from https://docs.unity3d.com/ScriptReference/SceneManagement.SceneManager.html
[32]	Microsoft. (n.d.). General Naming Convention. Retrieved February 20, 2017, from https://msdn.microsoft.com/en-us/library/ms229045(v=vs.110).aspx
[33]	Free flowchart maker and diagrams online. (n.d.). Retrieved April 4, 2017, from https://www.draw.io/
[34]	Unity Technologies. (n.d.). Introduction to Scriptable Objects. Retrieved May 20, 2017, from https://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/scriptable-objects
[35]	Wikipedia. (2017, May 11). Singleton pattern. Retrieved May 20, 2017, from https://en.wikipedia.org/wiki/Singleton_pattern
[36]	4 major principles of Object-Oriented Programming. (n.d.). Retrieved February 4, 2017, from http://codebetter.com/raymondlewallen/2005/07/19/4-major-principles-of-object-oriented-programming/
[37]	Beal, V. (n.d.). API - application program interface. Retrieved February 4, 2017, from http://www.webopedia.com/TERM/A/API.html
[38]	Koirala, S. (2013, September 17). Can you explain Lazy Loading? Retrieved April 20, 2017, from https://www.codeproject.com/Articles/652556/Can-you-explain-Lazy-Loading

[39]	Modesttree/Zenject. (2017, May 14). Retrieved May 20, 2017, from https://github.com/modesttree/Zenject
[40]	Wikipedia. (2017, April 15). Ahead-of-time compilation. Retrieved May 20, 2017, from https://en.wikipedia.org/wiki/Ahead-of-time_compilation
[41]	Unity Technologies. (n.d.). IL2CPP. Retrieved May 2, 2017, from https://docs.unity3d.com/Manual/IL2CPP.html
[42]	Introducing JSON. (n.d.). Retrieved February 10, 2017, from http://www.json.org/
[43]	Unity Technologies. (n.d.). Object.GetInstanceID. Retrieved May 10, 2016, from https://docs.unity3d.com/ScriptReference/Object.GetInstanceID.html
[44]	Unity Technologies. (n.d.). UnityEvents. Retrieved May 20, 2017, from https://docs.unity3d.com/Manual/UnityEvents.html
[45]	Unity Technologies. (n.d.). Serialization in Unity. Retrieved May 12, 2017, from https://blogs.unity3d.com/2014/06/24/serialization-in-unity/
[46]	Unity Technologies. (n.d.). Input. Retrieved May 14, 2017, from https://docs.unity3d.com/ScriptReference/Input.html
[47]	InControl. (n.d.). Retrieved May 12, 2017, from http://www.gallantgames.com/pages/incontrol-introduction
[48]	Design Patterns and Refactoring. (n.d.). Retrieved May 17, 2017, from https://sourcemaking.com/design_patterns/command
[49]	Nystrom, B. (n.d.). Command - Design Pattern Revisted. Retrieved January 20, 2017, from http://gameprogrammingpatterns.com/command.html
[50]	Dunstan, J. (2016). Event Performance: C# vs UnityEvent. Retrieved April 20, 2017, from http://jacksondunstan.com/articles/3335
[51]	Hutong Games. (n.d.). Retrieved May 15, 2017, from http://www.hutonggames.com/
[52]	Unity Technologies. (n.d.). Extending the Editor. Retrieved May 12, 2017, from https://docs.unity3d.com/Manual/ExtendingTheEditor.html

[53]	Unity Technologies. (n.d.). Version control integration. Retrieved May 12, 2017, from https://docs.unity3d.com/Manual/Versioncontrolintegration.html
[54]	3.1 Git Branching - Branches in a Nutshell. (n.d.). Retrieved May 12, 2017, from https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell
[55]	Unity Technologies. (n.d.). Creating Your First Source Control Repository. Retrieved May 12, 2017, from https://unity3d.com/learn/tutorials/topics/cloud-build/creating-your-first-source-control-repository
[56]	Unity Technologies. (n.d.). Smart Merge. Retrieved May 11, 2017, from https://docs.unity3d.com/Manual/SmartMerge.html
[57]	Git Large File Storage. (n.d.). Retrieved February 1, 2017, from https://git-lfs.github.com/
[58]	What is CPU (Central Processing Unit)? (2017, April 26). Retrieved May 12, 2017, from https://www.computerhope.com/jargon/c/cpu.htm
[59]	Unity Technologies. (n.d.). Introduction to the Profiler. Retrieved May 12, 2017, from https://unity3d.com/learn/tutorials/topics/interface-essentials/introduction-profiler
[60]	Unity Technologies. (n.d.). Performance Optimization. Retrieved May 12, 2017, from https://unity3d.com/learn/tutorials/topics/performance-optimization
[61]	Unite 2016 - Tools, Tricks and Technologies for Reaching Stutter Free 60 FPS in INSIDE. (2016, December 01). Retrieved May 7, 2017, from https://www.youtube.com/watch?v=mQ2KTRn4BmI
[62]	[5.6 DRAFT] Unity Test Runner docs. (n.d.). Retrieved May 20, 2017, from https://docs.google.com/document/d/1SeNOAVYaq9HUjsKAC2ZvRwKLD2MCNyV4LwcsP3BXm0s/edit#heading=h.vp7dx6id5d21
[63]	Freemant, T. (1997, September 12). Creating A Great Design Document. Retrieved May 20, 2017, from http://www.gamasutra.com/view/feature/131632/creating_a_great_design_document.php

9 Ludography

[1]	Okomotive GmbH. (n.d.). FAR: Lone Sails - An Atmospheric Vehicle Adventure Game. Retrieved May 20, 2017, from http://www.far-game.com Steam & consoles, release date Q4 2017.
[2]	Ink Kit GmbH. (n.d.). DERU an atmospheric, cooperative Puzzle Game. Retrieved May 20, 2017, from http://www.deru.ch/ Steam & consoles, release date Q4 2017
[3]	Campo Santo. (n.d.). Firewatch. Retrieved May 20, 2017, from http://www.firewatchgame.com/ Steam & consoles
[4]	Playdead. (n.d.). INSIDE. Retrieved May 20, 2017, from http://www.playdead.com/games/inside/ Steam & consoles
[5]	Sand Sailor Studio. (n.d.). Black the fall - The Game. Retrieved May 20, 2017, from http://www.blackthefall.com/ Steam & consoles
[6]	Blindflug AG. (n.d.). Airheart. Retrieved May 20, 2017, from http://airheartgame.com/ Steam

10 List of tables

[t1]	FAR: Lone Sail - Technical challenges
[t2]	Table of definitions
[t3]	Unity vocabulary
[t4]	Description of the term "asset"
[t5]	Ten organization rules
[t6]	Editor keyboard shortcuts for FAR: Lone Sails

11 List of figures

[f1]	Mr. Whale's Game Service. From left: Martina Hugentobler, Don Schmocker, Melanie Vetterli, Goran Saric, Christian Schmidhalter, Dominik Haas. Self-owned image.
[f2]	Screenshot of FAR: Lone Sails. Self-owned image.
[f3]	Snippets from the FAR trailer during the Xbox E3 press conference. Self-owned image.
[f4]	FAR: Lone Sails team at IGF 17. From left: Goran Saric, Martina Hugentobler, Don Schmocker. GDC 2017 Award Step and Repeat. (2017, March 20). Retrieved May 1, 2017, from https://www.flickr.com/photos/officialgdc/albums/72157679440550760
[f5]	Cover " <i>Unity 5.x Cookbook</i> ". Nally, P. (n.d.). Technically, It's Technical. Retrieved May 15, 2017, from http://technicallyitstechnical.blogspot.ch/2015/12/unity-5x-cookbook-i-just-love-unity.html
[f6]	Cover " <i>Mastering Unity 5.x</i> ". Mastering Unity 5.x. (2017, January 01). Retrieved May 12, 2017, from https://www.packtpub.com/game-development/mastering-unity-5x
[f7]	Question: For how long are you working with Unity? Google forms. Self-owned image.
[f8]	Question: What's the Unity version you have worked on your current or last project? Google forms. Self-owned image.
[f9]	Question: Are you working with version control? Google forms. Self-owned image.

[f10]	Question: How do you cross-reference objects between scenes? Google forms. Self-owned image.
[f11]	Question: Do your level designers need to script code? Google forms. Self-owned image.
[f12]	Project view / window. Self-owned image.
[f13]	Hierarchy view / window. Self-owned image.
[f14]	Scene view / window. Self-owned image.
[f15]	Example: multiple scene editing in the hierarchy window. Self-owned image.
[f16]	Example: YAML structured text content of a *.unity file. Self-owned image.
[f17]	Example of the cloud example described above. Self-owned image.
[f18]	SceneCollection.cs code snippet. Self-owned source code.
[f19]	SceneCollectionCreator.cs code snippet. Self-owned source code.
[f20]	SceneCollectionInspector.cs code snippet. Self-owned source code.
[f21]	Inspector view of the SceneCollection asset. Self-owned image.
[f22]	API.cs code snippet. Self-owned source code.
[f23]	The instance id of each component can be shown by activating the debug mode in the inspector. Self-owned image.

[f24]	Example of the described scenario. Self-owned image.
[f25]	Standardized control mapping of InControl. InControl: Standardized Controls. (n.d.). Retrieved May 15, 2017, from http://www.gallantgames.com/pages/incontrol-standardized-controls
[f26]	Debugger: keyboard shortcut that helps to visualize the borders of each scene / section. Self-owned image.
[f27]	Visual changer feature: 1. picture: mesh mode, 2. picture: mix mode, 3. picture: collider mode. Self-owned image.
[f28]	Inspector of the AnimationFixedPosition.cs script. Self-owned image.
[f29]	Example of a serialized UnityEvent in the Unity inspector. Self-owned image.
[f30]	EditorKeyboardShortcuts.cs code snippet. Self-owned source code.
[f31]	Example method to toggle visuals by tag. Self-owned source code.

12 Licence

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by/4.0/>

