Creating a Unity3D editor technical framework to streamline the prototype programming process in GameLab Oost

Alexey Khazov | 414422

Creative Media and Game Technologies

Saxion University of Applied Sciences

Student Name: Alexey Khazov

Student Number: 414422

Submission Date: 18 June 2019

Exam Code: T.35809/T.35910

Graduation Supervisor: Yiwei Jiang

Company Supervisor: Keesjan Nijman

Word Count: 8,500

Table of Contents

Abstract

This research paper and graduation assignment investigate creating a programming framework for the GameLab Oost company to streamline and improve their prototype programming process. The assignment uses the design thinking research method to explore the problem, empathize with the target audience, prototype the framework and test it. The main testing method has the programming interns at GameLab Oost start a totally new Unity project using a GameLab Project template and create a small interactive program using the features the framework provides. They then fill in a survey about their experience working with the framework, which is mostly positive. Most interns rate the framework very intuitive and easy to use, and are not missing or lacking any features. The framework is future proof and easily expandable, and does indeed streamline the prototype programming process in GameLab Oost, allowing them to produce higher quality products in a shorter period.

Introduction

The purpose of this graduation report is to provide an overview on the central problem, the client, the objectives, the methods, and the conclusions of the research. The research will focus on figuring out how to create a Unity3D (Unity) technical framework for the programming interns at GameLab Oost (GameLab) to improve and streamline their game prototyping process. From this point forward, any mention of the prototype or development process at GameLab refers only to the programming part of it. The research and testing will be conducted in-house on GameLab grounds, with various different departments and people of varying skill levels.
The major research and testing target will be a game project Holland Casino hired GameLab to make. However, while Holland Casino is the entity offering the assignment and is, therefore, the client for the game project, this research will focus on creating a technical framework for GameLab, making them not only the company, but also the client for this research and graduation assignment.

Client

The client and the company for this graduation assignment are both GameLab Oost. The client is located on Ariensplein 1, Enschede and specializes in creating serious game prototypes for various clients in different industries. They create serious game prototypes on various different platforms, such as PC, VR, AR, mobile and console based on what is best for the client.
GameLab Oost is a non-profit organization that mainly hires students as interns in an attempt to provide them with a learning environment, and the possibility to start their own start-up based on the game they developed.

Reason for the Assignment

Holland Casino hired GameLab Oost to help them create a game prototype that would bring more millennials, which are people “born in the 1980s, 1990s, or early 2000s”, to the casino for the first time (MILLENNIAL | meaning in the Cambridge English Dictionary, n.d.).
Besides Holland Casino, Jarabee, another company, hired GameLab Oost to produce a game to help train potential adopters in taking care of adopted children. At the same time, the client started developing a game prototype about their own company to better show their clients what they could do for them.
The client uses the Unity3D game engine for most of its projects, but unfortunately, they do not get many programming interns with proper Unity or even programming experience (See the Empathize chapter for more details on the lacking knowledge and experience). While the lack of experience allows the interns to research and gain more knowledge about the engine, it turns

every project into a playground, instead of a solid, quality product. Furthermore, the amount of time the interns spend on learning the basics of Unity takes away from the time they could invest in improving, polishing and adding to the GameLab's projects.

In addition, the client normally works on multiple projects simultaneously with the interns split into separate groups. This arrangement and the client's lack of a project framework result in the groups recreating the same systems, and wasting their resources on unnecessary, time consuming systems, like inventory and item management, game saving, and even basic user interface (UI) and menu managers.

Moreover, the interns' lack of experience in the professional field hinders their ability to efficiently work together and communicate properly with other team members. This results in chaotic, messy and barely functioning prototypes that the client cannot improve upon, and extend into a fully-fledged game for their own clients.

The graduation assignment will, therefore, focus on creating a base technical framework that attempts to solve the above mentioned problems and streamline the programming part of the prototype creation process in GameLab Oost.

Preliminary Problem Statement

The client's question is how to compensate for the interns' lack of experience with Unity so that they could produce more polished, stable and extensible prototypes that satisfy the needs and requirements of their clients.

Game development and programming is much easier nowadays thanks to game engine evolution, and the various powerful tools they include. Today's technologies let anybody pick up a game engine and create stunning scenes, on all the popular computer platforms, with spatial audio, artificial intelligence, and realistic physics simulations. Nevertheless, engines like Unity aim to be as flexible as possible. This forces them to be abstract and prevents them from including more specialized tools, such as a questing system, mainly because the requirements for these systems differ between games and must, therefore, be implemented by the engines' end-users.

To solve the client's problem, this research will examine the most reused tools, systems, and problems in GameLab Oost, extract the most important elements into an overarching framework, and let other programming interns use it as a test.

Theoretical Framework

To determine whether the preliminary problem is the client's actual problem, this preliminary research will investigate existing Unity3D game programming frameworks. It will take into account how these frameworks operate, what problems they solve, whether they are specific enough for GameLab's needs, and which parts, if any, can be extracted from them into a custom framework for GameLab.

Customizing the Unity3D Editor

Due to Unity's abstract nature, the responsibility to extend and specialize the engine falls onto the end-user. Nevertheless, the Unity team thought of this issue and provide different ways to customize and extend their engine.
One example of specializing the engine is a quest editing tool the graduating student created during this graduation assignment for the Holland Casino project -

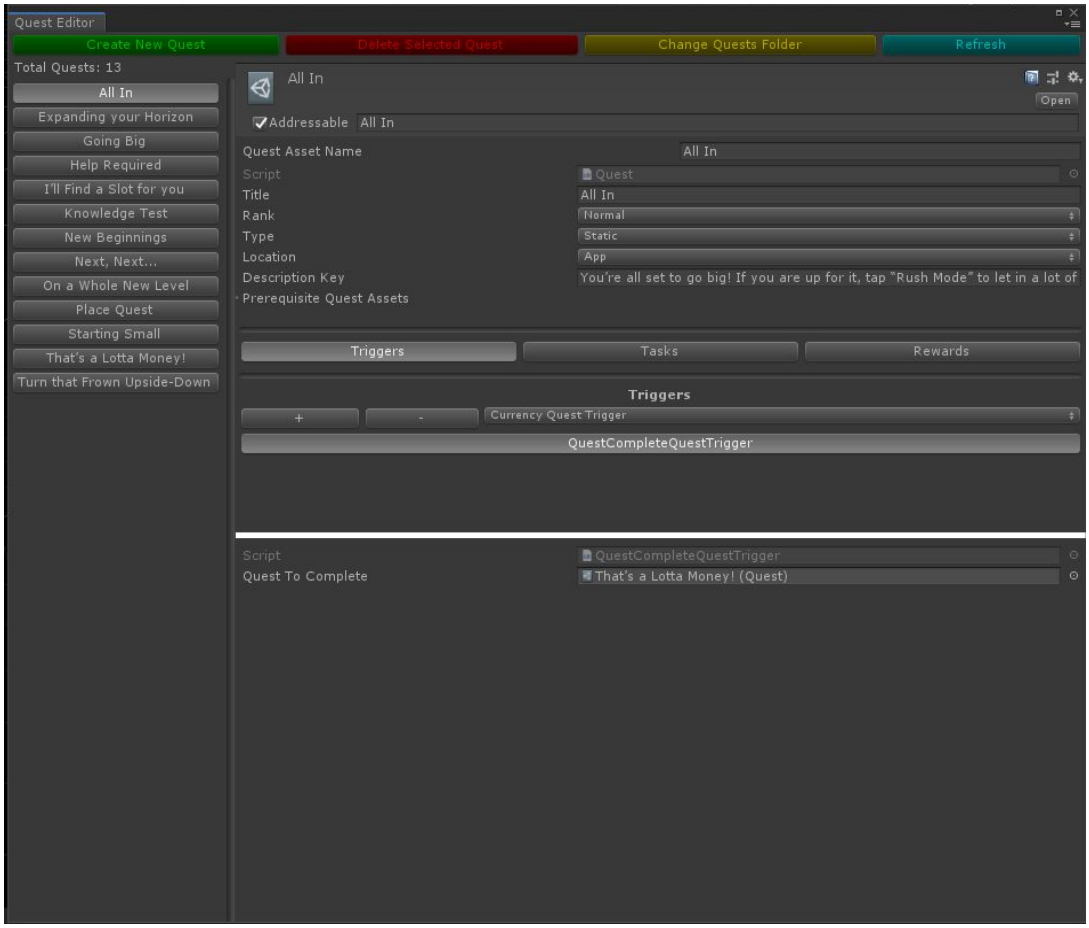

Figure 1. Custom quest editor window made in Unity by Alexey Khazov.

The most common method of creating specialized tooling in Unity is custom editor scripts, like custom inspectors, property drawers and windows. Editor customization allows turning a complicated setup into one click of a button. As an example of that, a popular Unity extension is Playmaker, a visual scripting tool for the engine, that can be seen in the figure below (Hutong Games LLC, n.d.).

Figure 2. Example of what PlayMaker looks like.

Developing such extensions is not simple, and requires a considerable time investment. However, Unity Editor extensions do not have to be very complicated, and can be as basic as useful functions to the context menus of scripts or the editor's main menu.

The most important Unity functions reside in the main menu, split into various categories, like Assets, and Component (See figure below). The main menu provides quick access to useful tools, like refresh the asset database, or focusing on the selected game object in the scene. They also allow quickly adding various components to game objects.

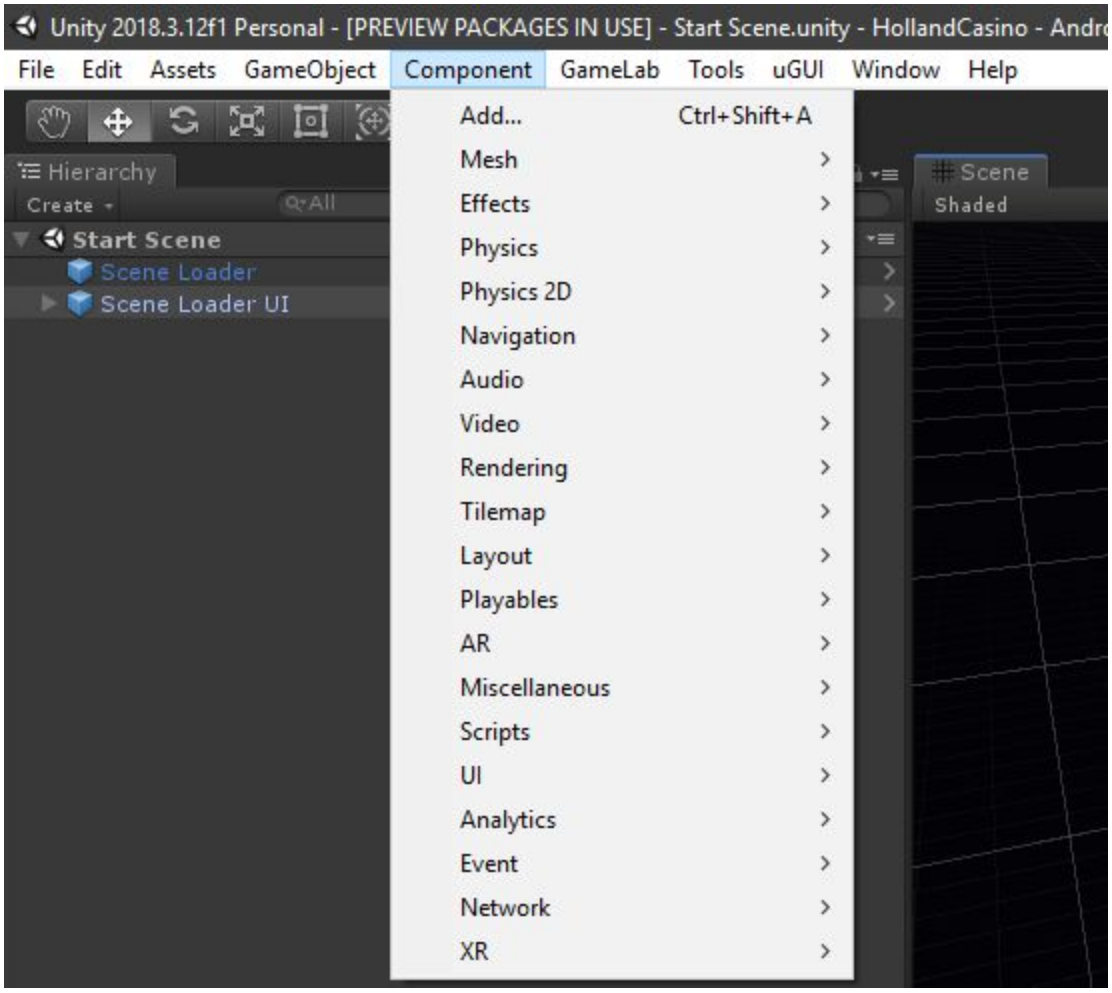

Figure 3. Various options available in the component category in Unity's main menu.

Because of how useful the main menu is, Unity made it incredibly simple to add new items to it. To achieve that, developers need only add the MenuItem attribute to a static method and give it a name (MenuItem, n.d.). Figure 4 demonstrates this in action -

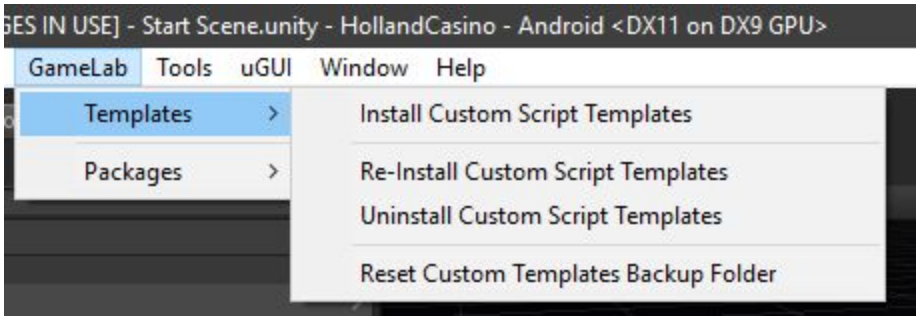

Figure 4. Custom GameLab Framework menu items.

Common Game Systems

A quick look through some of GameLab's previous projects, and a few discussions with company members revealed the following recurring mechanics, systems, and problems:

- Saving & Serialization
- Shop, Inventory & Resource Management
- Building System
- Scene Management
- Recreating existing functionality in a complicated way
- Chaotic projects
- No programming guidelines

Programming Patterns

The cause for most of the issues in GameLab projects is poor object communication. The programmers create different game systems, but have a hard time making them talk to each other. Unity has many ways of acquiring references to objects, however, the lack of proper programming guidelines causes the programmers to use many ways at once, thus creating complicated and fragile setups that are unmaintainable (See Chapter 1: Empathize & Define - Unity API for more information). Therefore, to solve this issue, the core of the GameLab framework will create and implement a set of guidelines that relies on the singleton and observer patterns for better object communication (See Appendices 4 and 5 for more information on the singleton and observer patterns).

Similar Existing Products

Creating a programming framework for Unity has been the goal of many developers since the engine's release. There are many code samples, scripts and entire packages aimed at beginners and intermediate developers to help them solve common Unity problems. The biggest resource for such assets is the Unify Community Wiki. The wiki contains a "large assortment of sample scripts and code snippets contributed by members of the community" (Unify Community Wiki, n.d.). It splits the scripts into various categories and subcategories, with many useful snippets, some of which are:

- StringUtil - Has "a few functions to wrap strings to a certain number of characters"
- MD5 - provides the ability to hash an input string using the MD5 hashing function.
    - This is greatly useful for multiplayer games with password input.

- Windows Saved Game Directory - Locates the dedicated Saved Games directory on windows machines.
- ClassTypeReference - Provides a serializable reference to System.Type with a custom property drawer and attributes to filter the type selection in the Unity inspector.

Unfortunately, the open-source nature of this resource means the code is inconsistent application programming interface (API) and convention-wise, and is not as optimized or is the best solution. Furthermore, every sample requires that the original author be credited, to have a link to the license under which it is distributed, and to always mention whether any changes were made (Creative Commons License Deed, n.d.). Despite the samples having a permissive license, they still require a lot of maintenance and taking care of, to avoid breaching any legal agreements. Moreover, the client, wishes to have their own framework they can do whatever they want with.

Save Systems

Unity does not have any specific game saving system tool. However, it does have PlayerPrefs, a more abstract system that could be used as one.

PlayerPrefs

Unity provides a quick and easy to use tool designed for "[storing] and [accessing] player preferences between game sessions" (PlayerPrefs, n.d.). The tool has some basic features to add, retrieve, and delete floats, ints, and strings from a persistent key-based dictionary. This tool makes it easy to persist and save data, even after the application closes. Based on the platform the game is running on, PlayerPrefs saves and loads its data from different locations. On Windows, for example, it is the registry, while on Linux, it is in a file found at "~/.config/unity3d/[CompanyName]/[ProductName] using the company and product names specified in the Project Settings" (PlayerPrefs, n.d.).

Serialization

While the tool allows developers to store float, int, and string values, it is very hard to represent certain game states with just those three types of variables. Therefore, to use PlayerPrefs as a proper save system, it is possible to use one of the many available serialization solutions, which take object instances and convert their data into a string representation, that can be used to rebuild that object instance, serialize all the objects responsible for the current game state, such as object transforms and item lists, and save the resulting string through PlayerPrefs.
The most popular serialization format is Json, and Unity includes a very basic JsonUtility class to serialize and deserialize objects to and from it. (Alba, L. D, 2016). Unfortunately, the JsonUtility class is too basic, and requires a lot of boiler-plate code to serialize complex data structures. Therefore, due to the widespread use and support of Json, alongside its "human readable code

[and] very simple and straightforward specification," the GameLab Framework would use the most popular Json library for .NET, Newtonsoft's Json.NET, that also has support for Unity for serialization (Alba, L. D, 2016, Json.NET, n.d.).

Disadvantages

Nevertheless, the PlayerPrefs system has a few drawbacks, mainly showing up on the Windows platform. Due to Unity's choice of storing player preferences on Windows machines in the registry, as opposed to files on other platforms, it makes it impossible for end-users to back up their save data and transfer it between different machines (HonoraryBob, 2017). Furthermore, the gaming community is accustomed to finding save files in specific locations, that most games use, on their systems. Using PlayerPrefs on a Windows machine would break these expectations, and result in annoyed clients for GameLab.

Furthermore, PlayerPrefs is not made specifically for save data, and can therefore, contain other keys and values, such as player settings. The PlayerPrefs API provides a very useful, but also dangerous, function, called DeleteAll, which "removes all keys and values from the preferences" (PlayerPrefs, n.d.). With multiple developers working on different parts of the same project, one developer could use PlayerPrefs for their configuration system, and then call the DeleteAll function, thinking that it would only reset their settings preferences. However, this call also deletes the player's save data, and now the developer must instead manually delete every key they have set in order to add the ability to reset their configuration. Therefore, to avoid these issues, the framework would need to implement a custom saving system that stores serialized save data to a predefined location on disk.

Existing Save Solutions

Saving game data is a very common task. Hence, there are many solutions available for Unity that do just that. One very well-rated solution is Easy Save, which is very "easy and well documented" for amateurs, and is "fast, feature-rich and extremely flexible" for experts (Moodkie, n.d.). However, the downside of this asset is that it has a price tag, something that the client cannot afford, due to their non-profit nature.

There are also free tools, such as Quick Save (Clayton Industries, n.d.). This tool uses Newtonsoft's Json.NET library, and is fairly easy to use. However, as a learning experience, to avoid licensing issues, and to have a tool that fits nicely with the rest of the framework and provides an even easier interface and more intuitive workflow, the GameLab framework would still implement its own custom Save System.

Menu Systems

With the introduction of Unity 5.0, the Unity team released an entirely new UI system. While the system made creating UI a much easier task, all the different UI components and prefabs still

needed to come together and interact with the rest of the game. Like anything else in Unity, the engine expects the user to use UnityEvents, direct reference setup in the inspector, or reference detection at runtime to allow functionality such as the player opening up an inventory menu, or updating the HUD.

There are plenty of menu manager solutions available for unity, one being ZUI - Menus Manager (Khalid, n.d.). This asset makes creating and managing menus a very easy task, but, once again, it has an associated cost with it, which GameLab cannot afford. Unfortunately, there do not seem to be any free menu manager systems on the Unity asset store. Therefore, the GameLab Framework would need to create a custom menu manager system that focuses on gaining access to the different menus in the game and providing a simple, easy and intuitive interface to opening and closing them at any part of the game code.

Peer Review

One very important aspect of a project's life cycle is code peer review. Peer code review "is a systematic examination of software source code ... to find bugs and improve overall quality of the software" (Devart, n.d.). Code review is very important and has the following benefits:

- Finding bugs early, when they are cheap to fix
- Helps maintain consistent coding style and standards across the company
- Teaching and sharing knowledge as team members gain a better understanding of the code base and learn from each other
- Helps maintain a level of consistency in software design and implementation
- Review discussions save team members from isolation and bring them closer to each other
- Build the confidence of stakeholders with regard to the technical quality of the execution of the project

Source: Devart, n.d., Cuelogic Technologies, 2019

Peer Review Tools

Generally, peer reviews happen face to face as two or more programmers sit together and discuss the code base. However, there are tools that attempt to up the quality, frequency and ease of code reviewing. One such tool is Review Assistant, that integrates directly into Visual Studio, and works with Git, SVN, and many other version control products (Devart, n.d.). This tool allows inserting review comments directly into the code that anyone can see and reply to at their own free time, with notifications, reports and statistics. It also lets teams assign reviewers to specific parts of the code base, create moderators, and keeps track of unreviewed or buggy code (Devart, n.d.).

Best Practices

Thankfully, code reviewing is a very common practice and has been around for a while. That allowed a set of common best practices to emerge, mainly focusing on quality rather than quantity. Some of these practices include (For a full list of best practices and their explanations, see Appendix 3:

- Review fewer than 400 lines of code at a time
- Take your time. Inspection rates should be under 500 lines of code per hour
- Do not review for more than 60 minutes at a time
- Use checklists

Source: Best Practices for Code Review, n.d.

Final Problem Statement

The conclusion from the preliminary theoretical findings is that the client, GameLab Oost, requires a custom, specialized, well-documented, and easy to set-up technical framework consisting of pre-configured and streamlined solutions to the most common problems their interns keep coming up against.

The main question this research paper will attempt to answer is -

> *How to create a Unity3D technical framework to streamline the prototype programming process in GameLab Oost?*

To answer the main question, the research will consider and investigate the following sub-questions:

- What are the common processes and challenges that every prototype in GameLab Oost faces?
- Is the framework easy and intuitive to use?
- Does the framework actually improve on the prototype programming process at GameLab Oost?
- Is the framework future-proof and expandable?

Concerned Parties

- GameLab Oost as the client

- Holland Casino as a secondary stakeholder
- Keesjan Nijman as both the company coach and the primary stakeholder
- GameLab Oost interns as the target audience
- Yiwei Jiang as the graduation coach
- Bram den Hond as the second graduation assessor
- Alexey Khazov as the student

Scope - Limiting Conditions and Project Boundaries

The research and graduation assignment will focus on creating a technical framework in the Unity3D game engine. The reason for focusing on Unity3D is that it is the engine GameLab Oost mainly uses for their projects and is the engine most of their interns study and work with. The assignment will last for eleven weeks and yield a framework consisting of a core with the singleton pattern and many utility and extension methods that will act as a base for the rest of the framework, an event component that will represent the observer pattern implementation, and a ui and save components as tools for interns to speed up development, that GameLab Oost can use in their projects. Furthermore, the framework will come fully documented, and with the ability to retrieve any updates with ease. Moreover, the framework will include a starter tool that adds a new project template to Unity that creates a new GameLab project with a predefined folder structure, starter scripts, and the framework pre-installed.
The framework will not include any additional recurring systems, such as localization, due to time constraints.

Chapter 1: Empathize & Define

GameLab Projects

All of GameLab’s projects fall entirely under the responsibility of student interns with various skill levels and game making experience, ranging from some to none. This research focuses on understanding and attempting to help, the programming interns in particular, create higher quality prototypes with greater ease.

Löp Wa Los

One of GameLab’s most notable projects is Löp Wa Los, a game for the ZGT hospital. The goal of the game is to help elderly people over the age of 50 to recover faster from an open chest surgery through various exercises. The game achieves this through mini puzzle games that require the user to perform different exercises to receive coins to buy rewards with. The rewards mainly constitute customization items that the user can place in their own virtual room.

Holland Casino

Another very notable project is Holland Casino. The purpose of the game is to entice millennials to visit Holland Casino for the first time. The game does this through a casino building and management simulation that contains Holland Casino games, tables and machines, alongside a questing system that teaches the player about the casino. As the user plays the game, they purchase various games and decorations from the in-game shop and place them in a grid-based expandable casino. Every placed game attracts virtual casino players that walk around the venue and play different games. Each of the virtual players has a chance of becoming addicted, and it is then the responsibility of the player to take care of them. This in turn teaches the player about responsible gambling.

GameLab Kit

Lastly, yet another interesting project GameLab is currently working on is an in-house puzzle game called GameLab Kit to help potential clients decide on what sort of game they would like the company to develop. The game handles this task through a lot of in-game dialog and a few mini-games that focus on each of the core ingredients to making a game, such as genre, art style and platform. After completing all the minigames, the results are collected in one central room and demonstrate all the choices their potential client made. This information then helps GameLab take the correct path towards developing the client's vision.

Recurring Problems

GameLab Oost is constantly reworking old projects and starting new ones. While the projects are all different, they all share certain tools and systems, and all run into the same development problems.

Unity API

Most of the development issues boil down to a poor usage of the Unity API, which results in bad performance and confusing code. Unity strives to create and has very good API, however, it is very versatile, which causes uncertainty and inconsistency in the code base.

Object References

One example of the harm caused by Unity's versatility is the method to getting references to other game objects and components from within a script. There are five main mechanisms to accomplish that:

- Finding an object by its name using Find(string name)

- Finding an object by its tag using FindWithTag(string tag)
- Finding an object by its type using FindObjectOfType(Type type) or FindObjectOfType<T>()
- Getting an object that is a part of the script's hierarchy using GetComponent(Type type) or GetComponent<T>()
- Direct reference injection through dragging and dropping in the Unity Editor inspector.

Because of both their lack of experience and the fact that most Unity tutorials make use of the drag and drop method, most programming interns at GameLab would choose direct references in the editor rather than acquiring them through code. The main idea behind this method is to make Unity designer friendly in such a way that they need only drag objects around to build gameplay. However, there are a few downsides to this practice. First, it is impossible to identify through code where references are set, making it hard to find out how and/or where they were modified. Second, scripts that expose direct reference fields in the editor may consider an un-set reference valid. This could lead to confusion when forgetting to set a reference about why is something not working, despite there being no errors displayed. Lastly, all the direct references are serialized and saved inside of Unity scene files. This means that changes to different game objects and scripts within the same scene modify the same file, causing conflict issues when using version control.

Component Properties

Unity provides developers with helpful properties to the most used components such as a game object's transform and the main camera. Unfortunately, these properties are incredibly performance taxing. The main camera Camera.main property states in the Unity documentation that it "uses FindGameObjectsWithTag internally and does not cache the result," but luckily, it also says that "it is advised to cache the return value of Camera.main if it is used multiple times per frame", as otherwise it would take a lot of processing time to repeatedly find the main camera game object in the scene, especially when there exist many game objects and done in many different scripts.

Unluckily, the transform property's documentation simply states that it is the "transform attached to this GameObject." At first glance, since the transform component exists on all game objects, intuition states that this property is most likely cached and repeated calls to it do not incur any additional overhead or performance costs. However, this is not the case, as internally, Unity caches the transform component on the native side of their engine, and marshals it into C# managed code every time the transform property is used (xVergilx, 2019).

Utilities

Nevertheless, Unity does provide many helpful and optimization-focused utilities, like the most common vector directions, some of which are:

- Vector3.zero = (0, 0, 0)
- Vector3.one = (1, 1, 1)
- Vector3.right = (1, 0, 0)
- Vector3.up = (0, 1, 0)

These properties are static and are created only once, which allows programmers to reuse them without creating additional objects in memory. However, Unity does not put much emphasis on these properties. Even their own vector tutorials do not use them. Instead, they show example code that explicitly creates new Vector3 instances (Understanding Vector Arithmetic, n.d.).
To their defense, Vector3 is a struct, which means all its instances spawn and exist on the stack, which is an incredibly fast, specialized and optimized part of the computer's memory.
Nonetheless, such examples build up poor programming habits, especially in beginners. When a new programmer falls into the habit of constantly creating new instances, and those instances, instead of spawning on the stack, go to the heap, which is a much larger, yet slower space in memory, their code ends up running very slow, results in a lot of garbage and is prone to many errors.

Duplicate Code

Recreating Unity Features

Most programming interns at GameLab do not spend much time investigating Unity's API, its existing features or learning about its best practices. Instead, eager to program, they recreate a lot of functionality that Unity already provides. For example, GameLab Kit features a minigame similar to the popular mobile game Flow. The goal of the minigame is to connect tiles together in a specific pattern on a two dimensional grid. The programming intern in charge of the minigame wrote a grid script that generates and manages tiles (See Appendix 1). However, Unity already provides a built in two dimensional tile map solution for working with, generating and displaying tiles. Therefore, the programmer wasted development time on recreating an already existing system, instead of focusing on using that system to further the project.

Creating Duplicate Systems

The duplicate code issue becomes more apparent when looking at multiple GameLab projects side by side. As already established, the client's projects share certain tools and systems. While their application may differ between different projects, their core is always the same. For example, every single one of GameLab's projects requires the game to be able to save user data and progress. There is no standard solution to this task, thus, every project group created their own system. In the case of the Löp Wa Los project, the programmers wrote a save system that

relies on the Unity SimpleJson library for serialization, which requires a lot of boilerplate code. At the same time, the customization system in that same project also relies on Json serialization. However, rather than utilizing the already existing SimpleJson library in the project, the programmer responsible for the customization system chose to use Unity's built-in JsonUtility library. Even further, many other systems that relied on Json serialization decided to include a third library by the name of Json.NET, instead of taking advantage of the already two different Json libraries in the project.

Coding Conventions

Every programmer, be it new or seasoned, comes from a different programming background. This entails different habits, knowledge, coding style and conventions. For a team with only one programmer, this does not matter, but, when the programmer has to work together with other programmers, each with their own style and conventions, the code base becomes messy, and hard for programmers to take over others' tasks and existing work.
Additionally, one programmer's bad habits have the potential to propagate throughout the entire project as they create an unclear and unsafe API. Their fellow programmers make certain assumptions based on the naming conventions in the API, and end up using it in unexpected ways, causing a lot of avoidable bugs and problems.

Menus & UI

More than that, one of the biggest features that appears in every single project in GameLab Oost is UI. Ever since Unity 5, the engine revealed an immense UI workflow overhaul. Originally, developers had to create, place and position all of the UI in code. With the new system, developers can create gorgeous menus and UI elements visually directly inside of the Unity Editor. However, as established in the Object References section before, there are many ways to set those up. The GameLab programming interns constantly struggle with this issue, and end up creating hard to modify and extend UI and menu systems. For example, the Löp Wa Los project has many UI menus with multiple pages and categories.

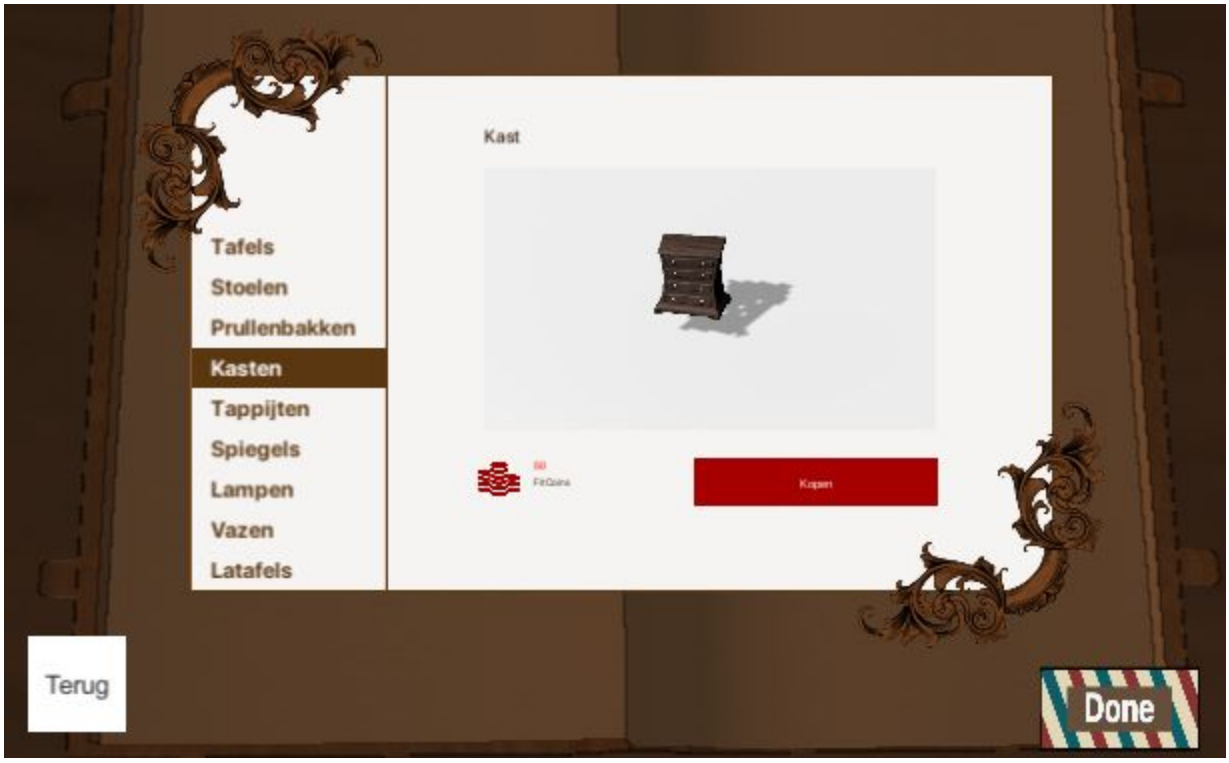

Figure 5. Shop menu in Löp Wa Los with a page back button (Terug).

Every menu has back and forward buttons to navigate between the pages, however, the way in which the buttons affect the currently active menu and category is very odd and fragile (See Appendix 2). The client needed to revisit this project and make it so that these buttons would properly appear and disappear when the player was on the first or last page of the current category of the currently active menu. However, this turned to be a not so easy task due to the nature of how the buttons interacted with the UI.

Conclusion

Beginner and even experienced programming interns at GameLab Oost constantly run into many common issues and problems during their project. The question then lies in how to solve these issues. But, before there can be a solution, there needs to be a concrete problem. Most of the recurring problems the interns face boil down to a lack of proper guidance and knowledge. Therefore, the main problem is how to compile some of the best practices for Unity, alongside a small set of helpful tools and guides to prevent the programming interns from feeling lost and attempting to recreate already existing functionality, as well as enable them to write good and consistent code.

Chapter 2: Ideate

Programming Framework

Programming Patterns

The framework would implement the well known and proven singleton and observer patterns to solve the issues of acquiring references and communicating between different objects inside Unity.

UI Framework Component

The UI specific component in the GameLab framework would build upon the core framework, making use of the singleton and observer patterns to provide the programming interns with an easy, intuitive and extensible way of identifying, referencing, and managing menus.

Save Framework Component

The GameLab Framework would include a standard game saving system, using the singleton pattern to provide an easy and globally accessible API.

Coding Conventions & Peer Review

With regards to the coding conventions issue, the best solution is to show the programming interns a standard coding style. As mentioned in the peer review section in the theoretical framework, ensuring coding conventions is one of its benefits.

The conventions come in a few different forms. The first is a document, with guidelines, the reasoning for them, exceptions and examples. This document provides the programming interns with a constant reference point to compare their code with.

Unfortunately, getting the interns to follow the guidelines is no easy task, especially ones that are already used to some standard. This is where the convention reasonings come into play, with the idea behind being that “when people understand [one’s] argument, it helps reduce skepticism borne from the unknown” (Shah, 2015). Once people reach an understanding of the relationships between the coding choices and the reasons behind them, reaching agreement becomes much easier, and they are much more open to following them (Shah, 2015).

The second form of guidelines comes as a clear and fully documented code, with a documentation website available both offline and on the framework’s github wiki, that explains what each class’ purpose is, what each function does, what parameters it expects, with hints and suggestions for when to use which method.

General Research

Moving on to the issue of programming interns recreating already existing functionality, there is no simple solution to this problem. The only way around it is thorough investigation into whether a certain functionality already exists in one form or another. This includes looking both into official Unity features, as well as custom features the programming interns created.
Furthermore, proper coding standards and guidelines would alleviate this issue by directing the interns toward writing more general, abstract and reusable code, for instance, a script that rotates generic game objects, as opposed to a gear script with rotation logic that only applies to gear game objects.

Chapter 3: Prototype

GameLab Framework

The goal of the GameLab Framework is to streamline and standardize the way programming interns at GameLab work on and create game projects. Most of the issues the company faces arise from the lack of experienced programmers or sometimes even the lack of programmers entirely. Therefore, the framework focuses on solving the biggest programming issues mentioned in the previous chapters.

Extracting Current Solutions

The graduation assignment was executed while the graduating student was also working on and acting as the lead programmer in all of GameLab's current project, with the main focus being on Holland Casino. To avoid committing the same mistake programming interns make of duplicating code, the first step to building up the GameLab framework is to analyze the solutions already created during the Holland Casino project.
The project contains many solutions to the numerous shared systems and tools between GameLab's projects, namely, singletons, menu management, event based communication, localization, building and item management. While these are all important, for the purposes of the GameLab Framework prototype, and due to time constraints, it only includes certain mechanics from Holland Casino.

Singleton Pattern

Holland Casino was designed around the Singleton pattern to allow for different components and systems to communicate between each other with much more ease. To achieve that, it has a base

Singleton class that extends from BetterMonoBehaviour, which ensures there is only ever one instance of a specific component and has an easy to access property that returns it.

Performance

The Holland Casino project contains a couple scripts that focus on solving some of the performance costs associated with the Unity API calls discussed in the Component Properties section of the first chapter.
The first is called BetterMonoBehaviour because it improves upon the existing MonoBehaviour class by caching the transform and rect-transform of every game object it is attached to.
The second is a CameraManager singleton that caches the main camera retrieved through the Camera.main API.

Extensions

One of the core game mechanics in Holland Casino is building and placing casino games and tables on tile grids. To accomplish that, the project relies heavily on Unity's Bounds struct. Therefore, to eliminate duplicate boiler-plate code and provide additional useful functionality, the project contains a bounds extensions class.
The main features of this class are:

- Rounding up bounds sizes to a whole number
- Snapping to other bounds and positions
- Preventing mathematical errors by letting the user specify the bound's main vertices through an enum
    - TopRightBack, rather than bounds.center + new Vector3(bounds.extents.x, bounds.extents.y, -bounds.extents.z).

Asides from the bounds extensions, the project includes many other extension methods, for instance, Color32 comparisons, FileInfo extensions for writing and reading to and from a file, and GameObject extensions that allow calculating bounds and ensuring that a component exists on an object.

Scriptable Objects

Holland Casino depends on scriptable objects for its questing and item systems. Scriptable objects are incredibly powerful and simple enough to use. However, they are physical asset files stored in the project, which means one must be very careful when modifying their data at runtime. Scriptable objects work on the same basis as the sharedMaterial property of a renderer in Unity. Any changes made to the sharedMaterial property directly affect the material asset, and do not reset when the developer leaves play mode in the editor.

To deal with this issue, the project contains a RuntimeScriptableObject class that extends from ScriptableObject. The class comes with helper methods to create runtime instances from an asset that can be safely modified without changing the original. It further has a unique identifier property for serialization purposes, that also enables comparison between different instances of the same asset.

Attributes and Property Drawers

There are a few editor utilities that help avoid user error. The utilities are custom attributes, akin to Unity's Range, SerializeField and HideInInspector ones, with custom editor property drawers to change the way they are displayed in the inspector.
The first attribute is Layer, which creates a dropdown field with the currently set up layers in the project, as opposed to writing a layer's name or index, which could have a typo or not exist.
The second is a scene picker attribute that performs the same function as the layer one, with the exception that it allows dragging and dropping a scene asset, rather than choosing a layer.
Finally, there is a class type reference class from the Unify Community Wiki, with an extends from and implements attributes that allow for choosing a class type from a dropdown directly in the editor.

Putting it all Together

Core Component

The core component of the GameLab Framework includes and builds upon all the solutions extracted from Holland Casino. It's focus is on solving the trouble programming interns at GameLab have with acquiring references and communicating between different systems, as well as focusing on improving their usage of the Unity API.

BetterMonoBehaviour

The BetterMonoBehaviour component uses the one in Holland Casino as a base and adds an additional CachedBounds property that calculates the world-space axis aligned bounds of the game object it is attached to whenever its transform changes.

Singleton & Manager

The Singleton base class improves upon the Holland Casino one by letting users control whether their singletons persist throughout the entire application, or get destroyed with the scene they belong to. The core further adds a Manager class that extends from Singleton, but has no additional functionality, other than helping polymorphism and acting as a specific distinguisher between basic singletons and manager classes.

Extensions

The core component includes all the extensions from Holland Casino and adds a couple extra extension classes for Lists and Transforms. The list extension focuses on lists of gameobjects, and provides a utility method to disable all gameobjects in the list, except one. The transform extension builds upon the bounds extension, and allows snapping transforms together based on their bounds.

RuntimeScriptableObject

The asset identifier property of the RuntimeScriptableObject in Holland Casino was not deterministic. Every recompilation of the project caused it to change, so, the core includes the RuntimeScriptableObject component from Holland Casino, but it uses Unity's AssetPostprocessor class to detect the asset identifier Unity generates for assets created from the RuntimeScriptableObject instances, and uses that instead of creating a custom one.

Custom Attributes

The core includes all the same custom attributes and property drawers as in Holland Casino.

Script Templates

The core provides an editor tool that can install and add new C# script templates to Unity. The templates define starting code when new scripts are made from them. To create new templates, users need to add text files to a folder called CustomScriptTemplates anywhere in the project that follow Unity's naming convention of [Priority]-[Menu Name]-[File Name].cs. The tool then automatically detects any new templates, and prompts to install them. The users also get main menu options to install, reinstall and uninstall these templates.
The core also comes with two custom script templates, one that replaces the default script template, and instead creates a new script that automatically extends from BetterMonoBehaviour, and another that automatically extends from the Manager singleton class.

Event Component

The event component of the GameLab Framework furthers the focus on solving the issue programming interns have with communicating between different systems by implementing the observer pattern in a global, easily accessible event manager that allows sending different event messages to create modular, stand-alone and decoupled components.

GameLab Event

The GameLab Event class is very basic and serves as the base class for event definitions. This class solves UnityEvents' restrictions with how many parameters they can pass, since the

programming interns can add as many variables as they need to their event definition classes. Furthermore, they can even add methods and utilities to help them deal with the event data better. The GameLab Event also defines a Consumed property that indicates whether this event should continue propagating towards any remaining handlers or not. Once a handler consumes the event, no other handlers will receive the event.

Event Handler

The event handler is similar to UnityEvent, but it adds a couple extra features and focuses on making event registration as simple as possible. The event handler is an abstract class that uses dynamic C# delegates at its base, and adds a priority setting that lets users control the order in which events are handled. Higher priority event handlers get called first, and lower ones get called last. This allows, for example, for UI to add high priority event handlers to act upon input events and prevent them from propagating further into the game.
The event handler has two derived classes, the first being a generic version that defines which GameLab Event it handles and casts the dynamic C# delegate to an Action that returns the GameLab Event instance as a parameter. The reason for the cast is that dynamic delegates are very slow, as they have no predefined information about the methods they call and the parameters they take. However, the workflow of casting the dynamic delegate in the event handler base class to a concrete delegate makes it possible to create custom event handlers that take any type and number of parameters. This is exactly what the second derived class, ParameterlessEventHandler, takes advantage of. It functions exactly like the generic EventHandler class, except that it allows for callback methods that do not care about the event information. One use case for this handler is UI that needs to refresh itself when a certain event happens in the game. The refresh methods are usually standalone and take no parameters. Therefore, instead of creating a new method whose sole purpose is to ignore the event argument and call refresh, the parameterless event handler allows registering the refresh method as a callback directly.

Event Manager

The event manager singleton ties everything together in a globally accessible API that lets the programming interns raise events, and add and remove methods as callbacks to specific event types.

Script Templates

The event component takes advantage of the custom script installer of the core component and provides a template for GameLab event classes. The template creates an empty class that extends from GameLabEvent by default.

Save Component

The save component of the GameLab Framework aims at providing a standard and streamlined way of saving and loading game data in Unity. It eliminates the programming interns' need to create custom save systems from scratch every single time.

ISaveable Interface

The ISaveable interface allows any class in the game to start saving and loading data. It defines a few basic methods that get called when the game is being saved or loaded:

- SaveGame
- LoadGame
- LoadNewGame
- LoadGameCoroutine
- LoadNewGameCoroutine

All methods, except the new game ones, have a save data dictionary parameter that the programming interns can read from or write to. The coroutine methods take advantage of Unity's coroutine architecture and allow pausing a load operation until certain conditions are met.
To make using this interface even simpler, it provides a couple extension methods that register and unregister ISaveable instances to and from the save manager. The only downside is that due to a C# limitation, these methods only appear when called through the 'this' keyword. The best way to solve this issue in the future is to take advantage of default interface methods when Unity upgrades their own custom compiler and support C# 8.

Save Slots

Every save slot maps to a specific save data file on disk. These handle serialization, and writing and reading the data to and from disk. The purpose of save slots is to allow for multiple separate save files for different players.

Save Manager

The save manager singleton brings the entire save component together and provides a globally accessible API that lets the programming interns save and load the game from anywhere within the application with as little as one line of code. It also provides methods to retrieve, delete and create new save slots.

Script Templates

The save component takes advantage of the custom script installer of the core component and provides a template that creates a serializable struct to act as save data. The programming interns can then create there the variables they need to save and load, and add instances of those structs to the save data dictionary.

Json Component

The purpose of the Json component is to provide one standard serialization library that all the projects in GameLab would use. This eliminates the issue of having multiple libraries and different APIs for the same purpose all within the same project.

Newtonsoft Json.NET

This is the most popular Json library that also has Unity support. It is very powerful and allows serializing and deserializing entire object hierarchies with a single line of code.

Converters

The Json component provides a couple Json converters that define custom logic for serializing and deserializing certain types of objects. The first is a quaternion converter that handles Unity's quaternion struct. The second is a scriptable object converter that makes sure to use Unity's ScriptableObject.CreateInstance method for creating new instances of scriptable objects instead of calling their constructor.

JsonInitializer

The JsonInitializer takes advantage of Unity's RuntimeInitializeOnLoadMethod attribute to setup the default global Json.NET serialization and deserialization settings. Most notably, it makes sure to include all type names to make deserialization hassle free, and includes the two converters discussed above.

UI Component

The goal of the UI component is to simplify the way UI communicates with the rest of the game. This solves the issue programming interns had with tying UI logic and behaviour between the game logic and other UI components.

Menu

The menu class is the core of the entire UI component. It provides the base for all menus in a game, such as popups, notifications, stores, inventories, etc. The menu class can also represent HUDs, such as player UI with different stat bars. The class comes with some useful events and

callbacks, such as when the menu is opened and closed, to allow for custom opening and closing logic. Nevertheless, users can leave out any extra logic and just use the defined defaults. In such case, the menu class automatically detects a content container object, and toggled it on and off when the menu opens and closes.
Furthermore, the menu defines a linked menu system. This system allows menus to be linked together, so that when a menu opens or closes, all of the menus linked to it also automatically open and close.
Moreover, the menu supports procedurally generating content using MenuListings (See below). The Menu class defines utility methods to create, remove, disable and destroy menu listings based on type, index and instance reference.

MenuWithTabs

The MenuWithTabs class is one utility implementation of the menu class that takes advantage of the linked menus architecture and uses them as tabs. Every menu that is linked to the MenuWithTabs instance is a tab that contains its own content, and the class comes with the ability to switch between different tabs through the CurrentTab property.

Menu Listing

Menu listings represent one piece of content that can exist in a menu. These are meant to be a standard base class for all menu content, such inventory items, shop listings, in progress quests, etc. These can be set up manually in the inspector or procedurally generated through the menu class.

Menu Asset Loader

The menu asset loader provides an interface to load menu prefabs from disk at runtime. The class itself does not provide any concrete logic and depends on users extending it and implementing the resource management architecture available in their project, such as Unity's Resources or Addressables. By default, the UI component comes with one implementation of the menu asset loader that uses Unity's Resources API to load menus from the Resources folder.

Menu Manager

The menu manager singleton provides a globally accessible way of managing and interacting with all the menus in the game. It contains a reference to all the existing menus in the game, and allows the programming interns to find, open and close menus by their type. The manager also provides helpful properties and events, such as checking and detecting when certain menus are open.

Script Templates

The UI component takes advantage of the custom script installer of the core component and provides a template that creates a new script that extends Menu by default. The template also overrides the SetupCustomMenuSettings method by default to highlight how the interns can adjust the menu settings in code.

Documentation

The goal of the documentation is to explain as much detail about the inner workings of the different classes provided as part of the framework as possible. This in turn provides the programming interns with a deeper understanding of how each class works and how it interacts with others. Some parts also provide remarks and examples to help avoid misuse and unexpected behaviour.

XML Comments

The GameLab Framework was developed using Microsoft's Visual Studio IDE. Fortunately, Microsoft has a great feature included called XML comments. These comments are similar to normal programming comments, but are more feature-rich, and even allow automatically generating documentation websites through third party tools. Furthermore, Microsoft's intellisense can parse these comments and display nice and simple tooltips as the framework is used. Therefore, every class, property and method is fully documented to provide as much insight and information as possible.

Sandcastle Help File Builder

Sandcastle Help File Builder is one of the well-known third-party tools capable of parsing XML comment files and automatically generating different types of documentation builds, for instance, HTML website and md help files.
This tool parses all the assemblies and projects assigned to it, alongside their generated XML comments and creates easily navigable and searchable documentation.

Website and GitHub Wiki

The GameLab Framework comes bundled with an HTML website containing all the documentation for it. This way, programming interns can always refer to the documentation even without an internet connection. Moreover, due to lack of time, this feature is not included in the current prototype of the framework, but in the future, the framework could expand to have every component open its respective local documentation page when pressing the Unity help button in the inspector.

Besides the website, the framework hosts all the documentation on its GitHub wiki using the md format. Thus, the documentation are always accessible for anyone part of the GitHub from anywhere, and will always contain the most up to date version.

Unity Package Manager

For a long time, Unity developers distributed libraries and assets through the .unitypackage format. However, starting with recent version of Unity, the team started developing a new utility called the Unity Package Manager (UPM). This tool allows for a much easier and faster update of targeted components and packages. UPM is still in development and does not have full third-party support, however, the Unity team is hard at work to allow other developers to add their own packages to this tool. The way developers can do this at the moment is through hosting their packages, alongside a special package manifest file, on GitHub, and manually adding the link to the repository in a specific file in the Unity project.
As this is not very user friendly, the framework comes with a GameLab project template (see below) that already has all the framework packages included.

Package Updater

When working with GitHub repositories in UPM, the tool automatically locks the packages to the GitHub commit it pulled. This is made with the intention that packages do not automatically update and break existing code that depends on one specific version of a package. Unity always provides a package manager window, however, that only works with official Unity packages, and does not support updating or removing custom GitHub packages. Therefore, the core component of the framework introduces a main menu tool that updates all the included GameLab Framework packages at the press of a button.
Using this utility, all GameLab projects can always have the latest and most up to date framework code without going through the hassle of manually downloading a .unitypackage and installing it. Furthermore, all the UPM packages are pre-compiled and are hidden away in a separate Packages tab in the project hierarchy. Thusly, they do not clutter the project and do not affect compilation times.

Future Expandability

By using UPM, future developers that work on the framework can easily update the existing code hosted on the client's GitHub. They can also add new packages with additional features and responsibilities, and all existing and future projects simply need to add a reference to the GitHub repositories and update their packages through the main menu bar.

Documentation

The only downside is that Unity has its own compiler and builds all the framework component assemblies through its own compilation pipeline. This means that there is no way to automatically include the XML comment files with the packages for them to appear in the Visual Studio IDE. The current work around for this issue is to manually place the XML comment files in the Library/Script Assemblies folder in every GameLab project.

Project Template

The aim of the project template is to provide a standard project structure shared between all of GameLab's projects, so that anyone can easily navigate and find items within it. This also eliminates chaotic project structures and attempts to guide interns towards a more organized setup.

Folder Structure

The template includes a standard project folder structure, as seen in the following figure:

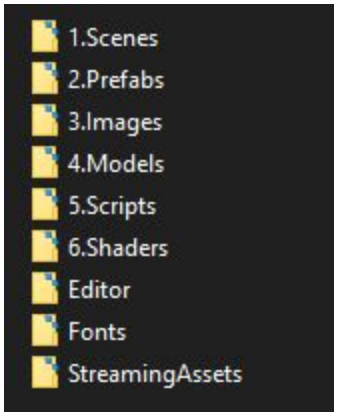

Figure 6. Project Template Folder Structure.

Starter Scripts

The project template comes with a set of starter scripts that provide some basic core functionality. One starter script is a singleton GameManager class that has the ability to pause and resume the game. These scripts are not included with the framework packages as they are meant to be modified and extended based on the needs of the specific project.

Unity Package Manager Packages

As mentioned in the Unity Package Manager section above, to include custom GitHub packages in UPM, developers need to manually modify a specific file. That file is manifest.json and is located inside the Packages folder in the project. The project template sets this file up properly to include all the GameLab Framework packages by default.

Project Template Installer

For a while now, Unity has had support for creating new projects using specific pre-included templates. It is possible to add custom project templates to Unity by placing a project structure inside the Templates folder in the Unity Editor installation location. To make this process as user friendly as possible, the framework includes a project template installer tool that automatically detects where the user has their Unity installed, and copies the GameLab project template to there. This adds the GameLab Project template to Unity, and programming interns can create new projects based on this template with the framework automatically included, as seen in the figure below:

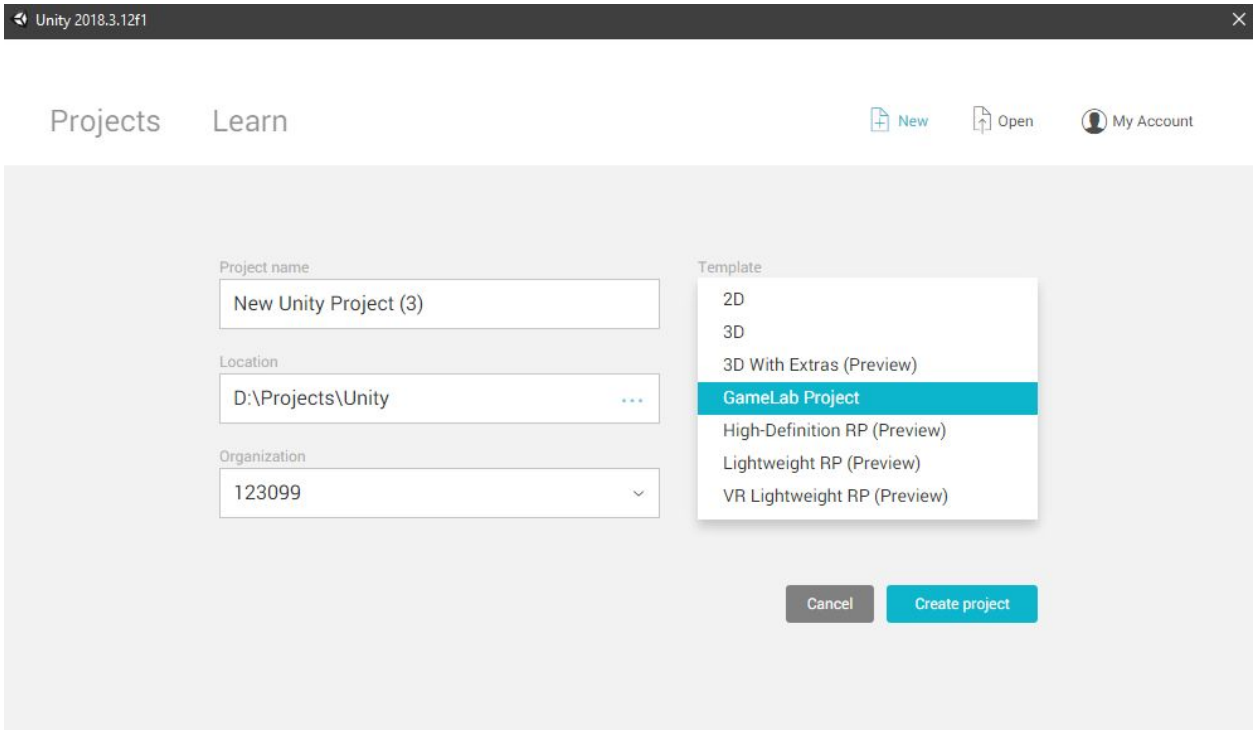

Figure 7. Creating new unity project with the GameLab Project template.

Programming Conventions Document

The programming conventions document is a collection of some of the best practices for Unity, C#, and the agreed upon conventions in the .NET community. The guidelines are based on the student's own personal programming experience, as well as on Valentin Simonov's best practices for Unity, and on Konstantin Taranov's naming conventions guidelines (See Annex 1). This document shows how to write clean and effective code that is pleasant to work with and that any developer can easily pick up and understand. The document complements every convention

with a set of examples and reasonings behind every choice in an attempt to eliminate skepticism and create agreement.

Chapter 4: Test

Now that the GameLab Framework prototype is complete, it needs to be tested to verify whether it actually solves the client's problem and streamlines the technical part of the prototype creating process in GameLab.
Since the framework is designed for programming interns with little to no experience, it is of utmost importance that all tests be carried out without the involvement of the graduating student. One of the goals of the tests is to see how easy and intuitive to use the framework is.

Implementing the Framework in Holland Casino

The first test involves updating the GameLab project from which the GameLab Framework was born. The test would check whether implementing the already existing features is easier, more intuitive, and faster using the framework. The programmer tasked with executing this test is another intern that worked together with the graduating student on the Holland Casino project. The intern would use the framework to:

- Re-implement all of the UI menus and HUDs in Holland casino
- Implement a saving system with automatic saves

The results of the test would be determined by how fast the intern was able to re-make all of the currently existing UI, how easy it was to do, and whether there was any functionality lost or potentially even gained. They will be further determined by a survey the programming intern would fill out about the framework.

Implementing the Framework in other GameLab Projects

The second test involves the remaining programming interns implementing the GameLab Framework in their current projects. The interns would install the GameLab Framework in their projects and use its components to implement any mechanics necessary, such as dialogs, pop-ups, menus, events, and game saves.
The results of the test would be determined by a survey the interns would then fill out about the framework.

Creating a Simple Project

The last test involves the programming interns, and other programmers in GameLab and affiliated companies, such as Conceptlicious, setting up and creating a small simple Unity project based on the GameLab project template. The programmers may use as many features as they want from the GameLab framework to create a basic interactable program. They will then fill in a survey about the framework to determine the results.

Framework Survey

The GameLab Framework survey is the main tool to determine whether the framework actually works, whether it is clear and well documented, and whether it is easy and intuitive to use. All tests will conclude with the participants filling in a survey, answering a set of questions about their experience using the framework.

Chapter 5: Results

Client Response

The client, GameLab Oost, is very satisfied with the final product and mention that they were never able to create such quality products in such a period of time before the framework. The product matches all of their expectations and specifications, and meets their needs. Therefore, from the perspective of the client, the framework is a success, and manages to improve their prototypes and streamline their programming process.

Holland Casino Re-Implementation Result

The Holland Casino project contains about six different menus, and a few menu tabs. According to the programmer in charge of implementing the framework in Holland Casino, he was able to remake all of the menu logic within one working day, which he considers to be fast. The process was very easy, made the code cleaner, and he managed to achieve new functionality, such as opening multiple menus at the same time, checking more specifically which menus are open, and having a greater sense of control over the menus.

Survey Results

The GameLab Framework survey results back up the client’s statement. Based on seven responses, on a scale of one for very hard to five for very easy, most of the framework

components have an average result of four for ease of use. For intuitiveness, on a scale of one for very unintuitive and five for very intuitive, the framework components also have an average of four. While this is not a large enough sample size to make a concrete conclusion, it is sufficient to estimate that most programming interns would find the framework intuitive and simple to use, which means that it is streamlined.

Updating the Solutions Based on User Feedback

The GameLab Framework survey asked the programmers to specify any feedback they had, such as confusion or limitations they experienced, or any features they were missing or would like added. Some of the feedback revolved around the project template installer and the package update process. The first wished to see the project template installer automatically detect the Unity Editor installation folder. Therefore, after receiving this feedback, this tool now attempts to detect where on the system the user has Unity installed, and still allows them to manually select a different Unity folder if they so wish. The second talked about the annoying message that pops up when attempting to update the framework packages. The message stated that due to a Unity limitation with the UPM updater, the user had to manually remove and restore focus from the Unity Editor window. Unfortunately, while Unity does allow developers to include custom packages in UPM, they expect them to modify the package relevant Json files manually. Therefore, there is no public API to force the packages to update. However, after receiving enough complaints about this message and in an attempt to make the tool more user friendly, it now automatically minimizes and restores the Unity Editor window, forcing Unity's UPM updater to kick in and start the update process.

Conclusion

The GameLab Framework implements two well known and proven to work programming patterns to help programming interns structure and write better code. To help that even further, the framework includes a set of programming guidelines and conventions, each of which comes with a set of examples and the reasons behind it. The framework builds upon the singleton and observer programming patterns to create many utility classes and methods, like the Event Manager Save Manager and Menu system. The framework focuses on ease of use and intuitiveness, trying to making some components as easy to easy as writing a single line of code. It also comes with a set of external and internal Unity tools to speed up the prototyping process and make it very simple to update and use the framework. Due to the framework's integration with Unity's Package Manager, the framework is easily extendable and future-proof.
Based on the test results, the framework is easy and intuitive to use, such that less experienced programming interns managed to create a simple unity project within a matter of a few days to even a few hours. The framework documentation is sufficiently clear and explains every aspect

of it in detail. Hence, the framework streamlines and improves the prototype programming process at GameLab Oost.

Recommendations

The GameLab Framework leaves itself easy for extension to cover any future needs. While the framework already achieves its goal, there are still some features lacking.
First, there is currently no way of installing the framework in an already existing project, other than manually including each of the framework packages in the manifest.json file in Unity.
Second, it would be great to replace the observer and singleton patterns with Scriptable Objects, based on Ryan Hipple's ideas (See Appendix 6 for more information).
Third, the current ISavable interface provides many utility methods, but they are rarely all used at once. Therefore, when Unity supports C# 8, this interface should provide default implementation for the coroutine methods, and move the registration extension methods into the interface itself.
Fourth, as Unity Package Manager support improves, or maybe through an additional custom tool, the framework should find a way to include XML comments so that Visual Studio's intellisense picks up on them and displays them as programming interns use the framework.
Lastly, due to the scope of the graduation assignment, the framework only includes the most important systems common to all of GameLab's projects. It had to exclude systems like localization and item management. The framework could get these two systems as separate component packages in the future.

References

Alba, L. D. (2016, November 07). Data Serialization Comparison: JSON, YAML, BSON, MessagePack. Retrieved June 15, 2019, from https://www.sitepoint.com/data-serialization-comparison-Json-yaml-bson-messagepack/

Best Practices for Code Review. (n.d.). Retrieved June 17, 2019, from https://smartbear.com/learn/code-review/best-practices-for-peer-code-review/

Camera.main. (n.d.). Retrieved May 06, 2019, from https://docs.unity3d.com/ScriptReference/Camera-main.html

Carr, R. (2009, June 7). Observer Design Pattern. Retrieved June 16, 2019, from http://www.blackwasp.co.uk/Observer.aspx

Clayton Industries. (n.d.). Quick Save. Retrieved June 15, 2019, from https://assetstore.unity.com/packages/tools/integration/quick-save-107676

Creative Commons License Deed. (n.d.). Retrieved June 15, 2019, from https://creativecommons.org/licenses/by-sa/3.0/

Cuelogic Technologies. (2019, February 11). Code Review Process: Best Practices. Retrieved June 16, 2019, from https://medium.com/cuelogic-technologies/code-review-process-best-practices-3eeecab26ded

Design Pattern - Singleton Pattern. (n.d.). Retrieved June 15, 2019, from https://www.tutorialspoint.com/design_pattern/singleton_pattern.htm

Devart. (n.d.). Code Review Benefits. Retrieved June 16, 2019, from https://www.devart.com/review-assistant/learnmore/benefits.html

Devart. (n.d.). Code Review Add-in for Visual Studio - Review Assistant. Retrieved June 16, 2019, from https://www.devart.com/review-assistant/

Dunstan, J. (2016, January 25). Event Performance: C# vs. UnityEvent. Retrieved June 16, 2019, from https://jacksondunstan.com/articles/3335

GameObject.transform. (n.d.). Retrieved May 06, 2019, from https://docs.unity3d.com/ScriptReference/GameObject-transform.html

Hipple, R. (2019, April 16). Game Architecture with Scriptable Objects. Retrieved June 16, 2019, from https://www.schellgames.com/blog/game-architecture-with-scriptable-objects

HonoraryBob. (2017, February 04). PlayerPrefs or save file? Retrieved June 16, 2019, from https://forum.unity.com/threads/playerprefs-or-save-file.454630/

Hutong Games LLC. (n.d.). Playmaker. Retrieved June 15, 2019, from https://assetstore.unity.com/packages/tools/visual-scripting/playmaker-368

Json.NET. (n.d.). Retrieved June 15, 2019, from https://www.newtonsoft.com/Json

Khalid, Z. (n.d.). ZUI - Menus Manager. Retrieved May 12, 2019, from https://assetstore.unity.com/packages/tools/gui/zui-menus-manager-96251

MenuItem. (n.d.). Retrieved June 15, 2019, from https://docs.unity3d.com/ScriptReference/MenuItem.html

MILLENNIAL | meaning in the Cambridge English Dictionary. (n.d.). Retrieved June 14, 2019, from https://dictionary.cambridge.org/dictionary/english/millennia

Moodkie. (n.d.). Easy Save - The Complete Save & Load Asset. Retrieved June 15, 2019, from https://assetstore.unity.com/packages/tools/input-management/easy-save-the-complete-save-load-asset-768

PlayerPrefs. (n.d.). Retrieved June 15, 2019, from https://docs.unity3d.com/ScriptReference/PlayerPrefs.html

Shah, J. (2015, June 16). The Science of Persuasion: How to Get People to Agree With What You Say. Retrieved May 12, 2019, from https://buffer.com/resources/the-science-of-persuasion

Simonov, V. (2018, December). How to set up a smart and efficient development pipeline in Unity. Retrieved May 13, 2019, from https://unity3d.com/how-to/unity-common-mistakes-to-avoid?utm_campaign=saas_global

_nurture_2018-Paid-subs-CLC-Stage-A&utm_content=2018-CLC-artist-common-mistakes-to-avoid-PRO/PLUS&utm_medium=email&utm_source=Eloqua

Singleton. (n.d.). Retrieved June 15, 2019, from http://wiki.unity3d.com/index.php/Singleton

Taranov, K. (2019, April 08). C# Coding Standards and Naming Conventions. Retrieved May 13, 2019, from https://github.com/ktaranov/naming-convention/blob/master/C# Coding Standards and Naming Conventions.md

Understanding Vector Arithmetic. (n.d.). Retrieved May 06, 2019, from https://docs.unity3d.com/Manual/UnderstandingVectorArithmetic.html

Unify Community Wiki. (n.d.). Scripts. Retrieved June 15, 2019, from http://wiki.unity3d.com/index.php/Scripts

Unite Austin 2017 - Game Architecture with Scriptable Objects[Video file]. (2017, November 20). Retrieved June 15, 2019, from https://www.youtube.com/watch?v=raQ3iHhE_Kk&feature=youtu.be

UnityEvents. (n.d.). Retrieved June 16, 2019, from https://docs.unity3d.com/Manual/UnityEvents.html

xVergilx. (2019, January 15). Cache transform: Really needed? Retrieved May 06, 2019, from https://forum.unity.com/threads/cache-transform-really-needed.356875/

Appendices

Appendix 1 - Custom Grid Class

The programmer in charge of the GameLab Kit minigame created their own grid class that manages tiles in a two dimensional array, rather than using the already built in Tilemap solution Unity provides.

```
public class Grid : MonoBehaviour
{
    private Tile[,] coordinates;
    private Color32[] pixels;
    private int pixelCount = 0;
    private List<GameObject> currentGrid = new List<GameObject>();
    private int gridIndex = 0;
    [SerializeField] private GridMapStruct[] gridMaps;
    [SerializeField] private GameObject tilePrefab;

    private void Start()
    {
        CreateGrid();
    }

    public void NextGrid()
    {
        foreach(GameObject tile in currentGrid)
        {
            Destroy(tile);
        }
        gridIndex = (gridIndex + 1) % gridMaps.Length;
        CreateGrid();
    }

    private void CreateGrid()
    {
        pixels = gridMaps[gridIndex].GridMap.GetPixels32();
        coordinates = new Tile[gridMaps[gridIndex].GridMap.height,
gridMaps[gridIndex].GridMap.width];
        pixelCount = pixels.Length;

        for (int y = 0; y < gridMaps[gridIndex].GridMap.height; y++)
        {
            for (int x = 0; x < gridMaps[gridIndex].GridMap.width; x++)
            {
                int index = (y * gridMaps[gridIndex].GridMap.width + x);
                SpawnTile(new Vector2Int(x,y), pixels[index], tilePrefab);
```

```
            }
        }
    }

    private void SpawnTile(Vector2Int gridPosition, Color32 color, GameObject tile)
    {
        GameObject tileObject = Instantiate(tile, new Vector2(0,0), Quaternion.identity,
this.transform);
        Tile tileReference = tileObject.GetComponent<Tile>();
        currentGrid.Add(tileObject);
        RectTransform rectTransformInstance = tileObject.GetComponent<RectTransform>();
        rectTransformInstance.anchorMin =
            new Vector2((gridPosition.x * (1.0f / gridMaps[gridIndex].GridMap.width)),
gridPosition.y * (1.0f / gridMaps[gridIndex].GridMap.height));
        rectTransformInstance.anchorMax =
            new Vector2(((gridPosition.x + 1) * (1.0f / gridMaps[gridIndex].GridMap.width)),
(gridPosition.y + 1) * (1.0f / gridMaps[gridIndex].GridMap.height));
        rectTransformInstance.offsetMin = rectTransformInstance.offsetMax = Vector2.zero;
       tileReference.ChangeState(color, true, false);

        if(color.Compare(gridMaps[0].AccessibleColour))
        {
            tileReference.CurrentTileType = TypeOfTile.Accessible;
        }

        for (int i = 0; i < gridMaps[gridIndex].TubeColours.Length; i++)
        {
            if (color.CompareIgnoreAlpha(gridMaps[gridIndex].TubeColours[i]))
            {
                if (color.a <= alphaCutoffPoint)
                {
                    tileReference.CurrentTileType = TypeOfTile.EndTube;
                }
                else
                {
                    tileReference.CurrentTileType = TypeOfTile.StartTube;
                }
            }
        }

        tileReference.CheckState();
        coordinates[gridPosition.x, gridPosition.y] = tileReference;

        if (color.Compare(gridMaps[gridIndex].InAccessibleColour))
        {
            tileReference.CurrentTileType = TypeOfTile.Inaccessible;
            tileReference.CheckState();
            return;
        }
        tileReference.OnMouseHoover += OnTubePlaced;
```

```
    }
```

Appendix 2 - Löp Wa Los UI Page Buttons Script

This code sample demonstrates very messy UI management code that is prone to bugs and issues through the way it acquires references to the currently active category of the current open menu.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class ItemPageButton : MonoBehaviour
{
    public GameObject backButton, forwardButton;
    public GameObject categoryParent;

    private GameObject category;
    public int pageCount = 0;
    private int maxPage = 0;

    public void ChangePage(bool countUp)    //give the ChangePage void
a bool to check if the button will count up or down
    {
        for (int i = 0; i < categoryParent.transform.childCount; i++)
//loop through all the child objects in the categoryParent
        {
            maxPage = i;
            GameObject obj =
categoryParent.transform.GetChild(i).gameObject;
            if (obj.activeInHierarchy)  //check if the category is
active in the hierarchy
            {
                category = obj;
            }
        }

        if (countUp) //if the button's bool is true
```

```
        {
            //add 1 up to the pageCount
            if (maxPage != pageCount)
            {
                pageCount++;
                SetPagesUnactive(category);
            }
        }
        else    //else if the button's bool is false
        {
            //subtract 1 to the pageCount
            pageCount--;
            SetPagesUnactive(category);
        }
    }

    private void SetPagesUnactive(GameObject category)
    {
        int pagesAmount = category.transform.childCount;        //make
an int for the amount of pages in the category
        GameObject[] itemPages = new GameObject[pagesAmount];   //make
an array for the itemPages
        for (int i = 0; i < pagesAmount; i++)
        {
            itemPages[i] = category.transform.GetChild(i).gameObject;
//set all the pages unactive
            itemPages[i].SetActive(false);
        }
        backButton.SetActive(pageCount < 1 ? false : true); //if the
pageCount is lower than 1 set the backButton unactive, if its higher
that 1 (meaning you are on the second page or higher) turn it on
        forwardButton.SetActive(pageCount >= pagesAmount - 1 ? false :
true);   //if the pageCount is higher or equal to the pagesAmount -1
set the forwardButton unactive, if its lower turn it active

        itemPages[pageCount].SetActive(true);   //set the correct page
active
    }
```

```
}
```

Appendix 3 - Code Review Best Practices

This appendix demonstrates some of the best code reviewing best practices and the reasonings behind them. The source for all these reviews is Best Practices for Code Review, n.d., however, many other sources, such as Cuelogic Technologies, 2019, share similar practices.

Review fewer than 400 lines of code at a time

One study of a Cisco Systems programming team showed that developers should only review between 200 and 400 lines of code, since beyond that amount, the brain's effectiveness decreases, and its ability to detect bugs and defects diminishes.

Take your time. Inspection rates should under 500 LOC per hour

Reviewing code at a high rate decreases defect detection. The most effective code review happens at a slower pace for a limited amount of time.

Do not review for more than 60 minutes at a time

Engaging in an activity that requires concentrated effort results in performance drop after about 60 minutes. Studies show that taking breaks can greatly improve work quality. Therefore, more frequent reviews are recommended.

Set goals and capture metrics

The team should decide on how to measure the effectiveness of peer review and set some tangible goals.

Authors should annotate source code before the review

Annotations guide the reviewer through the changes, showing which files to look at first and defending the reason behind each code modification. Annotations ease the process and even help the author find additional errors before the peer review even begins.

Use checklists

Checklists are the most effective way to eliminate frequently made errors and to combat the challenges of omission finding. These also provide the team with clear expectations for each type of review.

Establish a process for fixing defects found

Besides reviewing and detecting any defects in the code, there still needs to be a method behind fixing the found issues. The best way to ensure that defects are fixed is to use a collaborative code review tool, such as Review Assistant, which can log bugs, discuss them with the author and approve changes in the code.

Foster a positive code review culture

Criticism is hard to handle. To ensure code review is successful, the team and management need to create a culture of collaboration and learning.

Naming

Keep a steady watch that proper naming conventions are used and followed as per the review plan that is to be agreed upon by the entire team.

Appendix 4 - Singleton Pattern

The singleton pattern is “a way to ensure a class has only a single globally accessible instance available at all times” (Singleton, n.d.). It is “one of the simplest design patterns...[and] provides one of the best ways to create an object” (Design Pattern - Singleton Pattern, n.d.). The pattern involves creating a private constructor so that only the singleton class can create instances of itself, and then declaring a private static instance variable that holds a reference to the single instance of the class in the game. The singleton then also provides a public getter to this static instance so that other parts of the code could access it.
Since the singleton pattern can be applied to multiple different classes, but the implementation is always the same, the ideal solution is to create a base Singleton class that classes that want to use the singleton pattern can extend from (Singleton, n.d.).

Advantages

According to Ryan Hipple and the Singleton pattern page on the Unify Community Wiki, some of the benefits of the singleton pattern include:

- Developers can access anything from anywhere
- Has a persistent state throughout the lifetime of the program
- Easy to understand and plan
- Supports interfaces and inheritance.

It is very simple for programmers to create a singleton, like a PlayerManager that provides access to the players in the game, and then simply make an EnemyManager that accesses the players in the PlayerManager (Unite Austin 2017 - Game Architecture with Scriptable Objects, 2017). With this pattern, the team can just “start building things without having to put a lot of thought into how those systems communicate with each other” (Unite Austin 2017 - Game Architecture with Scriptable Objects, 2017).
The Unify Community Wiki provides a great implementation of the singleton pattern for MonoBehaviours in Unity, however, it does not have all the requirements necessary for the GameLab Framework (See Core Framework chapter for more information).

Disadvantages

The singleton pattern does indeed solve many common programming problems, such as getting references between different systems, it does come with its own share of disadvantages:

- The pattern results in rigid connections, as different singletons refer to each other.
    - This makes the different singletons not modular.

    - Any modifications to the singleton could easily break all other dependent code.
- Singletons can only return one instance of one type, making polymorphism impossible.
- Singletons are not testable because of the above mentioned disadvantages

Appendix 5 - Observer Pattern

The observer pattern “defines a [loosely coupled] link between objects so that when one object’s state changes, all dependent objects are updated automatically” (Carr, 2009). This pattern “defines a manner for controlling communication between classes or entities” through two main roles - a subject and an observer (Carr, 2009). The subject is an object that publishes changes to its state, while the observers are objects that “depend upon the subject [and] can subscribe to it so that they are immediately and automatically notified of any changes to the subject’s state” (Carr, 2009).

Without the observer pattern, the subject needs to know about other specific parts of the application to call certain logic that should happen when its state changes. As an example, consider a game with a player object that has a health variable. When the health of the player reaches a value of zero, the game needs to play a death sound, and display a game over message on the screen. Without the observer pattern, the player object would need to directly reference the sound and UI components, and call each of them directly when its health drops to zero. This creates tight coupling between the player and other parts of the game. Tight coupling may not seem like a great issue at first, but any changes to functionality require revisiting the player class and updating its logic, which is a potential for bugs, especially at a later stage of development. As another example, the game requires that the player also play a particle effect when they die. To accomplish that, the developer needs to directly reference the particle effect, and update the player’s death logic to also play it. On the other hand, with the observer pattern, each of the components that wish to act upon the player’s life state change, need to simply register to the player, and run their own logic when the state change happens. At the same time, the player object only needs to notify that their life state changed and that they have died. The player does not care who or if anyone is listening or responding to this state change. Thus, developers can add, remove or modify existing functionality based on the player’s death event, without the need to revisit or modify the player code, which also prevents causing new bugs.

.NET Framework Events

The .NET framework, which is a set of libraries written by Microsoft that Unity and the C# programming language use, implements a version of the observer pattern in their event model (Carr, 2009). Rather than creating explicit base classes and/or interfaces, and implementing them in sub-classes, the framework allows creating events with a specific method signature anywhere in the code. Other classes can then directly subscribe methods to these events as long as their signatures match the ones the events define. This makes it very easy and effortless to define a set of events in a subject class, and then invoke them when its state changes, with observer classes listening and acting upon them.

Unity's Event System

Since events are very prominent and important in game development to create a clean, extensible and maintainable code-base, Unity provides two built-in event systems.

UnityEvent

The first of the event systems Unity provides is built on top of .NET Framework's event implementation, and is packaged in a simple class called UnityEvent. UnityEvents "are executed from code like a standard .net delegate," and can be configured directly from the inspector (UnityEvents, n.d.). This second characteristic gives them a great advantage over normal C# events, as it allows for even more decoupling, through setting up and injecting references in the unity inspector without writing any code. On the other hand, such a setup makes it much harder to debug the game, as IDEs no longer have any information about where certain functions are references or called from, since they are serialized in the inspector and are only invoked at runtime.
Furthermore, while UnityEvents use C# delegates under the hood, they are much slower than standard C# events, with numbers ranging between 2.25x and up to 40x slower (Dunstan, 2016). Moreover, UnityEvents require extending and creating a new class for every event definition that needs to define arguments, and the inspector can only serialize UnityEvents with up to one argument (Unite Austin 2017 - Game Architecture with Scriptable Objects, 2017). On the other hand, C# events can define an unlimited amount of arguments, either through a one-line custom delegate definition or directly in the event definition through one of the already existing generic delegates available in the .NET framework, such as Action.
Therefore, UnityEvents are a great tool for certain use cases, like simple message events that happen infrequently, but are not as good as the built in C# events.

Unity Event System

With the introduction of the new overhauled UI system in Unity 5.0, the Unity team also introduced an EventSystem component alongside many different event handling interfaces, the likes of IPointerDragHandler, to handle pointer drags, and IPointerDownHandler, to handle pointer clicks. This event system is very powerful and simple to use. Developers need to simply implement one of the many supported interfaces on a MonoBehaviour and have a raycast target, such as a collider, on the game object with the that MonoBehaviour.
Unfortunately, this system is too specific and has main disadvantages.
The first is that the system "is a way of sending events to objects in the application based on input" (UnityEvents, n.d.). While the system is incredibly powerful and simplifies the process of handling different types of input, such as mouse clicks, touch, and keyboard, it does not allow passing and handling custom game events, like player death, quest completion, etc.

The second is that the system only works with MonoBehaviours, meaning that plain old C# classes cannot listen to and handle events. This forces developers to create components for everything, even if it does not make sense for an object to be a component, such as, for example, a quest task. Quest tasks only represent data and need to update their state based on events in the game, to see whether they are completed or not. It does not, therefore, make sense for them to be MonoBehaviours.

Conclusion

Despite Unity providing developers with great event tools, they are either lacking in functionality or are too restricting. Therefore, the GameLab framework would need a custom observer pattern implementation based on the already existing C# events that allows handling any and all types of events, with any number of arguments, from any class, that has little to no impact on performance.

Appendix 6 - Game Architecture With Scriptable Objects

Ever since Unity released Scriptable Objects around 2011, many developers, including Ryan Hipple in particular, looked into, experimented with and exploited them (Hipple, 2019). There are a lot of different ways to use Scriptable Objects, but the Unity manual is unfortunately too abstract, and only states that they are data containers for saving large amounts of data and that their main use cases are:

- Saving and storing data during an Editor session
- Saving data as an Asset in your Project to use at run time

Fortunately, thanks to the many efforts of talented developers in the Unity community, they have figured out very useful patterns revolving around Scriptable Objects that put a lot more power in the hands of designers, decouple systems and eliminate rigid references, help create a more modular code and make it easier to edit and build games.
One very popular source of these patterns is a Unite 2017 talk by Ryan Hipple from Schell games. In this talk, Hipple starts off with talking about the singleton pattern, its advantages and disadvantages, and how scriptable objects can solve some of its disadvantages while still keeping some of its benefits (Unite Austin 2017 - Game Architecture with Scriptable Objects, 2017).

Variable Pattern

The variable pattern aims at keeping the benefit of being able to access a certain singleton's variable with ease from anywhere in the game, for example, the game HUD displaying the player's health as a bar. Using the singleton pattern, the UI code is directly coupled to the singleton that contains a reference to the player, making it less modular and always dependent on that singleton existing in the game to function. In a situation where a designer wishes to test the UI, they must also include the singleton that contains the reference to the player. Furthermore, that singleton might have dependencies on other singletons, meaning that the designer has to also include them. Even further, those dependencies may have dependencies of their own, and by the time the designer finishes placing all the dependencies in their test scene, they have recreated the entire game, and their debugging became exponentially more difficult (Unite Austin 2017 - Game Architecture with Scriptable Objects, 2017). The variable pattern solves this issue by having a stand-alone scriptable object that represents one single value. Components can then reference this scriptable object to both read from and write to. Since Unity makes sure that Scriptable Objects always exist in the game's memory, there are no longer any additional dependencies other than the variable Scriptable Object itself. In other words, the UI component can reference and read the data from a player health variable Scriptable Object, and display that. It does not care about how and where that variable changes. This lets designers test their

components in isolated environments, and even manually modify the values of the Scriptable Object variables they are using to make sure their logic is working as expected (Unite Austin 2017 - Game Architecture with Scriptable Objects, 2017).

Runtime Sets

The equivalent pattern to singleton managers with Scriptable Objects is Runtime Sets. In short, Runtime Sets are scriptable objects that contain a list of a specific type of object. Components that need to be part of a set, take in a reference to it, and then register and unregister themselves from it. At the same time, components that want to know about and operate on a specific set, take in a reference to it and perform any operations they need on it (Unite Austin 2017 - Game Architecture with Scriptable Objects, 2017). In such a manner, there exists a global persistent state of objects that can be accessed from anywhere at any time without the disadvantages of managing a singleton and its associated race conditions.

Event Pattern

The event pattern is very similar to the variable pattern, but solves a completely different problem. The pattern allows designers to create game events and hook up responses to those events without writing any code. The event pattern works just like the variable pattern, where designers create a separate Scriptable Object for every event and pass them around as references to components in their scenes and prefabs (Unite Austin 2017 - Game Architecture with Scriptable Objects, 2017). This allows designers to create new events at need without bothering the programmers, and more than that, eliminates the need for a singleton event manager class that is in charge of storing and managing all the different classes that listen to and handle events.

Enum Pattern

The enum pattern is very similar to the event pattern. Normally, enums can only be created through code, and any changes to them, such as adding, removing and reordering elements requires revisiting the code-base. This can become a large issue later into development as teams may not have programmers with time to update the enums, resulting in production bottlenecks. Even worse, data, such as save files, could break when an enum changes, because enums are usually serialized based on their index, and any changes to the enum would invalidate the serialized data. The enum pattern mimics the goal of enums in programming languages, which is to create a strongly typed identifier for a specific value. This pattern replaces the code-driven enum identifiers with Scriptable Objects, where each instance represents an enum value. In such a manner, order no longer plays a role, and designers can easily create and remove “enum” values from within the editor. They can then pass these values on as references to various components in the game and use them as identifiers to drive behaviour (Unite Austin 2017 - Game Architecture with Scriptable Objects, 2017).

Disadvantages

The various Scriptable Object patterns have many advantages, however, they also come with some great disadvantages that could do more harm than good.

First of all, all of the above mentioned patterns are aimed mostly at MonoBehaviours that exist within a prefab or a scene. This may not seem like a big disadvantage at first, however, when considering the event pattern for example, this means that plain old C# classes cannot listen to, handle or raise events. They must, instead, rely on being referenced by MonoBehaviours and being fed Scriptable Object events. On the other hand, with a singleton event manager, any class from anywhere in the code can always access the event manager, raise events, and register itself as a listener, and some of its methods as handlers.

Second of all, unlike explicit reference set up in code, references to Scriptable Objects created through the Unity inspector are only ever initialized at runtime. Due to that, the debugging tools of IDEs have no information about which functions call which other functions, or where in the code certain variables change. This, in turn, makes it much harder to debug and trace certain bugs, since it is more difficult to pinpoint the starting point of a specific operation that could have gone wrong (Unite Austin 2017 - Game Architecture with Scriptable Objects, 2017). Therefore, the Scriptable Object patterns require a lot of pre-thought and planning to avoid bugs that are hard to trace, something that is difficult for a less experienced programmer to do.

Lastly, and most importantly, Scriptable Objects are serialized and stored on disk. That means that any changes to the values of the original asset in the editor, even during play mode, get saved (Unite Austin 2017 - Game Architecture with Scriptable Objects, 2017). In certain cases, this is an advantage, such as being able to tweak and balance the game at runtime. However, in most other cases, this is a big disadvantage, as poor handling of the Scriptable Object asset references could result in wrong values and unexpected bugs, which are hard to trace and take a while to find and fix. Avoiding such issues requires a lot of discipline, something that, once again, is difficult for a less experienced programmer to do.

Conclusion

Despite the disadvantages associated with Scriptable Objects, they solve many problems developers face often, and bring out the true power of Unity. Unfortunately, due to the nature of the experience the interns at GameLab have, the disadvantages may hinder the quality of their projects. Therefore, this graduation assignment will focus on creating a more intuitive and easy to use programming-based framework that relies on older, yet trusted patterns, like the singleton and observer ones, with some safeguards to eliminate some of their own disadvantages. Nevertheless, Scriptable Objects would be a great addition to the framework in a later point in time.

Annexes

Annex 1 - Programming Guidelines

This document demonstrates all the programming guidelines and conventions the graduating student created for GameLab as part of his framework.

PascalCasing

Use PascalCasing for all class names, all method and function names, all properties, all enum elements and only public variables.

PascalCasing makes code easier to read, especially when accessing members of a class through the dot operator.
It is also the C# and .NET community accepted convention.

Examples

Correct: public void UpdateAnimationState() { }

Incorrect: public void updateAnimationState() { }

Correct: private int GetCurrentTabIndex() { }

Incorrect: private int getCurrentTabIndex() { }

Correct: public static readonly int MaxSaveSlots;

Incorrect: public static readonly int MAXSAVESLOTS;

camelCasing

Use camelCasing for method arguments and non-public variables.

camelCasing is a quick way to differentiate between public and private variables, and helps solve unclear naming issues when a variable and a class share the same name, such as Tile tile, vs. Tile Tile.

Examples

Correct: private int currentTileMapIndex;

Incorrect: private int CurrentTileMapIndex;

Hungarian Notation

Do NOT use hungarian notation.

Hungarian notation was useful before IDEs, and helped identify the type of every variable from a quick glance. However, nowadays, with IDEs being powerful and providing all the necessary information with the hover of a mouse, this notation just makes code uglier and harder to read.

Examples

Correct: int counter;

Incorrect: int iCounter;

Correct: public int AddNumbers(int num1, int num2);

Incorrect: public int AddNumbers(int pNum1, int pNum2);

Clear Names

Use clear names the describe the purpose and/or meaning of variables and methods.

Clear names make it much easier to rationalize and understand a piece of code and help fix bugs and logic errors. They also allow other members of the team to quickly dive and work on your code without having to rely on the context and the logic of the code to understand what the purpose of a variable is or what a method does.

Examples

Correct: float amplitude;

Incorrect: float A;

Correct: private void NextLayer();

Incorrect: private void SetLayer();

Booleans

Name booleans with an affirmative phrase and prefix them with question words whose answer is yes or no, such as has, is, do, was.

Boolean prefixes make it easier to distinguish a boolean from non-booleans and instantly clear what its value represents. The affirmative phrases avoid bugs where a programmer gets the wrong value due to force of habit because the boolean variable is inverted.

Examples

Correct: bool isFalling;

Incorrect: bool falling;

Correct: bool shouldBeDestroyedOnLoad;

Incorrect: bool shouldNotBeDestroyedOnLoad;

Abbreviations

Avoid abbreviations, write full names and do not shorten words.

Exceptions

Commonly known abbreviations, such as ID, HTTP, URL.

Abbreviations make code harder to read, and may not always be obvious to everyone. Some may understand what is meant with pwd, but others may not. Always prefer to be clear in your meaning rather than brief.

Examples

Correct:	string userPassword;
Incorrect:	string usrPwd;
Also Correct:	int ID;

Underscores - Snake_Casing

Do not use underscores for variable names.

The established convention in the C# and .NET community is to use mostly PascalCasing and sometimes camelCasing (See PascalCasing and camelCasing chapters).
Snake casing also makes it hard to find and search for certain variables and having different conventions in the code at the same time makes it harder to work with other team members.

Examples

Correct: private string userPassword;

Incorrect: private string _userPassword;

Correct: public const UserSaveDataPath;

Incorrect: public const User_Save_Data_Path;

Access Modifiers

Always specify the accessibility of any declared variables, classes, enums, etc.

Exceptions

Locations in code where access modifiers cannot be specified, such as inside a method.

Access modifiers instantly make it clear how and where a method or a variable could be used. For example, the assumption is that no access modifier means that a variable is private by default, but it is actually internal.

Examples

Correct: `private int health;`

Incorrect: `int health;`

Correct: `private void Start() { }`

Incorrect: `void Start() { }`

Default Values

Always assign default values to variables and class members.

It is not always obvious what value a variable has by default, and sometimes the default values are not correct, such as a boolean variable that should actually be set to true by default is false because it was never assigned a default value. This also helps avoid certain null reference exceptions when working with containers, such as dictionaries and lists.

Examples

Correct:
```
private int health = 100;
```

Incorrect:
```
private int health;
```

Correct:
```
public int GetTabIndex(Tab tab)
{
        int index = -1;
        ....
}
```

Incorrect:
```
public int GetTabIndex(Tab tab)
{
        int index;
        ....
}
```

If a string variable needs to be initialized as an empty string, use string.Empty.

Even though "" and string.Empty are equal to each other, the former creates a new object every single time, while the latter is a constant variable that is only created once at startup. string.Empty makes it so that your code does not generate any potential additional garbage.

Examples

Correct: `private string name = string.Empty;`

Incorrect: `private string name = "";`

Returning Lists From Functions

Always return an empty list, set, or array instead of a null value from a function when these are empty.

When creating a function such as Get that collects a set of items based on specific requirements, if there are no items found, the function should return an empty set instead of null. This avoid boilerplate code that requires users of your function to check for null all the time and prevents rogue null reference exceptions.

Examples

Correct:

```
public T[] GetComponents<T>()
{
        List<T> componentsFound = new List<T>();
        ...
        return componentsFounds;
}
```

Incorrect:

```
public T[] GetComponents<T>()
{
        List<T> componentsFound = new List<T>();
        ...
        if(componentsFound.Count == 0)
        {
                return null;
        }

        return componentsFounds;
}
```

Properties

Properties that only execute one line of code should be defined using the new C# 6 expression body definition style.

Expression body definitions keep the code shorter and concise by avoiding unnecessary indenting and curly braces. While this reason could also be applied to if statements, and loops, there is no danger in attempting to add more lines of code to expression body definitions like there is with if statements and loops, because the extra code will not be in a valid location, and the compiler will throw an error.

Examples

Correct:
```
public string Name => “John”;
```

Incorrect:
```
public string Name { get { return “John” } }
```

Correct:
```
private float ultimateCharge;
public float UltimateCharge
{
        get => ultimateCharge;
        set
        {
                ultimateCharge = value;
                RaiseUltimateChargeChangeEvent();
        }
}
```

Incorrect:
```
private float ultimateCharge;
public float UltimateCharge
{
        get
        {
                return ultimateCharge;
        }
        set
```

```
        {
            ultimateCharge = value;
            RaiseUltimateChargeChangeEvent();
        }
    }
```

Variables that back up a property must have the same name as the property, with the only difference that the backing variable is camelCased.

Having the same name for the property and its backing variable makes it easy to understand which variable the property modifies and helps avoid bugs caused by using the wrong variables.

Examples

Correct:
```
private int health;
public int Health => health;
```

Incorrect:
```
private int health;
public int CurrentHealth => health;
```

Types

Use the simplest variable type notation.

Using the simplest type simplifies the code and avoids specific platform dependencies and/or performance issues, such as integers being 32 bits on 32bit systems and 64 bits on 64bit systems.

Examples

Correct: `int currentRotationAroundY;`

Incorrect: `int32 currentRotationAroundY;`

Always use explicit type names. ***Use of var is punishable by death****.*

Explicit type names allow you to easily understand what a variable is and what it represents. More than that, explicit types let you instantly see what sort of value a method returns and allow you to reason and debug code with much more ease.

Examples

Correct: `Dictionary<string, Item> items;`

Incorrect: `var items;`

Interfaces

Prefix Interfaces with a capital I.

This may seem like a violation of the hungarian notation rule, but this is a widely well-accepted convention in the C# and .NET community. Breaking this convention risks alienating other developers that are very used to it, such as new team members.

Examples

Correct: public interface IManager { };

Incorrect: public interface Manager { };

Curly Braces

Vertically align matching curly brackets. This includes classes, structs, enums and if-statements and loops, even if they only contain a single line of code.

Unlike expression body definitions, the curly braces, even for one-liners, make it clear which part of the code belongs to which statement and what its scope is. Furthermore, they help prevent bugs when someone wants to add extra lines of code to a statement, especially in a more complicated method that contains a lot of indentations, and forgets about the curly braces, making their new code execute as part of the wrong scope and result in unexpected behaviour.

Examples

Correct:
```
if(player.IsAlive)
{
    AttackPlayer();
}
```

Incorrect:
```
if(player.IsAlive) AttackPlayer();
```

Correct:
```
public enum Direction
{
    Left,
    Right,
    Up,
    Down
}
```

Incorrect:
```
public enum Direction { Left, Right, Up, Down }
```

Source Files

Name source files according to their class names.

Naming source files after the class names they contain makes it easier to navigate through the project and find where specific code resides. Additionally, Unity components will not work unless the source file name and the component class name match.

Examples

```
public class Player { }
```

Correct: Player.cs

Incorrect: PlayerScript.cs

Do not put more than one root class, struct, interface, enum, etc. in a source file.

Exceptions

The source file may contain two root classes that have the same name, but one is a generic version of the other.

Just like with naming classes the same as their source files, the second root class, struct, etc. in a file violates that convention, making it hard to find in the project, and not working in Unity if it is a component.

Examples

Correct: **Room.cs**

```
public class Room
{
        private RoomCategory category = RoomCategory.Default;
}
```

RoomCategory.cs

```
public enum RoomCategory
{
	Default,
	Disco
}
```

Incorrect: **Room.cs**

```
public enum RoomCategory
{
	Default,
	Disco
}
public class Room
{
	private RoomCategory category = RoomCategory.Default;
}
```

Also Correct: **Room.cs**

```
public class Room
{
	public enum Category
	{
		Default,
		Disco
	}

	private Category category = Category.Default;
}
```

Correct: **EventHandler.cs**

```
public abstract class EventHandler { }
public class EventHandler<TEvent> : EventHandler { }
```

Incorrect: **EventHandler.cs**

```
public abstract class EventHandler { }
public class ParameterlessEventHandler<TEvent> : EventHandler { }
```

Single Concept

Pick one word per abstract concept and stick with it.

When starting to have a really big code base, the team might start failing to be consistent in their concepts. This might lead to something like Fetch, Get and Retrieve be equivalent methods in different classes.
This could be disastrous in cases like where a programmer uses Fetch to call an API that populates an object with data and Get to just return the value of some property. When these two start getting confused, the programmer always needs to check the behaviour of a method to make sure it does what they think it does.

Examples

Correct:

```
public class RoomManager()
{
        Room GetRoom();
        void ClearRooms();
}

public class Shop()
{
        Item GetItem();
        void ClearItems();
}
```

Incorrect:

```
public class RoomManager()
{
        Room RetrieveRoom();
        void RemoveRooms();
}

public class Shop()
{
        Item FetchItem();
        void DestroyItems();
}
```

Name Repetition

Do NOT repeat the name of a class or enum in its members.

The class or enum already makes it clear what the context is and what certain methods or members represent, since they exist inside that class. Therefore, repeating the class or enum name adds no value to the code, except making it longer.

Examples

Correct:

```
class Employee
{
        public void Get();
        public void Delete();

        public void AddNewJob();
}
```

Incorrect:

```
class Employee
{
        public void GetEmployee();
        public void DeleteEmployee();
}
```

Class Structure

Declare all variables and nested structures at the top of the class in the order below. Every group of elements (★) always starts with public items, then protected, private and internal.

Exceptions

Backing variables should always be one line above the property they are backing.

Having the same structure in all classes makes it very easy and quick to navigate between different classes in the project and find specific parts of the code, such as events and static variables.

Order

- ★ *Nested Structures*
  - *Classes / Structs*
  - *Enums*
- ★ *Group 1*
  - *Const Variables*
  - *Static Variables*
  - *Delegates*
  - *Events*
- ★ *Group 2*
  - *Properties*
  - *Variables with the SerializeField attribute*
  - *Variables*
- ★ *Initialization Methods*
- ★ *Other Methods*

Examples

Correct:

```
public class Player
{
    public enum MovementState
    {
        Walking,
        Falling,
        Flying,
        Swimming
    }

    public const bool ShowDebugInformation = true;
    public static readonly DebugRayColor = Color.red;

    public delegate void MovementDelegate(MovementState);

    public event Action<int> OnDamageTaken;
    public event MovementDelegate OnMovementStateChanged;

    protected const int maxWeapons = 1;
    private static readonly int shootingDistance = 10.0f;

    private event MovementDelegate onStartFalling;
    private event MovementDelegate onEndFalling;

    private int health = 100;
    public int Health
    {
        get => health;
        set
        {
            int damageTaken = health - value;
            health = value;
            if(damageTaken > 0)
            {
                OnDamageTaken?.Invoke(damageTaken);
            }
        }
```

```
        }

        [SerializeField] private Weapon weapon = null;
        [SerializeField] private float maxWalkSpeed = 5.0f;

        private float currentWalkSpeed = 0.0f;
        private MovementState currentMovementState =
                                                MovementState.Walking;

        private void Awake () { }
        private void Start() { }

        public void Kill() { }

        private void UpdateAnimationState() { }
    }
```

Incorrect:

```
    public class Player
    {
        [SerializeField] private Weapon weapon = null;
        [SerializeField] private float maxWalkSpeed = 5.0f;

        public delegate void MovementDelegate(MovementState);

        public event Action<int> OnDamageTaken;
        public event MovementDelegate OnMovementStateChanged;

        protected const int maxWeapons = 1;
        private static readonly int shootingDistance = 10.0f;

        private event MovementDelegate onStartFalling;

        private int health = 100;
        public int Health
        {
            get => health;
            set
            {
                int damageTaken = health - value;
```

```
                        health = value;
                        if(damageTaken > 0)
                        {
                                OnDamageTaken?.Invoke(damageTaken);
                        }
                }
        }

        private event MovementDelegate onEndFalling;

        private float currentWalkSpeed = 0.0f;
        private MovementState currentMovementState =
                                                MovementState.Walking;

        public enum MovementState
        {
                Walking,
                Falling,
                Flying,
                Swimming
        }

        public const bool ShowDebugInformation = true;
        public static readonly DebugRayColor = Color.red;

        public void Kill() { }

        private void Awake () { }
        private void Start() { }

        private void UpdateAnimationState() { }
}
```

Enums

Use singular names for enums.

Exceptions

Use plural names for enums whose values are bit fields.

An enum defines a group of values, each of which represents only a single value. A variable that holds a Direction can only ever represent one direction, which is why a singular name makes more sense than a plural one. However, if the enum has bit fields and a variable of that enum can have more than one value, such as being both the Up and Left directions at the same time, then the enum should use a plural name.

Correct:
```
public enum Direction { }
```

Incorrect:
```
public enum Directions { }
```

Correct:
```
[Flags]
public enum KeyModifiers
{
    Alt,
    Control,
    Shift
}
```

Incorrect:
```
[Flags]
public enum KeyModifier
{
    Alt,
    Control,
    Shift
```

```
}
```

Only define the explicit values of an enum if they are actually used.

Enums by default extend from int and their values start counting from 0. Defining explicit variables creates additional unnecessary work that provides no value, and may cause errors if more than one enum definitions have the same value.

Examples

Correct:

```
public enum Direction
{
        Left,
        Right,
        Up,
        Down
}
```

Incorrect:

```
public enum Direction
{
        Left = 2,
        Right = 3,
        Up = 4,
        Down = 5
}
```

Also Correct:

```
[Flags]
public enum KeyModifiers
{
        Alt = 1 << 0,
        Control = 1 << 1,
        Shift = 1 << 2
}
```

Always put the Flags attribute on enums that have bit fields as values.

The flags attribute gives access to built-in C# helper methods to deal with bit fields, such as HasFlag to check whether an enum value has a specific bit set.

Examples

Correct:

```
[Flags]
public enum KeyModifiers
{
    Alt = 1 << 0,
    Control = 1 << 1,
    Shift = 1 << 2
}
```

Incorrect:

```
public enum KeyModifiers
{
    Alt = 1 << 0,
    Control = 1 << 1,
    Shift = 1 << 2
}
```

Events and Delegates

Postfix Event subclasses from the GameLab Framework with Event.

The Event suffix makes it very easy to find event classes throughout the code base and helps you discover which events you can send through intellisense or any other IDE's code completion and parameter info system.

Examples

Correct: public class ItemPickedUpEvent : Event { }

Incorrect: public class ItemPickedUp : Event { }

Post-fix event delegates with EventHandler.

The EventHandler suffix is a well known and agreed upon convention within the C# and .NET community.

Examples

Correct: public delegate void PlayerHealthEventHandler(int health);

Incorrect: public delegate void PlayerHealthDelegate(int health);

Do NOT prefix events.

Event prefixes make the code unnecessarily long and provide no value. It is obvious from the fact that the event's name is a verb that it is an event.

Examples

Correct: public event Action<Enemy> EnemyDetected;

Incorrect: public event Action<Enemy> OnEnemyDetected;

Postfix events that happen right before an action with 'ing' and name events that happen right after an action in the past tense. Do not use 'before' or 'after' prefixes or suffixes to indicate pre and post events.

Keeping event names short and to the point makes code shorter and more readable.

Examples

Correct: public Action<Enemy> EnemyDetected;

Incorrect: public Action<Enemy> AfterEnemyDetected;

Correct: public Action DetectingEnemies;

Incorrect: public Action BeforeEnemyDetection

Keep event names as short and to the point as possible.

Keeping event names short and to the point makes code shorter and more readable.

Examples

Correct: public event Action<Enemy> EnemyDetected;

Incorrect: public event Action<Enemy> EnemyWasDetected;

Name all event method callbacks after the event they handle and prefix them with 'On'.

Exceptions

Event callback methods that are also intended to be used normally and be called directly in code should follow standard naming conventions.

Prefixing handler methods with On makes it incredibly easy to tell which methods are event handlers, and naming the methods after the event they handle instantly lets you know which event they are responsible for.

When a method is used both normally and as an event handler, naming it after the event it handles is wrong, because that gives the illusion that it is only ever called as part of an event. It also makes using it outside of the event awkward.

Examples

Correct:
```
private void OnEnemyDetected(Enemy enemy) { }
```

Incorrect:
```
private void EnemyDetectedHandler(Enemy enemy) { }
```

Correct:
```
public class PlayerUI
{
	private void Start()
	{
		EventManager.Instance.
			AddListener<PlayerDeathEvent>(UpdateUI);
		UpdateUI();
	}

	private void Update()
	{
		UpdateUI();
	}

	private void UpdateUI() { }
}
```

Incorrect:
```
public class PlayerUI
{
	private void Start()
	{
		EventManager.Instance.
			AddListener<PlayerDeathEvent>(UpdateUI);
	}
```

```
        private void UpdateUI() { }
    }
```

Use Action delegates to create events instead of EventHandler or custom delegates.

C# has a built-in delegate that handles passing in anything between no and up to four arguments. Furthermore, intellisense immediately shows the signature of Action delegates and what arguments they provide, making it easy to create a method callback with the proper parameters. On the other hand, custom delegates lack this ability, and require either relying on auto-generating a method signature, or having access to the source code and checking the delegate signature manually.

Examples

Correct:
```
public event Action<Item> ItemPurchased;
```

Incorrect:
```
public event EventHandler<ItemPurchasedEventArgs> ItemPurchased;
```

Correct:
```
public event Action<Player> PlayerDied;
```

Incorrect:
```
public delegate void PlayerDelegate(Player);
public event PlayerDelegate PlayerDied;
```

Magic Variables

Avoid magic variables such as numbers and strings. All explicit values that you use should be variables.

Exceptions

Basic math where numbers are easily identifiable, like adding 90 degrees to a rotation or adding 1 to an index, or literal strings that are only used once.

Magic variables make it a lot harder to understand what is happening in certain parts of the code, especially in more complicated pieces. Furthermore, they have the potential to cause a lot of bugs and errors due to typos or forgotten changes. Changing a speed variable in one location is much easier than updating a number in hundreds of different lines of code.

Examples

Correct:
```
int playerSpeed = 2.0f;
Vector3 motion = Vector3.forward * playerSpeed * Time.deltaTime;
```

Incorrect:
```
Vector3 motion = Vector3.forward * 2.0f * Time.deltaTime;
```

Also Correct:
```
rotationAroundUpAxis += 90.0f
```

Correct:
```
string errorMessage = “Something went wrong.”;
bool success = TryDoSomething();
if(!success)
{
        success = TryDoSomethingElse();
        if(!success)
        {
                ShowMessage(errorMessage);
        }
}
else
```

```
{
        ShowMessage(errorMessage);
}
```

Incorrect:

```
bool success = TryDoSomething();
if(!success)
{
        success = TryDoSomethingElse();
        if(!success)
        {
                ShowMessage("Something went wrong.");
        }
}
else
{
        ShowMessage("Something went wrong.");
}
```

Incrementing and Decrementing

Always pre-increment and pre-decrement variables.

Exceptions

The value of the variable before the increment or decrement operation is required.

Pre-increments and decrements simply modify the value of the variable. On the other hand, post-increments and decrements first store the current variable value in a temporary variable, return that to the caller, then modify the value of the original variable, and then delete the temporary one. Therefore, pre-increments and decrements have much less overhead, and improve performance and reduce garbage.

Examples

Correct: ++i;

Incorrect: i++;

Correct: --index;

Incorrect: index--;

Correct: int previousIndex = index++;

Incorrect: int newIndex = index++;

String Building

Use the new C# 6 $ operator to create strings with variables inside of them instead of the + operator instantly or StringBuilder to concatenate strings over a period of time.

Using the + operator to concatenate strings with other strings or variables creates a new string at every location the operator is used. "hello " + username + ". Welcome!" creates two new literal string objects - "hello " and ". Welcome!", but also, if the username is Alex for example, "hello Alex". The more string addition operations happen, the more garbage is generated. On the other hand, the $ operator takes all the variables you provide and formats the string directly without creating any extra garbage, so $"hello {username}. Welcome!" would only generate one string object. Additionally, when you use the StringBuilder to build strings from multiple parts, it does not create any additional string objects, but instead, appends the data you provide it directly to the already existing data in memory, without any temporary string objects.

Examples

Correct:
```
string welcomeMessage = $"Welcome {player.Name}!";
```

Incorrect:
```
string welcomeMessage = "Welcome" + player.Name + "!";
```

Correct:
```
string players = string.Empty;
StringBuilder stringBuilder = new StringBuilder();
for(int i = 0; i < players.Count; ++i)
{
        Player player = players[i];
        stringBuilder.Append($"Player {i} - ");
        stringBuilder.AppendLine(player.Name);
}
players = stringBuilder.ToString();
```

Incorrect:

```
string players = string.Empty;
for(int i = 0; i < players.Count; ++i)
{
        Player player = players[i];
        players += “Player “ + i + “ - “ + player.Name + “\n”;
}
```

String Comparisons

Use the == operator to compare two strings together.

Programmers who come from a Java background may be intimidated by the == operator for strings, because in Java that operator only compared references. However, in C#, this operator is overloaded to actually compare the contents of the strings as well. Therefore, in C#, there is no need to clutter the code with many calls to Compare or Equals methods.

Examples

Correct:
```
bool isPasswordCorrect = password == “Password”;
```

Incorrect:
```
bool isPasswordCorrect = password.Equals(“Password”);
```

Use the string static class built in comparison methods to check when a string is null, empty or only white-space.

The C# string class has a few utility functions that make code more readable and boolean statements shorter when working with strings. Oftentimes, you need to check whether a string you have is null or is empty. Instead of writing an or statement that makes sure that these two conditions pass or fail, C# already provides this functionality for you through the string.IsNullOrEmpty(string) method.

Examples

Correct:
```
string name = string.Empty;
if(string.IsNullOrEmpty(name)
{
        return;
}
```

Incorrect:
```
string name = string.Empty;
if(name == string.Empty || name == null)
{
        return;
}
```

Early Returns

Use early returns instead of nested statements when possible.

Early returns help eliminate a lot of indentations, make it easier to debug and breakpoint, and reason about functionality. Furthermore, the reduction of indentations makes the code less complicated and helps eliminate duplicate code in certain cases.

Examples

Correct:

```
public void UpgradePlacedItem()
{
        if (!IsInPlacementConfirmationMode)
        {
                return;
        }

        WorldItemScript upgradedItem =
                        itemPlacementInformation.WorldItem.Upgrade();

        if (upgradedItem == itemPlacementInformation.WorldItem)
        {
                return;
        }

        itemPlacementInformation =
                                room.PlaceItemDelayed(upgradedItem);
}
```

Incorrect:

```
public void UpgradePlacedItem()
{
        if (IsInPlacementConfirmationMode)
        {
                WorldItemScript upgradedItem =
                        itemPlacementInformation.WorldItem.Upgrade();
```

```
                    if (upgradedItem !=
                    itemPlacementInformation.WorldItem)
                    {
                        itemPlacementInformation =
                            room.PlaceItemDelayed(upgradedItem);
                    }
                }
            }
```

Correct:

```
for(int i = 0; i < 10; ++i)
{
    if(i % 2 != 0)
    {
        continue;
    }

    Debug.Log($"{i} is even!");
}
```

Incorrect:

```
for(int i = 0; i < 10; ++i)
{
    if(i % 2== 0)
    {
        Debug.Log($"{i} is even!");
    }
}
```

Formatting

Post-fix explicit floating point values with an f. If the value does not have a decimal point, add one, followed by a zero.

The 'f' suffix lets developers instantly see that they are dealing with floating point numbers, as opposed to integers and doubles. This avoids implicit conversions from integers to floats and also prevents potential integer division errors when you intended to divide an int by a float, but instead divided an int by an int.

Examples

Correct: float number = 2.0f;

Incorrect: float number = 2;

Group variables of related functionality and separate the groups with empty lines.

Grouping variables of related functionality makes it much easier to skim through the code and identify and find certain variables.

Examples

Correct:
```
private float maxWalkSpeed = 200.0f;
private float currentWalkSpeed = 0.0f;

private string name = string.Empty;
```

Incorrect:
```
private float maxWalkSpeed = 200.0f;
private string name = string.Empty;
private float currentWalkSpeed = 0.0f;
```

Insert an empty line after closing curly brackets and similar groups of logic and/or functionality.

Having every curly bracket on its own line makes it easy to visually identify where a scope starts and where it ends.

Examples

Correct:

```
if(tile.IsEmpty)
{
}

if(player.IsAlive)
{
}
```

Incorrect:

```
if(tile.IsEmpty)
{
}
if(player.IsAlive)
{
}
```

Insert an empty line between the last declared member variable and the first declared method.

The empty line helps make a visual distinction of where member declarations end and functionality begins.

Examples

Correct:

```
public bool IsAlive { get; private set; }

public Player() { }
```

Incorrect:
```
public bool IsAlive { get; private set; }
public Player() { }
```

Use tabs instead of spaces to indent code.

The tab character is meant for indentation. Furthermore, it separates the actual code from how it looks. A tab character will always remain as a tab character no matter the environment or the software. The settings of how many spaces a tab represents may be different, but that only affects visuals, without modifying the code.

Examples

Correct:
```
while(hasWavesLeft)
{
[tab] SpawnWave();
}
```

Incorrect:
```
while(hasWavesLeft)
{
[space] SpawnWave();
}
```

Only declare one variable per line.

Declaring multiple variables in a single line could cause confusion about the types of variables and their initial values. This is also a well agreed upon convention with many different programming languages and communities.

Examples

Correct:
```
int number1;
int number2;
```

Incorrect:
```
int number1, number2;
```

Declare variables in methods and functions right before they are used.

Declaring a variable as close to where it is used as possible eliminates the possibility of that variable being modified unexpectedly before it is needed, preventing issues, bugs and headaches.

Examples

Correct:
```
float rotation = 47.5f;
CachedTransform.Rotate(Vector3.up, rotation);
```

Incorrect:
```
float rotation = 47.5f;
UpdateUI();
UpdateVelocity();
CachedTransform.Rotate(Vector3.up, rotation);
```

Do not create comments. Instead, write code in a self-documenting and understandable way.

Exceptions

Comments that document complicated logic or math that cannot be simplified.

While short comments that describe the logic and the intent behind a line of code are great for when a certain piece of code is complicated, many comments clutter the code, provide no value to the functionality of the program and only make it harder to read and focus on the actual program. Even in complicated pieces of code, you should always try to refactor and simplify it, instead of having comments all over the place.

Examples

Correct:
```
public int DialogID { get; private set; }
```

Incorrect:
```
// Dialog ID
public int DialogID { get; private set; }
```

Insert a space after the comment symbol in normal comments.

Space after the comment symbol looks nicer and makes it possible to quickly double click the comment symbol and replace it without selecting whatever text is immediately attached to it.

Examples

Correct: // This is a comment

Incorrect: //This is a comment

Avoid #region wrappers. Let your code be seen!

Regions can contribute to creating code smells, such as long methods, which will potentially increase the number of bugs. These require more work which does not increase the quality or the readability of the code, does not reduce the number of bugs and only makes the code more complicated to refactor. Regions also make it harder and longer to look and skim through source code and find specific parts of the program. Furthermore, regions can hide important part of the code, such as an early return, making you not realize why your code is not working, and wasting hours of work time.

Examples

Correct: float number = 2.0f;

Incorrect:
```
#region variables
float number = 2;
#endregion
```

Paths and File Info

Use the C# static Path class to create and work with file paths.

The Path class provides a lot of helpful utility methods that make dealing with paths much easier and hassle free, especially when working with different platforms. This class can retrieve file names, extensions and much more from just a path, and it does not care about which type of directory separator you use.

Examples

Correct:
```
string modelsPath =
        Path.Combine(Application.dataPath, “Prefabs”,
        “Models”);
```

Incorrect:
```
string modelsPath = Application.dataPath + “Prefabs/Models”;
```

Create FileInfo and/or DirectoryInfo objects instead of working with the static File and/or Directory classes.

Exceptions

You only need to perform one operation on a directory or a file.

The File and Directory classes do not hold a state, and therefore, require many more parameters when working with. This makes the code more complicated and harder to read. Furthermore, every single operation with the static classes incurs an overhead due to security checks. On the other hand, the FileInfo and DirectoryInfo classes only incur a performance hit when being instantiated, but they do keep a state and are much nicer and easier to use. Do note, however, that these need to be explicitly Refreshed to retrieve new information from the disk. But, even still, they allow for performing multiple operations in bulk before needing to refresh, unlike the static classes.

Examples

Correct:
```
FileInfo fileInfo = new FileInfo(filePath);
fileInfo.Exists;
fileInfo.Directory;
...
```

Incorrect:

```
File.Exists(filePath);
Directory.Create(Path.GetDirectoryName(filePath));
...
```

ScriptableObjects

Scriptable Objects that act as a data template for a MonoBehaviour should be named after them and suffixed with Template.

Exception

The ScriptableObject is a stand-alone asset that can exist on its own.

When you want to make a MonoBehaviour and split its data and some helper functionality into a separate ScriptableObject asset that designers can create, it is not obvious what to call the MonoBehaviour and what to call the ScriptableObject. Ideally, they would both be called the same way, as the ScriptableObject represents the MonoBehaviour's data, and the MonoBehaviour handles the functionality. However, two classes with the same name would be really confusing to navigate through the project. Therefore, the front of the game, which are the MonoBehaviours, retain the original intended name, and the ScriptableObjects, which represent the data of these MonoBehaviours, get the MonoBehaviours name plus a Template suffix.

Example

Correct: public class ItemTemplate : ScriptableObject { }

Incorrect: public class Item : ScriptableObject { }

Scriptable Objects that have their data changed at runtime should extend from RuntimeScriptableObject in the GameLab Framework and use the static CreateInstanceFromAsset<T> function.

ScriptableObjects represent assets in the project. Therefore, any changes to the original asset reference in code will be saved in the project and cannot be undone. This is the same as using the sharedMaterial property in the Renderer class. It is okay to use the

original asset to read data from, but once the asset's data needs to be modified, it needs to be cloned, which is what the CreateInstanceFromAsset<T> function from the RuntimeScriptableObject class does. Furthermore, this function saves a reference to the original asset and lets you compare different instances of cloned assets.

Example

Correct:
```
public class Quest : RuntimeScriptableObject { }
Quest questInstance = Quest.CreateInstanceFromAsset(questAsset);
```

Incorrect:
```
public class Quest : ScriptableObject { }
```

MonoBehaviours

All MonoBehaviours should extend from BetterMonoBehaviour in the GameLab Framework.

The BetterMonoBehaviour provides utility properties and functionality. First, it caches the transform property so that there are no performance costs as opposed to using the built-in Unity transform property. Second, it provides access to other cached properties, such as a CachedRectTransform when working with UI, eliminating the need to do casting, or CachedBounds that are recalculated automatically for you when the object's transform changes., allowing you to snap objects to other objects, make quick collision checks, etc.

Examples

Correct: public class Player : BetterMonoBehaviour { }

Incorrect: public class Player : MonoBehaviour { }

Use the BetterMonoBehaviour CachedTransform property instead of the MonoBehaviour transform property (and any other cached properties instead of the default ones).

The transform property in Unity does not actually cached the transform component in the managed C# environment. Instead, the property is cached on the C++ native side of their engine, and every call to transform requires C# to marshal into C++ and retrieve that data. While relatively fast, the overhead adds up over time and causes loss in performance.

Examples

Correct: CachedTransform.position = Vector3.zero;

Incorrect: transform.position = Vector3.zero;

Convention Template

Convention description.

Exceptions

Convention exceptions.

Reasoning behind the convention.

Examples

Correct:

Incorrect: