Feral Interactive Limited

University Toulouse III – Paul Sabatier

Internship Report

Video game porting via graphics API calls hooking

Pierre CHOLET

Internship Supervisor: Ian BULLOCK
Supervising Teacher: Mathias PAULIN

Master “Informatique Graphique et Analyse d’Images”
(Graphics Computing and Image Processing)

From March the 4th to September the 6th
Year 2018 – 2019

Table of Contents

Acknowledgement

I would like to solemnly thank Mr. David STEPHEN, managing director of Feral Interactive Ltd, for the opportunity and for his caring.

Equally, I would like to respectfully thank Mr. Ian BULLOCK, head of technology, for his precious help and support from the application process until the end of the placement, for his caring and understanding, for the opportunity I was given, for taking our thoughts and feelings into account while guiding us regarding projects.

I would like to thank Ms. Stephanie HORROCKS and Mr. Alexander SIMPSON for walking me through the application process, and for their help and support during the placement.

I am grateful to Mr. Jason WYATT and Mr. Finn BRYANT for being my consecutive mentors throughout projects as experienced lead developers.

I would like to thank Mr. Tom MASSEY, for his dedication to properly manage projects and his patience over my confusions.

I am grateful to Mr. Ahsan F., Mr. Ben L., Mr. Christos L., Mr. Edward S., Mr. Luke D., Mr. Matteo B., Mr. Stan I.W., Mr. Stavros B., Mr. Thierry F., Mr. Zain R. for the help, the good mood during work, the enriching talks and the friendships.

In like manner, I'm also grateful to all Feral staff because it's an amazingly positive and caring place to work at, full of brilliant people.

I would like to express my friendly thanks to Ms. Harley G. because we all need a sysadmin friend that has your back like she does.

I would like to thank Prof. Mathias PAULIN, head of the STORM team from IRIT laboratory, for being my tutor teacher.

Likewise, I would like to thank Prof. David VANDERHAEGHE and Prof. Adrian BASARAB, heads of the IGAI master's degree, for supervising the internships and allowing me this thrilling opportunity.

Finally, I can't thank enough all the professors from IRIT (Toulouse Paul Sabatier University's Computer Science Laboratory), and especially the STORM and MINDS teams but also the external lecturers for the outstanding knowledge they taught us during this high paced year.

Abstract

There are just a few video gaming solutions on small market share platforms. Native support being quite infrequent, players either have to hope for **ports** or rely on **compatibility layers**. **Feral Interactive Limited** is a company dedicated to video game porting. The latter denotes a conversion work for games to yet untargeted platforms such as **macOS** or **Linux** systems in our case but also mobile or console devices.

Strengthened by an optimised suite of **compatibility layer libraries**, **C++ cross platform development** at Feral involves platform specific debugging and graphics troubleshooting but also logic and UI overhaul to bring recent popular "AAA" rated games to new platforms, redefining their attractiveness.

Prior to a **DiRT4 port** release on desktop platforms, the project underwent a **bug fixing** campaign to match the original experience along with high quality and performance standards.

A yet to be announced **project** was also worked on for several months to improve and implement the envisioned features for its porting including **heavy refactoring**, **optimisation** and a **main menu rework**.

Introduction

DirectX is Microsoft video game development library accounting for graphics, rendering, audio, input devices, networking and more, which is pretty much an exhaustive feature list game engines need to make a video game. First released on 1995 for Windows, there is a quasi-exclusivity on using this library because practicality and momentum. Alternative platforms such as macOS (formerly Mac OS X) and Linux were sort of excluded from the video game market ever since, as DirectX remained a Windows specific product. Yet, an open industrial consortium founded in 2000, the Khronos Group, proposed a set of open specifications to achieve interoperability and give all platforms a fair chance to acquire a balanced market share. Among those specifications was OpenGL, which is a graphics API specification aiming to be a serious open alternative to Direct3D, Microsoft's own graphics API powering most of the released desktop games on Windows. Only now are we seeing some change on the video game production ecosystem and an eventual threat to the monopolistic character Microsoft established. Thanks to the joined efforts of many developers across the world, macOS and Linux systems are now eligible platforms to play many recent games. The main reason is tools like Wine and DXVK, these are compatibility layers to translate at runtime the Windows and Direct3D system calls, in other words, it allows to run Windows games and applications on other platforms with decent performance. This is a major gaming changing statement as macOS and Linux user bases can now prove they are not an unprofitable minority and that game development can aim alternative platforms for a profit rather than a cost.

In the meantime, there is a lot of momentum, game engines slowly make the effort to support more open graphics API such as Vulkan, while at the same time Apple drops OpenGL support and leave Vulkan behind in favour of their own graphics API, Metal. Because game development usually take up to several years, we can not expect a such recent ecosystem change to be immediately reflected in the widely distributed and used technologies. Fortunately there are some companies that are dedicated to cope up with this situation. Feral Interactive Limited, among others, propose to publishers to port their games to platforms where their games remains unavailable because the lack of interest or profit the platforms represent. They are an interface between native platform support, yet quite rare, and compatibility layers which are unstable and under-optimised by providing a high performance compiled graphics API compatibility layer with specific native system calls.

As part of the Master's degree "Informatique Graphique et Analyse d'Images" (computer graphics and image processing) of the french University, Toulouse 3 – Paul Sabatier, I was recruited as a placement student at Feral Interactive Limited as a cross platform C++ developer. I was involved in the process of porting two Windows games to both macOS and Linux. The first game was DiRT4, a rally racing game which was

released by the end of March and the second shall remain unveiled until the official announcement according to the Non-Disclosure Agreement. To be more specific, I worked collaboratively with other game developers to improve the original game code base by fixing bugs of various natures and adding features while trying to preserve the best we can, the original desktop experience.

This internship report will first introduce the company history, activity, teams and functioning to replace the placement context and stakes. Moreover, it will describe the first steps of a newcomer video game developer to briefly explain the work environment in a second time. Furthermore it will thoroughly relate the assigned tasks throughout the whole placement and their technical entanglement. Finally it will summarise the placement experience and mention a personal hindsight in an ultimate conclusion.

Feral Interactive Limited

History

Feral was founded on 1996. It is located in Southfields, borough of Wandsworth, in South-East London. Its Managing Director Mr. Stephen was originally working in other media production and seized the opportunity of the video game market on Apple platforms. The company was indeed specialised in products aimed at Apple platforms and remains a heavy tradition in the company. It later opened to mobile markets by supporting Apple's iOS, then Google's Android systems, then shortly after Linux-based operating systems. Due the broadly opened architecture of GNU/Linux systems, they're officially supporting one of the most famously known and used in the name of Ubuntu, while developing on Fedora. Recently they even widened their horizons to console platforms such as the Nintendo Switch and streaming service Google Stadia.

To account for all these platforms and the growing number of simultaneous projects, the company scaled over the years, while there was around half a dozen developers around four years prior, there are now around thirty. As such, a new office was recently opened, Feral North, in which are the Art, Design, Localisation, Marketing and Public Relations teams. The main office progressively became dedicated to development and QA. The total number of employee is around 60, and as a matter of facts, it actually is a relatively small-sized company within the gaming industry. Indeed, slightly over the size of an independent video game studio development, while large companies releasing high quality games labelled as "triple A" usually employs hundreds of people, sometimes even up to thousands. "Triple A" games are labelled as such by investors as while they cost a lot of money to produce (ranging from

hundreds of thousands to several millions dollars), they also are expected to provide a fair return on investments usually in millions of sales.

Feral Interactive Limited business model resides in approaching publishers interested in broadening their products market to yet uncharted markets and agree on porting the games from the current game state source code to uncovered platforms. Those markets usually represents a quite lesser market share. As a consequence, the port is usually proposed late as a product life extension when most profit has already been made on usual platforms. The publishers usually hand out the source code as is, along the development, marketing plan and publishing exclusively to Feral. Though they are required to provide an ultimate approval on the product quality and artistic direction for any changes made. The game porting philosophy here at Feral is to keep the game as close as the original experience but sometimes some heavy adjustments might be mandatory (for example porting a console game to desktop platforms, or a desktop game to mobile devices).

Video Game Release Cycle

I will briefly go through the usual release cycle of a game port to better explain the context in which my placement took place. Firstly after the games are agreed on by publishers and here at Feral, they are bound by contract to release a game port with the publishers conditions. The original code base is then handed to Feral, and the mandatory step is securing the data on encrypted drives and synchronising them on the repository.

Once done, the actual development can start by setting up a project. Using a project template, which already provides an execution entry point for the Feral in-house launcher, it is required do the configuration process by specifying and setting up the dependencies and hooking the the game launch event with the original code entry point on a multi threaded environment. This is usually handled by senior developers as it requires the in-house library experience while being confronted to an uncharted legacy code base.

Once the code is hooked up to the targeted platforms build mechanisms such as xcode projects for macOS and iOS, CMake on Linux or Android or Switch, or even Visual Studio solutions for Windows, the first step is to successfully compile the game on each of these platforms. Of course there will be a tremendous amount of errors and warnings because of the inconsistencies between the usually used Microsoft C++ (MSVC) compiler toolset and both clang and gcc compilers, respectively used on macOS and Linux. There's no magic to it, it's a lot of rigorous and long efforts to fix them one by one and replace the unavailable dependencies. Hopefully the in-house libraries are now well furnished to provide most low-level I/O operations, memory

management, graphics API and hardware drivers compatibility required but some further features or library implementation can sometimes persists.

Once the game runs, it's usually in a rather unstable and bad looking state and now starts the same process again but this time about the runtime issues such as crashes involving memory management, rendering issues and other inconsistencies. This is the core of the work here at Feral.

A game port usually takes up to 4 to 6 months but can take more if the ports involve new designs, features as it's often the case when porting a desktop game to mobile or console platforms, this can take up to several years. I will have many opportunities to describe in-depth parts of this process in the report. By the time the game arrive to a stable state, Design and Art teams can be more closely involved if there is additional improvements or features as mentioned. Also QA (short for Quality Assurance) can start testing the game to find hard to unveil inconsistencies.

When the state of the game is satisfying, in terms of performance and graphics, a feedback is provided to the original developers and publishers in order for them to give a go on the release. Only then the game can be publicly announced. This planning allows Feral to actually be one of the rare video game studio not to perform "crunch times", known as widely criticised extensive effort periods prior release dates of products. The game is finally put on a separate branch with a frozen version of the in-house tools and libraries and starts the final race between QA and game developers to find the most bugs and inconsistencies and consequently fix them before release.

Close to release, the game is finally announced and the final marketing plan and assets are to be confirmed by the managing director. Prior to release day, several release candidates builds are generated after each consequent bug fixing to have a ready version to upload to distribution platforms such as Steam or the AppleStore on D day.

But even after release, it's far from over, as there usually are Downloadable Contents (DLC) to support, there is the customer service if anyone has trouble running the game and there will always be bugs to fix, whether found by the QA department or players. Finally some rather old projects can re-surface and benefit additional inquiries such as a 32bit to 64bit conversion, as the Mac AppleStore recently dropped support for 32 bits applications, or even benefit from a major graphics API improvement, for example lately replacing OpenGL for Vulkan.

Context of the placement

The French Computer Science Master's Degree "Informatique Graphique et Analyse d'Image", namely "Computer Graphics and Image Processing" involves a 6

month-long placement by the last semester closely related to the courses. Aiming at yet a new step towards either professional environment within a company or a research laboratory, the student is asked to answer a topic with a scientific process including the following steps, understanding the problem, finding appropriate academic research to address the problem, and finally propose and undertake an implementation.

This is were my personal sensibility oriented the placement course. I've long been wanting to build a career around the video game industry. After learning a broad range of disciplines at University, I finally focused on Computer Science and this precise degree allowed me this opportunity. Indeed learning the basics of 3D rendering brought me closer to game engines cores, the several graphics API. Also among the years, I cultivated a growing interest for Free and Open Source Software philosophy and became a Linux enthusiast. With this intention, I shortly noticed the exclusivity of the Windows platform regarding the desktop video game market and started monitoring the specific domain of video games porting to little market share platforms such as Linux. I finally discovered Feral Interactive Limited which did just that, and did it well. Until today I've always only seen acclaims for the quality of their ports. To back my speech I would refer to the multiple articles from Liam of the gamingonlinux website [1] [2] and the underlying comments or to Josh from the CheeseTalks website [3]. I was particularly grateful to the latter as Josh thoroughly explains the current context of the video game industry regarding the Linux platform putting in perspective native developments, ports and compatibility layers tools. These 3 aspects represent the whole range of the solutions brought to Linux and their intricate relationships. From this ascertainment, I owed to myself to be part of this and after reaching out and taking a supervised online programming test and an ultimate further interview, I was recruited by Feral as a placement student to undertake the Master's degree placement.

Goals

The initial entitlement and topic of the placement was "Video game porting via graphics API calls hook". Though, I couldn't have been aware and was actually very surprised when I first arrived at the company to witness how advanced and reliant the in-house compatibility and translations libraries already were. It successfully supported most DirectX versions already on most alternative platforms previously mentioned with all the graphics API implementations namely Vulkan, Metal and even started to deprecate OpenGL. As a matter of fact, I didn't get to work on that code base which is reserved, for several good reasons, to senior developers. The first obvious reason is the code base is more than dozen of years old and as a result is huge, very diverse and rather well optimised. A new employee would take months to catch

up with the code base and meanwhile, there are projects to undertake. Also all projects actively relies on these libraries, meaning if there's a small issue in improving a feature, it can slow down all developers across all projects.

Rather I was hired as a cross-platform C++ developer and the main role after starting on DiRT4 bug fixing, I was assigned for the duration of this placement was ensuring the Linux aspect of the main project later covered in-depth in this report. Because of Non Disclosure Agreement regarding the unannounced character of this project, I am not allowed to write down the title of this game or mention any gameplay part that would give away its identity. However I will try my best to describe the tasks I was assigned, and the processes I underwent to overcome each and every one of them. Here again the Linux translation layer was all set thanks to previous work on the title, so after a small period of setting the Linux build back on track, I actively took part in the game development among other experienced developers. In other words my actual work was to analyse, understand, improve or even add features while troubleshooting and fixing bugs in order for the game port to meet high quality standards and be released on time.

Stakes

In consequence, this report follow a rather unconventional form, as a precise technical topic is usually expected to be introduced, explained and solved with extensive bibliography and academic papers to back it up. Because I wasn't just dedicated to a specific technical approach to answer a need for a company product spread over the 6 month period probably like most of my classmates, I have been part of this video game industry release cycle which really felt centered around the project. However, because of the relentless and contingency-prone essence of game development, this main project management evolved during the internship. Indeed around the middle of the placement, the remaining work was re-evaluated to match the expected deadlines and as a result, a lot of developers were redirected from their original projects to this one, at least tripling the workforce dedicated to this project. It became a race against time to do great quality work and prioritize the efforts to get the best out of this project while willingly leaving some non prioritized minor and inconsequential issues behind.

Regardless, this was my chance to prove my skills by adapting quickly to any context or situation, to any platform development and to any task the projects would require, be it graphics, User Interface, low level system resources management, networking, audio libraries, or even actual gameplay logic. Besides, it was also my chance to gather as much knowledge on topics I'm passionate about and I will keep improving during my career by having the fortunate opportunity to work and collaborate with experienced and brilliant people.

Being a cross platform game developer starter

This part describes the first mandatory tasks and basic knowledge at the company, in order to introduce to the reader, the work conditions and the workload organisation and undertaking. This will allow me to focus on the technical part within the next sections.
Upon my arrival into the company, I had to catch up with a sufficient section of the knowledge base regarding technologies, tools, systems, facilities and even knowledge itself. I also had to set up my workstation. Due to several arrivals during the same time span and eventually some logistic hazards, I lacked the expected hardware for my main objective which was Linux development. However, following the company philosophy and tradition, everyone is expected to dedicate some of its work time to development on Apple platforms, but usually focused around macOS. Besides, working on cross-platform projects allowed me to undertake the required generic and non platform specific tasks regardless of the system I used. Thus I could use a spare iMac workstation.

Setting the workstation took almost the entire first day to get the development environment up and running. Here are the mandatory tasks :

- Encrypting the drive meant for the game source
- Cloning the source repository
- Retrieving the game and shared data from yet another repository
- Setting up environment variables to take the paths into account
- Configuring Apple XCode preferences and source trees
- Setting up credentials and developer IDs and keys
- Catching up with the internal coding guidelines

In parallel of theses tasks, given some of which are taking a long time, I could undertake the administrative and formal tasks as an employee of Feral Interactive.

I first needed to be acquainted to the toolchain to operate game development. The backbone of this toolchain is the Distributed Version Control “Subversion”. It is currently used to checkout and commit every aspects of the production. From the codebase to the all aspects of production including art assets and internal matters. Using a version control system has to advantage to make developers double check the code they commit to make sure they won’t break anything on anyone’s end, and when something breaks there’s usually a quick reaction to sort it out. To parse and have a convenient visual representation of code changes, there is web service instance of view the repositories. Of course, constant discussions are taking place to plan for a better, more productive and more reliant toolchain but that’s usually heavy processes that might hinder game development. Everything related to commits is usually notified by mails, and there can be hundreds of commits per day hence the advantage of email filters.

To ensure the projects and libraries are functional and stable there's an integration chain. It notifies about the status of all platforms states. Integration at Feral is a Jenkins supervised constantly building mechanism that checks out latest commit and try to compile most of the active code base (recent projects, dependencies, libraries). This has the advantage of immediately showing via a web interface providing useful information displayed on monitors across the office whenever there is a problem on a platform and make sure there is a quick reaction to compromise anyone's work as less as possible.

Next to integration is the build and deploy mechanism. Local builds are what developers use on a daily basis but are unpractical for any other use. Artists, designers, managers and even QA need stable builds with release notes to be aware of the state of a project. In addition, to ship a game there need to be an exhaustive flawless build mechanism to export consumer release ready builds ready to upload to distribution platforms. Sometimes there can be intricate additional processes given the platform. This build system interface allows to queue builds based on specific rules and settings for fully parametrised builds.

Once builds are deployed on platforms, for example application bundles or development branches on distribution platforms, QA submit feedback on a complete bug tracking and project management suite in the name of Jira, developed by Atlassian. Jira is organised by projects and is helpful to categorise bug or inquiries by priority, category or even department. It is crucial for the QA team to provide the precise context in which they discovered a bug and the steps to reproduce with the build number, and for game developers to notify in which version number will the bug be fixed. This is clearly an efficient interface between the two departments.

Finally to sum up, keep track and organise the internal knowledge of all aspects of the production cycle, we have at our disposal a wiki-like platform, Confluence, also from Atlassian.

I only introduced here the critical sections of the daily workflow, we also make efficient use of dozens of other services to keep track of games details such as mainstream information, localisation, crashes or company functioning related services. Nevertheless I won't mention them here as they mostly constitute internal private matters irrelevant to the placement topic.

Finally the last step to get started and be actively involved is to attend the several meetings, decisive for the work we have to undergo. The first kind of meeting is 1:1 developer meetings. All developers in the company are meant to see the Head of Technology on a regular basis. Here are discussed the feels and eventual concerns related to the work position or the projects themselves. The second kind of meetings are exclusively projects-related, they're mandatory to effectively make progress. Indeed, the office is usually a quiet place so this is the best occasion for everyone to

share the advancement they've made and here again, the eventual concerns they might have on certain tasks or features. Given the timeline of the projects, they can also serve to track the everyone's schedules and make adjustments if need be, to efficiently prevent most delays and keep the release date as is. Those are held weekly as they're dictating the projects pace.

With all this work environment related information, we are clear to get into the the exhaustive placement tasks following a weekly scheme. As such, the Appendix I – Placement Schedule shows the timeline of events and huge tasks held during the placement. Yet it should be kept in mind that this weekly scheme is indicative as tasks can obviously not be perfectly timed to precisely take place within a given week.

DiRT4

Dirt 4 is a "rally-themed racing video game" initially developed and published by Codemasters on the Microsoft Windows and Sony Playstation 4 and Microsoft Xbox One platforms by the beginning of June 2017. It is developed on the Ego engine, a proprietary in-house game engine from Codemasters which is a modified version of the former Neon game engine. Feral Interactive was in charge of the port to Apple macOS system and Linux distributions, officially only supporting Ubuntu 18.04.

The choice to only support Ubuntu is the result of several reasons. Firstly, Ubuntu is among one of the most used and widespread Linux distribution out there. Then, given the fact Linux is open to a broad variety of softwares and libraries stacks for any given system, there is also loads of different systems. Finally to optimise ecosystem coherence, indeed the online marketplace and video game management platform Steam, on which is usually released the Feral Interactive ports, is also only officially supporting and maintaining their software specifically for Ubuntu (not to mention their own [Steam]OS based on such distribution).

Week 1 (March)

My first assignments were bug fixes aiming at the public release by the end of the month.

Bug 1 was about missing settings in preference file. Feral Interactive has a XML format preference file exposed to other users to keep track of additional settings or game states that the original game lacked. Changing the "Aspect Ratio" and "Anisotropic" filtering settings in the game wasn't reflected into this file. For a first assignment the fix was seemingly trivial. The settings were already being implemented in the game under the shape of a tree data structure which is then mapped an other tree in the preference file. I only had to construct the settings value object at the end

of the settings vector written to the file. Given a unusual setting sub-category, I had to write a constructor overload to take this sub-category into account but using a tree was proven convenient enough for the fix to be straightforward.

Bug 2 related to an online menu crash and *Bug 3* to a macOS mission control use provoked crashes. I was unable to reproduce both of them and were confirmed closed by QA not able to reproduce either.

Later, I discovered an unreported crash when accessing track splines* cache. This very bug is very interesting for 2 reasons, first it's a bug that was introduced in original code and second, it is about the C++ syntax. In a range-based "for loop" statement, the "auto" keyword was used to replace the exact type held by the container. The loop was then iterating over copies of the objects directly allocated on the stack, rather than the actual objects via either a pointer or a reference. Returning a reference to such ephemeral object caused a "use of out-of-scope stack memory". The fix was actually surprisingly trivial, iterating over references of those objects by adding a '&' character to the "auto" keyword fixed the issue. Going through the rest of the file it looked like this was a benign typo at first sight, as most of the other methods did went over references instead of copies.

Bug 4 involved a texture override which was added by an other developer for generic controller button icons. Some controllers were inappropriately showing XBox controller icons and later showing the appropriate icons type but the wrong buttons. Input is proven rather complex on computers, 2 libraries co-exist at the same time, historical DInput and more recent Microsoft's XInput, and their architecture and functioning fundamentally differ. To resolve this problem, the company has set a mapping and reverse mapping mechanism in order to only use the most convenient library given the context and translates input to the other as seen fit. In this case I attempted to tweak the reverse mapping mechanism to but it was just a temporary fix for a handful of controllers. Controllers turned to be incompatible among themselves (e.g. a "3" button id would be a "4" on a different controller with the same layout).

	Saitek (P480 & P880)		Logitech RumblePad2	
Xbox reference	Library	InGame	Library	InGame
A	3	1	2	1
B	4	2	3	2
X	1	3	1	3
Y	2	4	4	4

Table I: Inconsistent mapping of several controllers

We later reverted the fix and focused on writing a controller-based override and manually edited each controller layout template so it is accordingly mapped and showing the appropriate icons. It consisted in json files listing buttons and axes ID (XInput) and their matching reverse mapped id (Dinput). I couldn't reproduce *bug 5* stating icons were steering wheels and game-pad controllers icons were mixed up.

At this point, I was also facing platform specific bugs. So it was time to setup a Mac Apple Store build, unfortunately because of my lack of knowledge about this specific platform, and despite the help of a senior developer, we couldn't manage to sign the build application correctly and run the game on this platform.

Later, on *bug 6*, QA reported some issues with steering wheel controllers, some buttons weren't recognised or wrongly bound or showing inappropriate icons, also ghost input* was experienced, that is to say when a button is pressed it keeps it's active state even after the player release, resulting in endlessly inconvenient repeating action. For this one I had to go through all input methods, keyboards, mouses, XInput wheels, XInput pad and virtual devices, to figure out how the input system worked and keep track of the relevant device state. It turns out we wanted to forward the wheels input to DInput into our libraries instead of XInput, as a lot of steering wheel controllers were designed with DInput in mind. But still some wheels were mistaken for Xbox 360 controllers. I had to override the XML list of supported devices making sure not to change anything in the original data files using the in-house technology Virtual File System which has a whole lot of purposes but it helped to override the files paths here. Supporting a wide variety of old and recent hardware, I had to add steering wheel controllers layouts using a HID calibration software and extract useful information such as vendor ID, product ID, model, and even axis ID. Some of those ID's were UUID and I could add them to the XML list of supported devices and create a new XML layout mapping for this device for the previously mentioned in-game mappings. The worst obstacle I faced during this bug fix was to deal with a special kind of model, the Ferrari GT3 which has a PS4 mode, a PS3 mode and a "hidden" hardcoded PC mode triggered when some proprietary packets are sent back and forth with the XInput library under Windows during connection on PS3 mode.

Week 2

I was then assigned to *bug 7* reported about the Feral Interactive game launcher aka "PreGame Options Window". The PGOW is based on chromium and acts like a browser using HTML and CSS. Some of the tabs transitions were significantly slower, and made the design feel uncomfortable. The reason was we had extremely detailed SVGs icons that took some time to rasterize* before being rendered. The fix was simple, I got in touch with the Art department who decided that with this level of detail we would rather use PNGs and we replaced the assets.

I will now go through *bug 8* as it is very representative of the bug fixing process in games. The context is, while on a multiplayer leaderboard screen querying friends records, one execution flow lead to a spinner icon (to notify network loading) would persistently stay on screen even after exiting this very screen.

The first step is finding the code responsible for the spinner behaviour because we usually don't have an exhaustive knowledge and understanding of the game engine. It is a methodical process even though it involves some guessing based on the classes and methods semantic. I looked for the place in code where the leaderboard screen is loaded, triggering the network logic. Once a relevant method call is found using a project search, placing breakpoints and following the execution path will eventually lead to the spinner management.

The second step after finding the spinner management calls, is understanding the functioning of this specific system. There are several spinners, one for loading settings, one for saving settings and the last is the network traffic indicator. Network being intrinsically different from basic file system I/O operations, this last spinner worked with an atomic value getting increased and decreased each time a network logic was triggered.

The third step consisted in finding out the cause of the flawed behaviour, I.e why is the spinner persistently displayed on screen ? It turns out when no friend was found, the method had an early exit statement preventing form reaching the atomic value decrease, resulting in the atomic value being out of sync with the game state.

Finally the last step and main goal is the fix process. But it's not that simple, the reason is this bug was labelled as a cross-platform issue, meaning it was also reported on Linux. The issue with cross-platform bugs is, a fix on a given platform might introduce new bugs on an other platform and the fix requires to be even more cautious than usual in changing the code behaviour because the other platforms might rely on different implementations. Here it seemed safe enough to decrement this atomic value just before early exit statements.

Due to the cross-platform nature of the bug and its fix, it involves an underlying ultimate step which is looking for regression on the other platforms. Not having the dedicated Linux hardware, this is a task I had no other choice than to pass on the QA department, explicitly mentioning an eventual regression in the fix and commit summaries. Hopefully the bug didn't introduce regression or other bugs.

To summarize the fix process, find the relevant code, build a sufficient knowledge on the specific system, find out the causes and execution flows involved and change the code based on previous steps to circumvent the flawed logic.

At this point, we were at 10 days from game release, the game and dependency libraries code had to be frozen on a dedicated branch. The development of our in-

house libraries constantly evolving, we couldn't take the risk having an in-house library introducing a regression resulting in a tremendous work load redirected to track down this regression and bisecting the libraries commits. I then learned how to setup and checkout on a branch and making sure the consecutive fixes would work on the "Master" branch as well as on the game specific branch.

Yet another PGOW related issue, *bug 9* was noticed after introducing PNGs following the fix of the previously mentioned *bug 7*. The icons now looked jagged on transitions. I had to delve a bit into front end web development but ended up finding the "image-rendering" CSS property eponymously selecting the rendering algorithm. The default behaviour had a jagged on transitions and eventually become smoother when standing still. The "optimizeQuality" property value (legal fallback synonym for "smooth") is always using bilinear interpolation as opposed to nearest-neighbour that took place during transitions. The computational trade-off was almost unnoticeable.

Week 3

Being close to game release, it was time to start building an understanding of the next project. Unfortunately this game wasn't announced yet, and because of the non-disclosure agreement I can not reveal or give out any hint that would compromise the game identity. Nonetheless I will go through the tasks I was assigned on this project with a raised level of abstraction regarding the game semantics and gameplay. I will relate to this project using the name "Main Project" as I previously mentioned, the Linux port of this game is the main topic of the placement.

Bug 1 bis reported a button not showing in the UI. I used the fixing process I just explained in previous pages. It lead me to build an understanding of the clever UI override system introduced by previous developers on this project. To cope up with the different versions which have different UI layouts, the system reads a configuration file label which is translated into a value itself decomposed into features coded in exponents of 2.

Base	Fix	Feature 1	Feature 2
0	2^0	2^1	2^2

Table II: Example of the UI feature enabling manager values.

E.g. game version A = $2^0 + 2^1 = 3$, version B = $2^0 + 2^2 = 5$, and so on.

Each version using a different game data override folder, I noticed, trying different value, that the assets number mismatched between several versions. I could then make the assets inventory for all folders and ultimately find the relevant asset folder.

By the 20th of March, my Linux hardware arrived. My tasks were re-prioritized on setting up the new workstation so I can work as soon as possible on Linux specific bugs on DiRT4. This took me the rest of the week.

The system administrators department had setup the several OS's (Ubuntu, officially supported for games, Fedora and Windows) but I will focus on the official work distribution, Fedora. I faced LDAP issues to logging but this was quickly sorted out thanks to the efficient reaction of system administrators. I basically did all setup I previously described on macOS but was also significantly different because of all the system specifics. I setup the environment variable needed for the game and libraries code building process. The next step was to install all the development dependency libraries for the tools and games. Finally I installed the configured the QtCreator IDE and set up udev rules to mount and decrypt the several drives at startup upon password input at boot time.

Because I missed a convenient script in the in-house tool code base, I lacked some crucial libraries I wasted time tracking down and install them while trying to compile DiRT4 using CMake.

Week 4

I finally could start working on Linux specific bug fixing. *Bug 10* was the first task in that direction. The issue was one specific keybinding layout showed a text fallback of an icon instead of the actual icon. I tracked down the low level keyboard input to build an understanding of the situation. It turned out this specific action input wasn't mapped on a icon path and went through to the text fallback. Simply adding that specific case to the code seemed to have fixed the bug.

In reality, I had just introduced a regression *bug 11*. The previous fix did its job, it displayed an icon instead of a fallback text. Yet it showed a controller icon rather than showing the keyboard icon. The fix, this close to release was hopefully really straightforward, I didn't went through all the way to seek what icon path was used. After a close examination, it turns out the path was pointing to the the controller assets, I simply made sure the icon assets path was updated accordingly to the used hardware, here the keyboard instead of the controller.

The game was finally released on March the 28th on macOS and Linux but I wasn't involved in the release process. In the very broad lines, releasing the games obviously implies a lot of back and forth validation from the original editors through an exhaustive list of all aspects around the project, final state of the game, release date, marketing. However on the technical side, we make sure to that all critical bugs were fixed having a last QA pass and we finally use our build systems to compile debug free builds (usually called "Gold" in the game industry) for all targeted platforms then the person in charge updates the builds on their respective marketplaces. Since the release I was fully assigned and dedicated to the "Main Project".

MAIN PROJECT

I was assigned on a new version of the project for which my main placement goal was the Linux aspect. All naturally successfully compiling the project and its dependency of this system was the first task.

Because the Linux port was already planned but finally got delayed, and thanks to the efforts of the previous developers on this project, all the project libraries and graphics API override and setup was already done. In fact this required a deep understanding of the in-house libraries I lacked at the moment, and probably would have delayed the project even more for me to acquire.

The project hasn't been compiled on Linux for months so I decided to assess the game state and note down the differences and the inconsistencies put in regard to the main development platform, macOS.

The game was using an older version of DirectX which wasn't supported by the in-house latest DirectX to Vulkan translation layer. As a result, the game was running fine using the OpenGL 4 implementation but using Vulkan showed a black screen because of a backbuffer presentation issue.

The backbuffer was commonly referred as the buffer in which the image is progressively drawn and finally "presented" (swapped) in the front buffer which is rendered on screen, that method is straightforwardly called double buffering. Nowadays the modern graphics GPU uses what is called Renter Target (DirectX) or Frame Buffer Object (OpenGL) that are intermediary buffers that allows for more efficient control and even post-processing effects.

Regarding Vulkan, and using *renderdoc*, I could witness the pipeline seemingly working and the menu frame being drawn but it somehow wasn't presented on screen. Renderdoc is a cross-platform and cross-API Free and Open Source Software aiming at frame capture and analysis widely valued for GPU pipeline debugging.

I also had to fix some data paths mismatch causing crashes because of missing data, the previous inventory I did was useful but I had to delve into specifics.

Getting Started – Week 5 (April)

After sorting out the data mismatch issue I looked in the yet untracked and undiscovered Linux bugs. The first of them was a false positive, as using debugger breakpoints and abruptly pausing the game caused the audio library to misbehave. The second one was a UI path causing a crash whenever a special game character happened to pass away but turned out not to be unsual as the UI developer confirmed me this path wasn't implemented yet. I ran across several failed assertions triggering a debugger breakpoint but turned out to only be warnings and was an expected clumsy

vector normalization in the original game code. Finally some UI hardcoded array values where obsolete and I got in touch with other developers who had previously wrote this code to investigate the issue, we hopefully fixed an out of bounds error this way.

The first round of investigation turned out to be rather promising and I couldn't find any critical issue that would require a heavy workload. I went back to try and fix the code base for the Vulkan implementation. There was some issue in the swapchain implementation. The swapchain is simply the mechanism referring to swapping the two buffers (or more) in methods such a the previously mentioned double buffering. Making minor changes to the Blit method calls (Blit usually describes a texture copy to a Render Target), I managed at some point to have the game rendering upside down. This was probably caused by a different image origin between the graphics API specific implementations. I also reached a "Unimplemented" failed assertion and finally noted down missing characters table in strings, inconsistent introduction video display and unhandled pixel format in a given game mode.

During the following meeting I was told we didn't support this old version of Direct3D with our Vulkan implementation and that there was no point in further trying to get it working fine. Instead, and because that Direct3D version was very constraining, I should just wait for the new Direct3D11 version of the game engine to be more functional. The reason we choose this version is that it still followed the classic graphics pipeline architecture unlike D3D12 competing with Vulkan, also that it would allow us to have better performance while taking advantage of more control over the graphics and finally because it was probably the best supported version in our in-house translation layer toward Metal and Vulkan.

I also was moved to a new desk to be closer to the developers assigned on this project and was asked, using OpenGL for just a few more days, to look into the cross-platform issues unrelated to the graphics API.

The original version of the game featured tooltips rendered upon cursor hovering game elements (indistinctly UI and meshes). Because of an ergonomic feature, those tooltips were converted into constantly updated fixed position label strings. For the new version, this feature was irrelevant and *bug 2 bis* stated the need of bringing back the old behaviour. However there have been quite some modifications and greatly welcomed enhancements that I needed not to revert. Simple changes were done, I set changed the display location to the former cursor position and I re-enabled the background drawing. Despite re-enabling the on-screen rendering of such tooltips, it did of course not suffice. The background size was off, the tooltip would stay drawn on screen and sometimes it wouldn't even show. I turned off a previous change that would fix the tooltip position for a this specific version, I also

prevented a previous change that would make the tooltip appearing instantly after the game loaded along with the default tooltip string. Finally after some meticulous changes such as font size and colour, trailing frame size, I could match the expected former behaviour of the tooltips following the cursor upon element hover.

I then faced a rather time consuming issue coming from the building tools. Deploying a specific version of a game would overwrite some project configuration files in the main branch of the repository and would disable the PGOW because it was driven by a build rule on Linux. I should have regulated those building rules but there was so many versions I wasn't in charge of, that I couldn't risk putting a stop to the production for other developers because I didn't thought that through well enough.

Week 6

In order to help the artists, I was asked to work on a debug camera that would allow for more control than the current fixed yaw and linear pitch tied to the zoom. The D3D11 implementation came to a stable and pretty achieved state the day after so I was reassigned to making the game work using Vulkan for the D3D11 implementation. During that time span I could only find out about the height and number of tiles limitations that I hacked to try to circumvent but nothing stable and convenient enough to commit and forward to the Art team.

Using a preprocessor define and subsequent preprocessor conditional directives, we could override sections of code to use the new pipeline implementation but as previously said the in-house translation layer was already supporting that version using Vulkan very well and I didn't get to look into that specific part.

The issues preventing a fine compile were exclusively path issues. In the first place, some new class files weren't added to the CMakeLists.txt, the classic Linux building solution "CMake"'s most important file. This happened mainly because the senior graphics developer works on macOS, the latter using its own dedicated PBXProject building mechanism. Additionally, unlike macOS and Windows which are using case preserving filesystems, Linux mainly uses a case sensitive filesystem, which resulted in some data mismatch and crashes after introduction of new assets.

Header Refactor – Week 7 → Week 10 (May)

From this point on, I inherited a huge task which lasted several weeks. The main objective was to get rid of some troublesome includes. Specifically the previous versions of the port used some old in-house library classes themselves using exclusive macOS headers, given the history company, throughout the code. The consequence of this was the code held local fixes every now and then but the overall solution was relatively unstable. Until then, we were able to compile the game successfully mostly using preprocessor conditional directives but those local fixes became scattered and

increasing efforts were put just to keep these fixes coherent throughout the game code.

We finally took the decision to refactor the headers after we hit an issue including an unevaluated template method on matrices causing compile errors. In other words the game legacy libraries defined “min” and “max” methods on underlying vectors and matrices types using templates and one of those methods included a typo probably caused by a quick unthought through code copy and paste. Nevertheless the solution was compiling fine as this code being unused, it actually never was evaluated, unused C++ templates are never instantiated. To this extent, my task was to expand an interface class whose purpose was centralisation of the troublesome includes. I was given a patch witch dozens of files already updated and I carried on.

The main changes were related to two classes, our own generic class acting as a container for the game and providing utilities to interface with our libraries and the inherited underlying class adding all the logic specific to this actual game. Both contains but are not limited to, all the low level setup and settings of the application window, the sound library interfacing, the I/O operations to write and read from files with path utilities. But also the game initialization as well, given the fact the game is composed of several expansions, the launcher let you select what expansion you want to play and initialize the different data paths accordingly.

To put it another way, I moved quite a lot of logic from the game code to that interface class so I could remove the troublesome headers and having them centralized in the interface class. The process went like this, first removing the local or global fix, then looking through the compile errors to trace back which header inclusion caused it, moving the header to the interface class and adapting or moving the affected methods to the interface class, rinse and repeat.

Among all the changes, 17 headers were to be removed. Those headers were to use apple specific “CoreFoundation” and “CoreGraphics” libraries. Respectively, CoreFoundation is “framework that provides fundamental software services useful to application services, application environments, and to applications themselves” [4] according to Apple. Whereas the part we were interested in from CoreGraphics in the first place were “display hardware, low-level user input events, and the windowing system”. [5]

I faced several challenges along the way, notably while I was undertaking all those changes during that 3 weeks time span, other developers kept adding and fixing the game features introducing conflicts with my changes. I regularly checked out the new commits and spent quite some time resolving merge conflicts and compile errors. Overall I ended up fixing the changes 13 times due to a new commit made in relation to the code affected by my changes.

Further, I faced with the constraint the changes should include all supported platforms, namely desktop platform macOS, Linux and Windows and mobile platforms such as iOS and Android. But also some specific preprocessor conditional directive branches. There was very little specific change for those platforms but the issue was to access and test all those platforms in a short amount of time keeping in mind that setting up only one platform meant undertaking an heavy time consuming process. Thus I worked in coordination with others developers to better understand the other platforms I wasn't developing on and submitted them patches they tested for me before I would commit anything to the repository.

In the long run, we had a centralized place in which all the classes using platform specific headers lied and we could place the troublesome ones behind a convenient preprocessor conditional directive that prevented frameworks inclusion such as CoreFoundation and CoreGraphics. The commit was huge, hence it had to be split in several smaller commits for clarity and would also allow for chasing down regression more conveniently.

- *Feral interface class*: expansion of the interface class, preparing for whole changes
- *Feral specific source*: all the major changes being exclusively our in house additions
- *Game source*: mainly all the changes that:
  - *Libraries*
  - *UserInterface*
  - *Display*
  - *Menus*
  - *Main game logic*
  - *Remaining unspecific classes*
- *Main Project specific game class*: huge enough to be its own commit in addition to preprocessor conditional directives.

Patches files	Lines changed	Header files	Source files
Feral Interface	~ 600	1	1
Feral Source	~ 10	0	3
Game Libraries	~ 40	1	6
Game UI	~ 100	2	19
Game Display	~ 50	1	8
Game Menus	~ 100	1	6
Game main logic	~ 60	0	5
Game miscellaneous	~ 140	2	16
Game template class	~ 90	2	1

Table III – Statistics about the major header refactor commit patches

Last, because it couldn't have been that easy, yet two more issues emerged. First upon compile GCC reported multiple definitions of a custom operator. It is likely due to the operator being globally defined in several translation units. Thanks to a senior developer, the fix was easy enough, inlining the operator resulted in the operator body to be copied in each place the operator was required and disposing of the need for declaration.

The latter was a the introduction of a hang during the game resulting in unplayable state. Because I mainly moved logic trying to minimize the actual consequences, I had no clue about the cause of this hang. I engaged a manual commit bisection of my several patches to found the patch file responsible for the issue. Altogether, a conditional statement occurring multiple times in a fairly heavy loop was performing I/O operations to check a value in our game register file. We refer to those game parts which are heavy computation called at each frame, as "hot code". It's obviously a good practice to cache or hold as many fixed data as we can rather than reading them on drive. I introduced the issue in the first place using slightly different semantics without having the time to comprehend the context in which my changes took place and their consequences. Though, using the static keyword for a static local variable (rather than using its static member mechanism), inside the Interface class namespace allowed for such variable to be initialized only on the first time execution flow arrives to it, all further calls skipping declaration and using the value as we would expect in plain C. [6]

Rest of Week 10

By the time those changes were made, the senior graphics developer added the "Cascade Shadow Mapping"* method to the game renderer. Given it's workload he used convenient code that the GCC compiler complained about while clang had no issue with. "Member "" with constructor not allowed in anonymous aggregate." The issue originated from structures used as uniforms for the CSM shaders*. Only one structure was used for different purposes and holding different data via cascaded internal structures and unions.

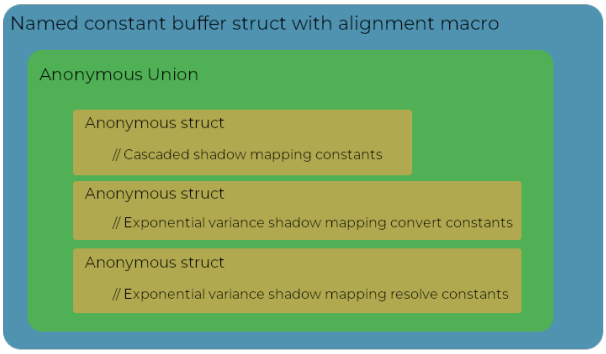


Figure 1: Structure of the constants buffer before decoupling

I was asked to decouple this structure into their own well defined structures to dispose of the unions. It was basically just declaring the most deep anonymous structures as plain named ones. All there was left to do is to appropriately fix the the structure identifier wherever they were used.

Later, because the underlying game matrices and vectors classes were non Plain Old Data (POD are c-like structures without methods or user defined constructors), GCC emitted warnings about losing packing attributes. Before I could further investigate a clean fix, I was assigned during the next meeting to chase down alarming memory leaks.

This very topic was actually quite complicated to get into. First because I wasted a long of time with flawed reports because I was using address sanitizer which drastically increase both execution time and memory usage. Address Sanitizer is a very convenient and widely used tool aiming at memory corruption bugs detection, using extra memory referred to as “shadow memory”. After disabling ASan, I used XCode’s “Leaks” and “Allocation” GUI tools on macOS and conjointly valgrind on Linux.

Besides having quite different results, I was having a tremendous amount of reports of extremely tiny memory leaks that wouldn’t have been worth to fix. I focused my efforts towards Metal graphics API calls as it seamed to leak around a hundred megabytes each consequent hour. After carefully investigating each draw call of the 13 different render passes, I couldn’t find any relevant case of memory leak. Ultimately I realized, XCode app memory usage report and its “Instruments” debugging tool “Allocation” reported different memory usage, at least it was my hypothesis after some research. “The Allocations instrument does not record OpenGL/ES or Metal texture memory. If your app allocates texture memory, your actual memory use will be higher than what Instruments reports.” [7] We concluded what we thought were memory leaks, could just have been imprecise reports from the tools and that it was probably not worth spending more time on this issue.

Yet I fixed only one minor memory leak involving the tile textures of the map in the main game mode but consequently I learned quite a bit from this experience. First about the underlying mechanism of Sanitizers such as ASan and its cost-benefit ratio. But also how to conveniently use valgrind on threaded applications for it to output a relevant and conveniently comprehensible report. And finally how to symbolize* a macOS executable by generating a dSYM file during the build process and provide it to “Instruments”. Indeed, sometimes, Intruments would report memory addresses in the call tree rather than comprehensible method names. A dSYM file is an Apple specific file storing debug symbols that can be generated upon build so that tools would be able to map memory addresses to the classes and methods names. It can also be used to troubleshoot a bare crash log. Symbols are usually excluded from

release builds because it becomes harder to reverse engineer an application and especially because it drastically reduces the binary size.

While trying to find the cause of the hypothesized huge memory leaks, I tried executing repeated behaviours of loading different modes, different assets and UI elements and I finally found some use of deallocated memory within the new UI features. I offered the developer in charge of this section to fix it for him but he gladly welcomed my report and deemed it would be faster for him to fix it himself. To sum up, the legacy game classes inherit from a base UI class with a virtual clear method which happened to be simultaneously called by UI elements and latter by their a parent UI element. The confusion probably sparked from the children actually having several extra methods including *divorce*, *unregister*, *delete*, which all have slightly different semantics and ended up a bit confusion given the huge amount of UI elements having their own behaviour.

Week 11

Prior to any further assignment I decided to look into a Linux specific water rendering issue. On the main mode map, picturing a continent, the water was flickering and seemed to have a texture corruption issue. I looked at the draw calls I previously identified to check the assets fed to the GPU but looking into troubleshooting the Share Resource Views one by one. A SRV is GPU resource descriptor used in the DirectX graphics API, holding the GPU virtual address of a Direct3D resource and describing the semantics for use in a shader. In other words it tells the shader how to read the texture we're interested in [8] [9]. I couldn't find any probing cause apart that I thought the ocean plane was rendered twice before noticing that the rivers actually used the same variables names but with completely different assets and shaders. I continued working on this issue only 2 weeks later but for the sake of clarity I will go through all the fix.

At this point I undertook a frame analysis using renderdoc. My previous use of renderdoc to troubleshoot the Vulkan implementation of the old Direct3D version was very rudimentary. This time I analysed the mesh during consequent frames. It appeared each 3 frames the ocean mesh was just rendered off-screen, letting whatever went through the background buffer be rendered on screen. Examining the vertex shader constants buffer uniform, I noticed some difference in the values between the consequent frames and especially the positionOffset.

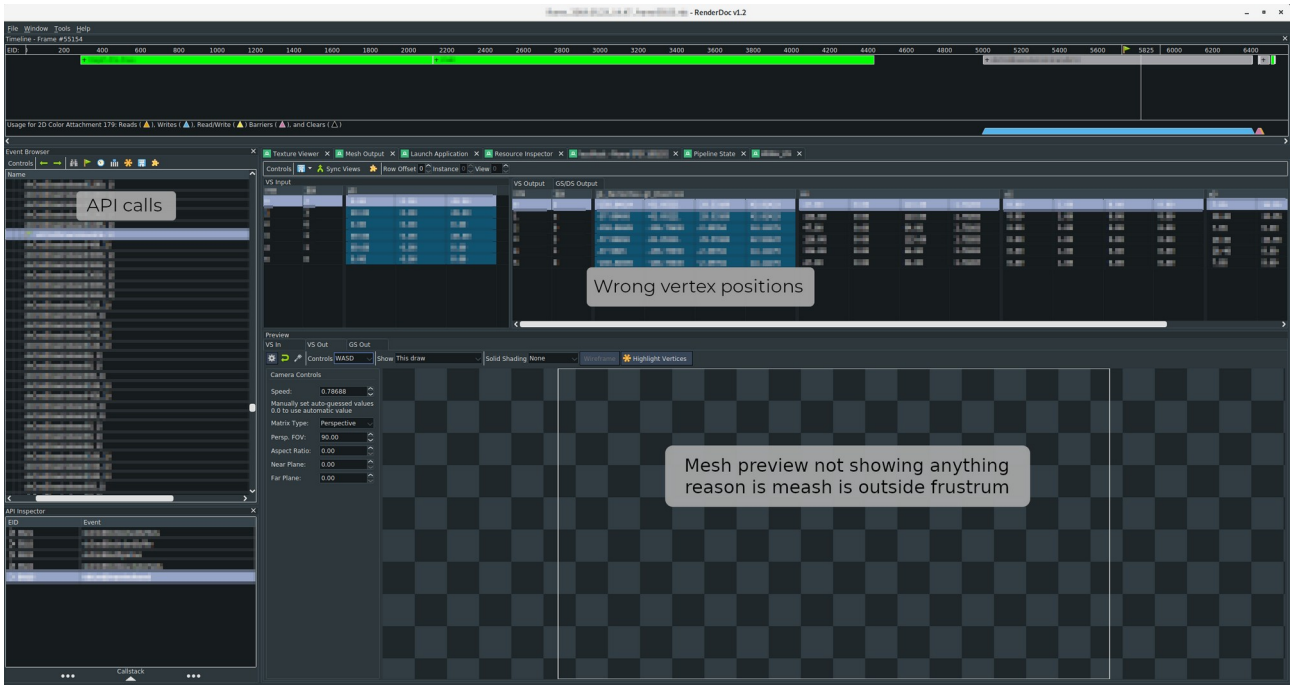


Figure 2: Redacted QRenderDoc screenshot showing specific mesh troubleshooting

The position offset is a Vec4 vector, meaning it is made of 4 components usually referred to as ‘x’, ‘y’, ‘z’ and ‘w’. Because we are very often using 4x4 matrices, using vec4 makes sense and can be interpreted as using homogeneous 3D coordinates. Though, even a slightly incorrect value in the ‘w’ component can have serious consequences because of the use of matrix operations such as matrix product. In the end, in turned out that this ‘w’ component was in fact wrong. The reason is the vec4 made out of a vec3 containing the correct positionOffset, was 0-initialized using clang but appeared to be initialized with random values with GCC. I could have change the positionOffset initialization in code but we agreed the shader shouldn’t be affected if some other position were to be incorrectly initialized. I hereby modified the HLSL shader in consequence. High Level Shading Language is the language used for the shaders in the Direct3D graphics API.

```
// position_offset is a vector4 type accessed through a constants buffer
// vertex_position is the position for the given vertex the shader is operating on
float4 final_position = float4(vertex_position, 1.0f)
final_position.xyw += position_offset.xyz;
```

Rather than:

```
final_position += position_offset; // Using randomly initialised ‘w’ messing up matrix operations
```

The HLSL shader is compiled with the DirectX Shader Compiler using a python script to automatise the process given the high numbers of shaders we have. Again all this time for just an uninitialized value but I actually learned a lot from it.

It was decided the game state was advanced enough for us to start deploy builds for artists and QA, yet we lacked the ability to control which version of Direct3D was used in those build as we did in local builds using the hardcoded preprocessor defines and consequently preprocessor conditional directives. We had to take advantage from each platform's own build system capabilities to enable this, keeping in mind the in-house automatic build and deploy system lets us use preprocessor definitions rules. Preprocessor definitions are flags that are alike environment variables that can be set for a given command. The CMake shape of this was surprisingly trivial compared to the other platforms, the in-house CMake toolchain was already covering this case it was just a matter of adding this special rule.

CMake (Linux) :

```
if(DEFINED <PREPROCESSOR DEFINE>)
        list(APPEND GCC_PREPROCESSOR_DEFINITIONS "ENV_VAR=${VALUE}")
endif()
```

XCode .xcconfig (macOS) :

```
GCC_PREPROCESSOR_DEFINITIONS = $(GCC_PREPROCESSOR_DEFINITIONS_$
(CONFIGURATION) $(ASSIGN_ENV_VAR_$(ENV_VAR)))
```

Visual Studio .props (Windows) :

```
<PreprocessorDefinitions Condition="'$(ENV_VAR)'!=''">ENV_VAR=$(ENV_VAR);%
(PreprocessorDefinitions)</PreprocessorDefinitions>
```

The following bug is probably one of the funniest and also quite indicative of the reason the Direct3D overhaul was required. For several meshes, one assert was triggered, mentioning "Unsupported format, texture creation will fail". It turns out the mesh texture was a DXGI_FORMAT_B4G4R4A4_UNORM. DXGI stands for DirectX Graphics Infrastructure, which is a framework to provide a common ground for the low level tasks of the DirectX libraries. This texture is actually "a four-component, 16-bit unsigned-normalized integer format that supports 4 bits for each channel including alpha." according to the Microsoft documentation [10].

While we can nowadays go up to 128 bits textures (32 bits per channel) rather than 16, I can't emphasize enough on how old this format is and how many mockeries it provoked when I naively asked my colleagues. Going down the stack trace, it should have seemed pretty straightforward, it was mapped to VK_FORMAT_UNDEFINED. Even though VK_FORMAT_B4G4R4A4_UNORM_PACK16 exists [11], it appeared we deliberately didn't bother supporting it and for good reasons.

The reason I only picked this issue on Linux just now is because, metal actually had conversion routines at its disposal to circumvent any unforeseen inconvenience like this. Behind the scenes, there was quite a tricky texture format management

system, with fallbacks for colour mode (which is now 32bits since quite some time anyway) or other hardware concerns. I finally made sure the mechanism wasn't introducing any undesired fallback, and just changed this texture format definition for this singular case. To be more specific when a texture was created with a call to the method finding the closest supported format, I explicitly made sure this method wouldn't return a 4 bit per channel mode by disabling a boolean set to false in its arguments.

Then there was some more crashes to look into occurring whenever we were to leave the main game mode. It happened during the parent/children deletion cascade of all the UI elements. I started fixing the range-based for loops with references, as previously done in DiRT4, we still most of the time want to use the object and not a copy. Then I tried replacing bare pointers with `std::unique_ptr` which are smart pointers that conveniently deletes the resource when they either go out of scope or are reassigned but even though it was a welcomed improvement it wasn't enough to fix the crash. After closely looking at the seemingly endless UI elements list, I discovered the UI element which actually was container visually representing a mission panel, was prematurely deallocated. My only chance was to understand why, hence how the UI element handled this specific resource and I went to check initialization where it was allocated in the first place. This is also where I figured the top-level UI containers are actually very abstract because this troublesome elements was actually added to the top level frame but also as the children to an other element of the frame. Subsequently, the deletion cascade obviously called deletion twice on the same memory address while going through the elements hierarchy. Fixing this was henceforth trivial, adding the element only to one place and updating all calls to this element to its actual ultimate place definitely fixed this issue.

Multi-threading – Week 12 → Week 14 (June)

Week 12 started with yet another missing CMakeLists.txt entries regarding the recent support of the DDS compressed texture format but also for the new High Dynamic Range skybox. High Dynamic Range or HDR, is roughly the result of having more light information than can be displayed usually in order to fake glare and enhance contrast with an extra render pass or add other fancy visual lighting effects. The skybox is a classic method to draw skies in video games, sky textures are projected on the indoor sides of a cube and this cube is rendered as background if for a given pixel there no underlying fragment (I.e portion of an object) from the scene.

Finally I faced yet another crash, fortunately it was quickly solved as coworkers mentioned they were a lot to have a registry flag enabled. Just for convenience, I initialised the troublesome pointer to nullptr, so it was not assigned a random incoherent memory address and pointer evaluation would return false instead of using a garbage pointer in conditional statements.

This next task was probably the most challenging task I faced during the internship. As the team was constantly trying to improve the game performances, and having already brought a lot of efforts in that direction, it came to the attention of the project lead, that path finding was eventually taking too much time within a frame and that we could potentially do something about it. I will further explain the path finding concept now, it refers here to all the computation happening each frame in battle mode to manage the movement of all units and underlying characters and resolve the actions and update their states. I was about to discover it includes quite a lot of details, the most obvious being finding the direction in which the unit should move to, whether it is player controlled to reach the place the player pointed at, but also the place the "computer controlled" army should go in reaction to the player's actions. For this the logic keeps track of the terrain geometry, taking into account the obstacles present in the name of trees, buildings, walls, tunnels, watery area and so on. It's however not limited to this, updating mesh animations, play according audio upon numerous events, but also resolve attacks, fire projectiles, update health, stamina, ammo values, handle characters deaths, are also involved.

As usual the first step is to investigate whether the path finding was significantly taking too much time to be processed. I used XCode Instruments' "Time Profile" tool. As previously said, Instruments is a convenient debugger and run-time analysis software with various sets of measurements, that I mentioned using to chase down memory allocation and eventual memory leaks. "Time Profile" keeps record of the total time each function call is taking and offers a tree for each thread starting with the most abstract methods such as *main*(). It also has a convenient feature to make timestamps and only consider the measurements between two timestamps. This way I could have a precise measurement of relevant execution flow of the game, discarding all the initial assets allocation. With all this in place, it remains tricky and subjective to interpret measurements. The first reason is because those measurements are based on the number of units, the density of assets a map can contain, and also relies on all individual actions and states each entity is currently undergoing. The key is to fathom the extreme edge-cases that would put the most stress on this system, assess if those cases are realistic and occurs regularly in terms of gameplay experience the original developers but also us were aiming at. It's worth mentioning this took place while the renderer for battle mode was simultaneously undergoing radical changes regarding buildings packing to save a tremendous amount of draw calls the legacy system was relying on. Finally my choice was to create battle with the most characters it was possible to involve in a battle, all clans with maximum units each with a maximum of more than 200 soldiers gave around 40 000 soldiers. It was an unrealistic battle, yet quite possible because allowed by the game. It's worth noting the game in it's original version warned the player after reaching around 6 000 soldiers

(vaguely only 2 clans with 13 units each). So the game probably wasn't made to welcome battles this huge in the first place. Anyway it still remained convenient to investigate about the timings. To hinder the renderer's implication on the measurements I chose almost empty terrain like deserts. It is basically a trade-off between we whether want to measure states updates with just few asset rendering or the actual obstacle avoidance on dense maps at the expend of having extra renderer cost. Indeed having low FPS can either be from CPU not having enough time to compute all the logic it should between 2 frames because it has to take too much into account, or it can be that there's a bottleneck on the GPU traffic, asking too many drawcalls* and/or sending huge buffers. If one is tremendously slowing down the application, it will probably hide the other factors, and that's the reason I tried to make the renderer insignificant in the measurements. I would launch the most crowded battles on deserts, giving actions to my units and then I would go as far as I could from the battle and zoom as far as I could into the ground. This way I was assured the renderer would actually do very little, being far from the battle and looking at the least amount of vertices I could, I was relying on the culling* so the graphics pipeline would discard most elements and would not try to render them. Finally the last question was about doing the measurements whether on Debug or Release builds. Release builds resulted in faster execution flow and higher FPS in game, while Debug offered convenient symbolizing of the measurements but introducing nondeterministic extra logging logic. To be sure not to have flawed results I actually measured on both. To summarise the results, I measured around 150 FPS on average with only 9600 soldiers on the terrain and consequently less than one FPS with all 38 400 soldiers under the same conditions with a Release build.

Going down several nodes with high spent time values in that tree, I found 2 methods which stood out the pack, the army update method and the arm advance method, respectively updating states and updating the position and animations. On a whole minute-long duration, the first one took 28 seconds while the latter took 24, the little remaining time was spent on graphics, input and low level routines. Regarding the first method, most time was spent on position update, unit task update (move, attack, defend and more low level state update) and finally formation within the unit. Each soldier having a dedicated place in its unit, loosing soldiers should end up in soldiers regrouping to avoid "holes" and visually respect a coherent dense units. On the other hand, the second method spent a lot of time checking the surrounding cells in a physical grid for each soldier to avoid obstacle, whether it be soldiers of the same unit or not but also terrain obstacles in the geometry or physical obstacle such as trees or buildings. Some of the most time consuming underlying methods included at some point either a lot of vector allocation, geometric calculation or even logging. Over a 10 minutes period, 48 seconds were put into VECTOR_2 operations, representing roughly 8%. Going down a bit, I figured operations such as minus or plus would allocate new

instances of this class, and it is actually the memory management which was time consuming.

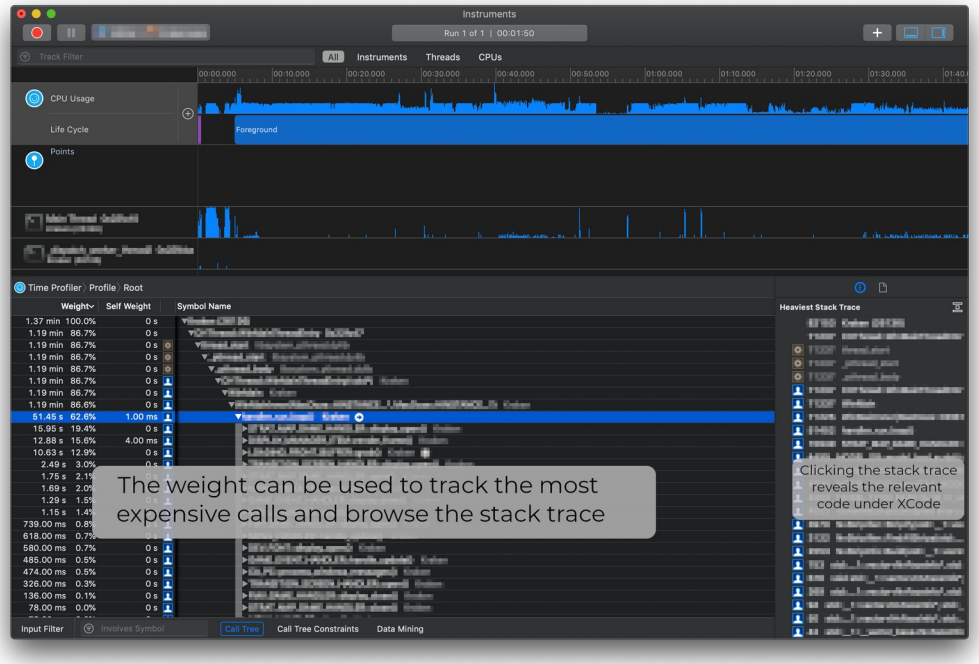


Figure 3: Redacted time profiling example using Xcode Instruments

Several solutions were considered. The first was optimisation in general, whether using efficient data structures and optimising the code logic could have enough significant breakthrough but also Link Time Optimisation. Link Time Optimisation or LTO is compiler side optimisations taking place during the linking process because the process can henceforth have a view on the whole application at this point. These optimisations can be discarding instructions with no effects (e.g useless loops that have no effect) or inlining portions of code if they are to be called often or even re-ordering routines for better memory coherence. Fortunately, enabling the LTO was just a preprocessor definition away. Anyway, there are mainly two reasons not to use LTO, the first is that the optimisations can cause very subtle bugs, and we would risk incoherent behaviour differences regarding the game states. The second is it requires an extra resource cost during compilation and end up with higher compile times which is probably to be avoided if we need to compile and test the game regularly. I tried it anyway, however, in this precise context the code base was so huge, and we added so many workarounds and quick fixes on top of that, that the result was rendering breakages and very frequent crashes. And so the LTO was out of the equation.

Next solution I had in mind was packing, more explicitly using SIMD instructions to perform more calculations at once. “Single Instruction on Multiple Data” instructions, are special processor instruction sets performing computation on

"packed data", for example multiple integers in a single registry. Despite that, the whole path finding logic would have required a full overhaul to use such parallelisation process and my knowledge of the custom data structures, the original legacy vector and matrices classes were way too intricate for me to envision such tasks. Not to mention it would also have meant supporting a very broad range of hardware and carefully study which SSE/AVX (most common instruction sets) versions to actually use. And so the packing was in turn, out of the equation.

Finally the most realistic and significant solution was dispatching calculations among multiple cores the processor is made of, in single sub-tasks called threads, also commonly referred as multi-threading. But again, this is not a light process to undertake and usually the code behaviour and structure has to be carefully thought through beforehand. Besides, here the scale of the multi-threading needed to be relevant. Indeed threading too high level abstract methods would result in threads taking a lot of time to execute their dense tasks, adding up many memory barrier to keep memory coherence along the way and taking the risk the threads would actually execute a very different amount of instructions among themselves. In contrast, threading numerous very small methods would result in taking quite some time to manage threads themselves, scheduling, starting, pausing, ending and we definitely not want to spend out time spawning threads in loops. Although the company having faced parallelism challenges before, we fortunately had a solution in our in-house libraries. A thread pool declaration and management with convenient methods to add method calls as lambdas. Lambdas are usually anonymous methods, crafted on the go to avoid the usual method declaration and definition, in C++ it has the ability to capture it's environment (variables and members) by value or reference. I was inspired by previous commits done on other projects to use this in-house thread pool class as I yet had to build knowledge about it. The most convenient place I found to declare a thread pool as a member was the unit class. Going down the nodes in the timings tree I could visualise and try to approximate the most relevant place.

1 World > 1 Battle > A dozen of armies > Dozens of units > Hundreds of soldiers

Placing the pool on the units level meant having around 160 tasks to distribute among the CPU cores. Also units are persistent classes throughout the battle so it didn't have to constantly allocate and de-allocate like some low level soldier logic, or soldiers themselves if they were to die or leave battle. I created a new preprocessor define, in order to circle my changes behind a conditional predecessor directive if those changes ever needed to be disabled or were to be thrown away. Of course the first try of parallelising the unit level advance method was a relentless failure, facing interlocks (situation in which all thread would wait one after one other and definitely stops their work) and also crashes. I am more than carefully aware that this is not the actual way of doing it, this process was very wrong. However in the time span allowed, my first goal was not to make a perfect implementation of multi-threading

but rather have a first proof of concept version to look into the feasibility but also the major challenges and obstacles to face, eventually in order to have a second more thought through clean implementation. First issue though was the logging part, which was using static classes and wasn't anywhere close to be compatible with multi-threading, I just disabled the logging as a start which made things better but not quite enough. Then I pursued by excluding some parts of the methods from the parallelisation with spinlocks, keeping only the read-only and safe memory accesses in the threaded logic by checking critical data dependency especially regarding the physical grid access talked about just earlier. Spinlocks are commonly referred to as a mechanism in which the thread would constantly try to acquire a resource in a loop until it's a free. It's a way to handle thread scheduling for critical sections. That said, it should carefully be used, as having a thread spending a lot of time acquiring a spinlock would result in a massive performance drop. In this situation, it is usually best to use an other mechanism such as more complex semaphore to put the thread to sleep and having it awaken when resource if free or the thread having actual work to perform. But for now I was still focusing on the proof of concept. To rephrase and summarise my procedure, I will now lists the consecutive steps I had planned:

1) Get inspiration from previous commit using the thread pool,
2) Set up the thread pool and add lambdas in loop,
3) Find the causes of the breakages,
4) Add barriers, but results in worst performances,
5) Add spinlocks on critical resources only,
6) Disable logging,
7) Get utility classes local: keep a coherent state thread-wise, and avoid re-allocations,
8) Fix sound manager calls to static library classes,
9) Fix concurrent access to physical grid (especially for object deletion),
10) Generalize to all character types (as dead soldiers were handled out of the units),
11) Clean the code, fix memory leaks and keep track of the whole process,
12) Prove the safety regarding the game state for the proof of concept.

Here are the design choices I was facing going through that proof of concept:

- threading at unit level,
- constantly re-allocated utility methods on the heap, make it static or put it on stack rather than heap, allocate as many instances as there are thread and put them in arrays to avoid any re-allocation,
- which critical resources should be safe,
- thread number (a many as logical cores),
- thread yielding (spinlocks) versus putting them to sleep,
- whether to have arrays of instances on the heap or a local instance of the stack for other resources.

Finally one last method was causing issues, and ironically enough it was the actual one doing the path finding. The reason is, it shared quite a lot of global members and classes, for seemingly more convenient but risky logic implementation. This class was also massively inherited for a whole lot of different "acts", actions that

the soldier could undertake, which means at least as many different underlying logic implementation. For example if the player were to die, or drown in water, these are 2 specific different branching logics, but those are evenly quite different from the ones where a soldier would attack or defend, itself quite different from a soldier who would move or charge and so on, not to mention damages and other states resolution. I also had to carefully monitor concurrent access to mesh animation updates as the skeleton had a very intricate implementation sharing a base global static class and an isolated local skeletal animation state tracking. Actually frame pacing was atrocious. Frame pacing should be as important as taking FPS into account, it is a a term used to describe if the time allocated for consecutive frames to be computed and rendered is steady and evenly distributed among frames or rather, in case of actual pacing, having frames taking way too much time than others resulting in visual discomfort in the shape of hangs. Though with all the changes that were undergoing at the same time, I couldn't tell if it was only my own responsibility but it felt quite alike. Anyway I finally got some mixed results.

TABLE

	Soldiers	Multi-Threading	~~Multi-Threading~~		Soldiers	Multi-Threading	~~Multi-Threading~~
Debug	19 200	30 FPS	45 FPS	Release	19 200	150 FPS	120 FPS
	38 400	10 FPS	20 FPS		38 400	50 FPS	10 FPS

Table IV: Results comparison of the multi-threading refactor in FPS

I have spent almost a hundred hours on this task, and reaching to the project lead, the best thing was to get an other senior developer involved. By the time I passed down the task to that senior developer, the renderer made so much progress that you just noticed we were reaching 10 frames per seconds on debug builds with max soldiers in the same conditions we used to obtain not even 1 FPS when I started working on this, weeks earlier. As a follow-up, the senior developer took a fundamentally more rigorous method, and spent even more time assessing the situation. The first changes were threading the logging logic which was inherently incompatible. Coming to the same realisation as me that all details of path finding had linearised intricate relations, he then started to break down the logic into "Pre" and "Post" path finding computation. This way he could parallelise the logic happening before the troublesome parts but also the states resolutions and updates taking place rather late, leaving the most intricate part intact. While succeeding, his results weren't significantly better, and after facing regressions in the battle mode state and issuing a few bugs, he raised concerns about a part I didn't even brought my attention on, multiplayer battle were at risk of suffering from de-synchronisations between players This is the worst thing that can actually happen. During a multiplayer experience, the tacit agreement players unknowingly agree upon is that they play the same game under the same state. Because the gameplay establishes rules, ending up with an

inconsistent game state that would break those rules, would also ruined the whole experience. Hence the reason, if we go for a multi-threaded implementation of the path finding it had to remain deterministic. And as network implementations such as multiplayer modes are opened to a lot a unexpected states, it's already complicated to have a clean and stable linearised multiplayer logic but safely converting it to a multi-thread implementation was just too risky and it finally called the end of this task in the project as a lot of work remained and our attention was required elsewhere.

By week 14 yet another CMakeLists.txt missing entries just to recall actively working as a team on a cross-platform project can sometime be more complicated than expected and that the schedule can very quickly be affected in drastic ways. By the time I also had confirmation that the person who used the iMac before me wasn't returning and I could finally benefit running macOS from the internal hard drive rather than the external one, hopefully improving my workflow. I had to transfer files and re-install some software but the transition was relatively quick and easy this time.

Main Menu Overhaul – Week 15

From now on and until the end of the placement I was exclusively assigned to the main menu overhaul. The reason is not having any developer active on this section of the game meant someone had to build knowledge around this part, and this person was likely to be in charge of the main menu until it is basically finished. We also started switching more developers to this project and established a Gantt chart to organise advancement on this project, those changes were likely followed by some internal discussions regarding the timeline of the project which for obvious can still not be disclosed.

We also started to assess deployed builds quality in order to ease the work of the QA team. Unfortunately we faced a data mismatch and the game couldn't file the data folder in Windows and Linux versions. This was a consequence of having the macOS application having a specific data folder and different implementations to find the game data root folder. The manual fix was easy enough though, moving the folder to the default data folder location. Yet it's not a satisfying solution and in no mean a state in which to ship the game. In the end this issue was related to our he building and deployment tools, hence could only be fixed by either the tools developers or adding overrides to the in-house data mapping virtual file system I previously mentioned about DiRT4. Sadly, the the mapping configuration for the project was already quite dense because of the previous versions worked on, and any change on it required carefully going through the thousands of entries to assess the dependencies, meaning it was a huge task on its own and I couldn't spend this much time on it. Not to mention issues with having Linux using case sensitive file systems while macOS and Windows are using case preserving file systems. The rule was lowercase everything, yet the Art team have conventions and is using upper-cased prefixes or

suffixes in the filenames on a daily basis. To this moment of writing those lines, we had a few improvements but the issue still have not been definitely addressed.

Fundamentally the goal for the main menu was do everything needed in order for it to look like the Game Design Document mockups provided by the Design team. The Game Design Document involved in most video game development is a reference covering each aspect of the game, from gameplay experience, to visual aspect going through market studies and marketing plan. In this case the GDD was limited to gameplay and visuals. On the other hand mockups refer to layout drafts for the user interface. The reason we needed mockups and a UI overhaul in first place, as we're supposed when porting the game to keep it as close as the original experience, is that given the hardware it's shipped to, we always need to adapt the controls and the UI. For example a game that is ported to mobile devices wasn't meant for these platforms, so we need to create a new interaction system to make the "feel" more comfortable to suit the intended experience. In this case high definition assets being brought to the game along with a renderer improvement, the UI was improved as well.

The assessment step consisted in going back and forth the Game Design Document and the actual state of the game and note down all mandated tasks. On the 6 sections of the main menu page in the GDD, I noted down 23 tasks mostly about visual updates but also about new logic implementation such as new screens, but I also took note of 44 bugs yet to be fixed specifically focus on the main menu.

As usual it was all about finding the code entry point and building knowledge, where in code are the main menu assets allocation, interaction input logic, screens management, also how is the game loading and storing assets, in what format are the data and configuration files. And the first step was to go fishing by doing a global search on the whole project any UI element interaction, setting breakpoints and going through the stack trace and execution flow noting down interesting places and classes.

The user interface having evolved to present 2 panels, layout screens shortcuts for the different modes on the left and the actual underlying layout screens of the right.

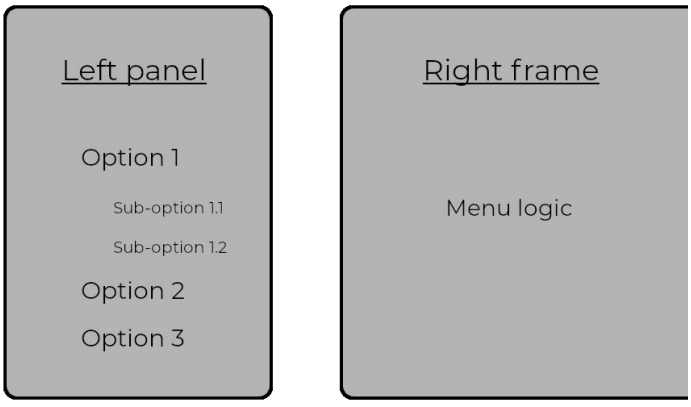


Figure 4: Main menu panels illustration

Notwithstanding, the original legacy main menu system used layouts stacks incompatible with this design. Each time a button would be pressed, the new screen was inserted on top of the stack, and left and right arrows were used to go further in the game menus or go back towards the top level screen. But having shortcuts on the left panel made possible to infinitely increment this stack potentially leading to very wrong issues. To detail the functioning of the main menu it used a base class the "narrator" which holds everything together, whether it is loading data, initialising menus, setting game modes and so on. This base entry point class allocates and hold a stack of menu class instances, those act as containers for all underlying UI elements, whether it is frames, texts, buttons, textures or anything. For each instance of menu class there is also an history stack keeping track of all successive layout screens the player went through as just said above.

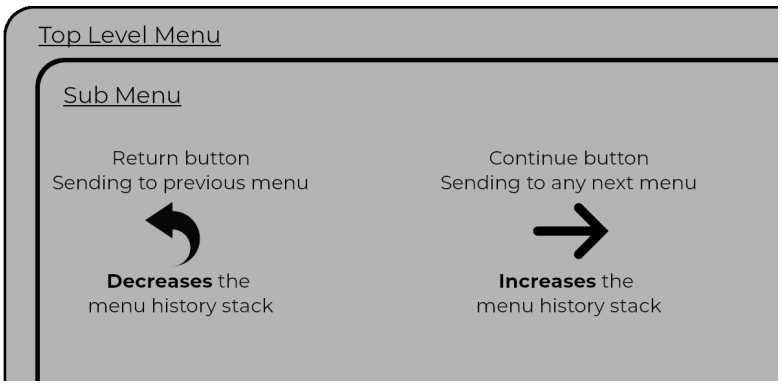


Figure 5: Menu history stack illustration

The first menu instance handles the background assets, the second one manages the actual menus. This allows to have a fancy fade in/fade out animation not affecting the background. Though going through the code, the animation is a simple inherited class for the base menu one which only overrides the drawing to screen method. While the base menu class had a lot more logic in it and this feels like such transition effect was a very late addition to the engine in the original game development timeline. The third instance is only holding superimposed dialogs, for example when deleting a save file or setting incorrect game mode parameters. It helped me to draw a tree to visualise the different sub-menus and which were leading to each others.

The menus are using with top priority an overridden folder in which we put XML format files specifying the elements in each layout screen. Those elements are identified by an eponymous identifier tag which is mapped to a user interface piece "UIP" class, designating an element which is finally implemented as a user interface element UI. E.g. when a button ID "id_button1" string is recognised, a "button1" object is allocated. It uses a custom "placement *new*" operator to reach a memory pool

belonging to the menu class the element is allocated into. I will talk about that more in depth later. I later found out about a file containing all the default layouts, I will refer to this file a “lnt” file yet it doesn’t stand for anything in particular. Those file will later be involved and discussed regarding the textures updates. I used those XML layouts to find out about each “top-level” buttons there was out there needing their their behaviour to be replaced. Anyway I first fixed this by adding a new button UI element overriding the behaviour to reset the stack whenever such “top-level” button was used, in other words I enhanced the base button class and appropriately update the constructors calls for the elements found in the XML. The UI button elements here usually have a virtual method which is triggered when the button is activated which is not quite the same as just clicking it for which there the mouse over, up and down methods. This is were I modified the behaviour, adding a flag to signal for a pending stack reset. This is were it became technically a bit more rigorous. The game as its own libraries making up for a good part of the engine. Following a quite sensed structure, the game usually knows about the libraries but not the other way around. But in our precise case, an inherited element from base class adds new logic and the actual history stack management and layout screen transition takes places in the libraries and there’s no way the libraries ever came to know about the child classes in the game code. The transition between layouts screens happen so that when a menu finished fading out it is de-allocated and the new one is allocated before starting a fade in transition, and this assertion will reveal quite important for the rest of the main menu work. To sort out the library issue, upon an active flag I added the stack reset between the fade out and the fade in because otherwise, we would face some null pointers dereferencing and that’s still very bad, we don’t want that. The final step was adding the flag to the transition method which is called by the child buttons and all was in place. Though because of some panels duplicated across two different locations I ended up with an issue where a save/load screen several sub-menus down, was going back to the top level menu because its stack got reset. I fixed that way later having yet a new button class whose calls to the layout transition set the flag to false.

The next step was updating the XML files in order for the UI elements to match the GDD. Those UI pieces declared in the XML with the identifier tag also had various tags such as horizontal and vertical positions but also the width and height tags for their size, texts have an (unfortunately unused) font one, while while buttons have a menu ID with the name of the menu to spawn and transition to upon activation and finally an art ID tag which references the name of a sprite that the UI element is meant to be visually represented with. These are all the relevant tags for now. It was a long work of updating all positions and sizes of the UI pieces in the the XML files, removing the irrelevant ones and moving some from a layout to another. Eventually re-ordering all elements so they would be grouped together for better readability for further changes and also standardising the mixed indentations. I also started to keep track of the strings that needed updates as we are using a convenient string depot

system manageable through web interfaces that automatically ships localised text files to the project repository when we run the according action via the build and deploy system. It is mainly used by the localisation team and we have the in-house library to support the localisation files on the game side.

I also came to fix the special weapons that were to used in a special game mode, the work on the previous version of the game implemented a new type of interaction and the UI elements for those weapons were constantly hidden behind a flag which would stay false because the interaction didn't make it to this version. I simply overridden back the logic so it doesn't take the flag into account for this specific version. I finally removed the the special weapons tabs UI elements which used to control the flag but were made ineffective.

I then fixed an issue where we wanted to have an original setting being controlled in several places, but because the original setting was behind an "advanced menu" flag, toggled by a checkbox element, it wouldn't appear unless this flag was true. Once again I added a new UI piece to adapt the behaviour as we wanted both versions, the one behind the flag and one which is not.

Week 16

The next task took the whole week. In the game modes there is a notification system which tells you about pretty much what is happening in the game in case you were to not pay close attention to all details. Design team wanted to have the ability to individually select which notification type was to be shown, in other words some kind of masking system for the notifications the player deems irrelevant and would hinder its experience more than actually assist him. The preferences class is an already existing static class that holds every single setting in the game, it also has methods to save and load those settings to file. It was the perfect place to add this extra setting. The idea was to have the menus using getters to display and setters to update the notification type individual flags from the preferences class, and having the notification system using getters to check whether a notification type would make it through or not. The first class is the base notification mask class holding the values for each individual flag, it was inherited into 3 distinct classes for the categories mentioned later. The second base class was the checkboxes list, it is a UI element holding a reference to one notification mask instance each. Based on an already existing UI list class, I only had to modify it a bit to fulfil my purpose, placing a checkbox on the left side and the text on the right side of an entry.

Though there are more than 150 notifications types so I was not going to add every single checkbox element out of a 150 entry-long list. Those notifications are split into 3 categories, Alerts, News and Reports and this was my first mistake. There is no fundamental difference for notifications of either type, yet I persisted to isolate them. Notification type being values of an enumeration class, we would need to map a

vector of boolean flags to those enums in order to hold then into the notification mask class. I first used a std::map<NOTIFICATION_ENUM, bool>. But the notification should also be rendered alphabetically, so I needed to also keep track of the order the enums were added to the displayed list. Finally I reckoned that it was too much complex logic for the purpose and it basically was duplicating initialisation and management logic in 3 places with just different values. I finally merged them all, it turned out the 3 notifications type categories partitions actually formed a set, thus I could drop the map container and have a single vector of boolean flags, taking advantage of the C++ implementation of std::vector<bool> optimisations, not significant but still interesting, memory management wise. The enumeration values would be used as indices to index this vector. Finally after an advice from a senior developer, he convinced me to use raw arrays because it would skip the std::vector allocations and initialisation (dynamic binding) because raw arrays values were already known and using primitive types, I could therefore benefit from static array initialisation at compile time (static binding). Finally I used for the settings array an in house bit array than I won't further mention as behave relatively the same as std::vector<bool> but just a bit lower level.

Finally all there was left was the serialisation logic to save and load these 150+ settings. The question was how to efficiently save this many settings, exporting as many entries was out of the question, and exporting 150+ comma separated consecutive integers was quite inelegant. Luckily the registry used to export settings already mentioned in the DiRT4 part and which is used for every game, could save keys in the form of primitive type but also raw binary data. What is better than raw binary to encore individual bits. The code was pretty straightforward, I declared an opaque binary data type from the in-house library specifying its size to (NUMBER_OF_EVENT - 1) / 8 + 1. This way I had exactly the minimum amount of bytes required. I then declared what looks like a parser as the binary data type has a method which returns a uint_8 pointer. Then all was left was trivial binary "OR" operations within a loop on that data type.

```
// "data" is the binary data type previously initialised
uint8_t* settings = data.getPtr();
// save logic
for (int i = 0; i < NUMBER_OF_EVENTS; ++i)
	settings[i/8] |= m_value_array[i] << i % 8;
return data;
// load logic
for (int i = 0; I < NUMBER_OF_EVENTS; ++i)
	m_value_array[i] = (1 << i % 8) & settings[i/8];
```

Finally in the notification system itself, the first version of this masking system trivially took only one line to take the change into account not accounting for the curly brackets and the preprocessor conditional directives. Upon the reception of a

new notification message, it would be routed to a category based on its type enum value, using the same in a conditional statement checked against the masking flag allowed us to instantly discard it or keep it.

Week 17

This week was dedicated to continue fixing and adding small features to the main menu.

In the first place added difficulty ratings for the playable ratings. This feature while present in the original version of the game, wasn't fully supported. I found the UI element in which the coloured coded difficulty text should have been added and implemented a value retrieval from a new json file created for the occasion in order to leave the difficulty indicator easily customisation for further Design decisions change about it. The coloured text rendering was already implemented but I seized the opportunity to fix it along the way. How the previously mentioned fade in/fade out transition between layouts works is just an update for the alpha of the globally defined sprite colour. And the coloured text wasn't affected by it, it probably was a typo in original code. All I had to do is make the overridden colour use the current alpha in the drawing method instead of having a fixed total opacity.

In addition I added checkboxes templates for features override for some game modes that weren't implemented yet and that Design still had to discuss. Checkboxes elements are part of the original game so that was pretty straightforward.

Moreover I added a campaign tutorial checkbox for a specific playable scenario as the tutorial confusingly used to be launched from its own specific place in the menus. A simple scenario check whether to display and enable or not the checkbox was just enough.

In like manner I added several missing strings to the depot to be localised.

Furthermore I fixed a missing button. Actually, the button was still there, but its sprite couldn't be found so it didn't rendered but it was still working. I just fixed the entry for the sprite in the aforementioned "lnt" file.

Week 18 (July)

This week signed the start of a major breakthrough, and a constantly increasing pace. What I designed as a duplicated left panel holding "top-level" buttons should have included more intricate logic. I quite didn't understand the goal we were aiming at by only looking at the mockups. While I take full responsibility on the confusion, this is an example of the limits prototypes and mockups (that could be compared to story boards in movie production) can ultimately show. I received new mockups of a more final visual pass along with final assets from the Art team. The

“top-level” buttons should have been expanding to unveil sub-menus with a fancy animation and highlighting assets instead of just leading to layouts as it did at this point.

This was proven quite a huge challenge. As I previously explained the stacked main menu functioning, any UI element loaded from the XML would belong to a layout and would be only allocated for the given layout. The new left menu panel should keep a persistent state and staying shown on screen not being affected by the fading transition effect. Although the first good news was because it mainly consisted of well spatially organised and aligned buttons, there was no need to dedicate an XML file to it. Indeed its major dynamic property meant it couldn’t easily mix with a static XML UI element referencing and the logic actually needed to be coded. As the design should have been shared with the game modes pause menus, it was mandatory to assess the feasibility not only for the main menu but also for the pause menus. I’ll come back to this topic soon enough but I learned to my extents that the main menu UI and the in-game UI were two fundamentally different and isolated systems not sharing any base class. It turns out the UI render pass shader was a simple pass through, usually designating vertex shaders that only pass down vertices to the fragment shader which itself only apply a simple diffuse colour or texture with any complex lighting or depth management. In other words the rendering code was stacking sprites in a buffer that would be drawn when reaching full capacity or when all the sprites accounted for. The game changing feature that saved me a lot of extra development time was the clipping region class. The clip region class is a legacy intermediary sprites rendering static class. To be more specific, it is a rectangle structure with 2 static methods *push*() and *pop*(). It worked by first declaring a clip region rectangle object and pushing it. Upon sprite drawing methods calling the renderer method. The renderer would retrieve this clip region rectangle and cleverly crop any part of the sprite ending up outside of the rectangle by passing down updated texture “UV” or not even passing down the sprite at all to the next method. “UV” refers to texture coordinates, using “u” and “v” letters to denote 2D image space axes commonly opposed to the 3D space axes “x” and “y” and “z”. Here we mostly used spritesheets which are texture containing multiple sprites. The .lnt file defined for each sprite, here called “regions”, 4 points: top-left, top-right, bottom-left and bottom-right which are representing the vertices of the bounding box of the sprite. With UV coordinates going from 0 to 1 in both axis on a spritesheet starting at the bottom-left, a sprite at the position {(256, 128); (320,128); (256, 256); (320,256)} in a 512x512 texture would be converted in {(0.5, 0.25); (0.625, 0.25); (0.5, 0.5); (0.625, 0.5)} in UV coordinates of that spritesheet.

The next renderer method was consequently called and would actually store the sprite in the sprite buffer to draw loaded on GPU but that’s as far as I was meant

to venture. Here is the plan I established to overcome this task aiming to create a panel class to decouple the left and right panels:

- Implement the top-level menu hierarchy with containers;
- Implement the underlying sub-menus lists for relevant top-level buttons;
- Implement the expanding "drop-down" capability;
- Implement the buttons interaction with correct behaviour (either expanding or spawning right side layouts)
- Allocate and dynamically place and render the art assets,

Yet the challenge remained to get a persistent left menu panel not affected by the menu layer level, menu class container. I started carefully looking at the place those menu class container instances were declared and allocated, coming back to that narrator class previously mentioned. As it hold everything main menu related, I found this place next to loading of resources such as sprites, fonts manager, strings banks. The first instinctive idea was to allocate this left menu panel in the background menu class container, the deepest layer in the menus stack. I tried this solution but as already mentioned the top element of the stack would capture any mouse input, leaving no interaction possible with deeper layers and ultimately our left panel attempt. So I came up with the solution of using the background layer memory pool but add it to the menus layer as a private member. This way I only had to augment the menu class event methods to pass down the event method calls chain to the left panel and I could even call the draw method outside of the transition effect logic of the draw method. This could seem like a very nonsense choice but this solution had 2 advantages, obviously the first being the direct connexion to the menus layer with seamless integration of all behaviour (update, drawing, input events). The second one is, the left panel would alike the background be allocated once for the entire period spent on the main menu, it is allocated first on main menu opening and de-allocated last on main menu exit, and finally it would keep a persistent states regardless of the successive layouts transitions.

This is where the "placement *new*" operator started to make a lot of sense. "*New*" is a C++ operator that "creates and initialises objects with dynamic storage duration, that is, objects whose lifetime is not limited by the scope in which they were created" according to the official documentation [12]. Roughly, it allocates objects on the heap and is used cooperatively with the "*delete*" operator that de-allocates objects form the stack. It was probably introduced as a convenient mechanism to avoid intricate *mallocs* (memory allocation) calls mandatory in plain C. Time passing by, smart pointers became a thing, they conveniently de-allocate themselves when going out of scope, an idiom emerged "never use new except when using a std::unique_ptr", stating the heap allocation should be used as little as possible to avoid unnecessary memory management issues and leaks. Indeed it is actually fairly easy to create memory leaks by allocating anonymous objects on the heap can easily be lost and remain unreachable for the whole execution of the program. By the time

std::make_unique<T>() appeared the idiom was updated to "*never use new*", answering the question of clean memory management. In spite of these good practices, there are a few cases where we want a more complex and customized memory management. It's probably to this purpose that the "placement new" operator was introduced long ago. Placement new operator uses the syntax "`new (placement_params) ( type ) initializer(optional)`" and specifies an "allocation function [to which are] passed [placement_params] as additional arguments". In other worlds it leaves the capability of passing down a memory address where the allocation should take place to the constructor. However we can customise further the behaviour by overriding the "`void* operator new(std::size_t, void*)`" operator. This is what was taken place in this game for all UI element related to the main menu. The new operator of the base component class being overridden, it prevented the definition of implicit constructors and no UI elements was ever allowed to be allocated without a subsequent valid menu class container object. It seemed rather overkill at first sight, and eventually rather inconvenient sometimes but it actually enforced a strict memory management free from any unwanted easy memory management issues. There are undoubtedly better solutions but this is efficient. To resume my words about the task I was assigned, taking advantage of the placement new to use the background layer memory pool while connect the left panel UI element to the menus layout was the key in making it real. However because of the uncommon essence of this UI element, I manually allocated the panel next to the allocation of the menu class objects, but it was fine as the background layer de-allocation would took care of it during main menu exit. At this point we should remember we're working on a rather old code base that didn't benefit from all the recent C++ additions, though to sum up this solution is a rather daring hack on top of an original quite old daring hack.

Week 19

With a persistent left panel element across layouts, I could start the actual logic of the class. The first parts were trivial, using an abstract base button vector to hold the top level buttons, I also created a proxy button. The proxy button was a button that would be expandable, not leading to any layouts. The reason for not having only proxy buttons is some "top-level" buttons would have no point in being expandable as their behaviour was to resume a previously saved game directly or quit the game and for this reason, they had to conserve their behaviour from original legacy game code. Those proxy buttons would in turn hold another abstract base button vector for all the specialised button leading to different layouts, there the reason is some layouts required game state initialisation prior to just summon the new layout. Obviously those buttons remain UI elements and were subsequently allocated on the background layer as well (the panel holding a pointer to it among its private members). Nevertheless, the button position wasn't determined upon creation as the it would dynamically change anyway upon any of the "top-level" button expansion. At first it

was fairly simple, during the update processing method called at each frame, the position re-calculation was triggered thanks to a “pending change” flag which was enabled on mouse events methods on expanding buttons. Inside this re-calculation logic there is a loop going through all the top level buttons. In turn, inside this loop I gave the buttons as much width as the left panel was wide, specifying the top left origin horizontal component the same as the panel element. Thanks to a convenient text justify feature the buttons were conveniently centered. Though this feature only computed the length of the string, taking the base origin point, it would add half the width to center the origin then subtract half the length of the string to result in a centered string. I manually used the same calculation multiple times in the coming parts. After each button addition I’d increment a Y axis offset that I would keep track of to specify the height of the top-left origin point of the buttons. They were attributed fixed height roughly over the vertical size of the font used. If there was an expanded button, and we were reaching it, I then go over the sub-buttons inside the vector of this “top-level” proxy button, updating their position and keep increasing the Y offset. Of course each button were followed by a fixed spacing, tall for “top-level” buttons and short for sub-buttons.

In this first version I had a functioning button panel. Overriding the input methods such as mouse up/down/over handling, I quickly managed to bring the original and intended interaction behaviour to the several buttons. But my journey was far from over yet. With this version the button immediately expanded and collapsed. And the visual goal was a smooth fancy transition where we could see the bottom buttons actually going down to make room for the appearing new sub-buttons.

It was time to put the clipping region class to good use, after a lot of trial and error in the spacings accounting, the procedure was actually rather simple. First we needed the height of the expanding sub-buttons list, which because the button height and the spacing was fixed was easy to compute. Then the trick was to introduce the timing, when an expanding button was enabled, a timestamp was saved, then all there was to it was scaling the height of the sub-button list with a cross-multiplication from the timestamp and the desired total duration. For debugging purposes I fixed that duration to several seconds, but Design final decision agreed on having the expansion/collapse happening in about a third of a second.

```
n * button_height + n-1 * spacing /* → factorised: */ (n-1) * (2 * button_height + spacing)
expansion_height = total_height * expansion_time / total_expansion_time
collapse_height = total_height * (1 - collapse_time / total_collapse_time)
```

It can be seen as total height multiplied by the percentage of the expansion process. Here expansion and collapse remained the same for convenience. With that expansion height, we keep track of the vertical offset after the last “top-level” button and have a separate vertical offset for the sub-buttons. The next “top-level” button is vertically placed at the last vertical offset plus the expansion height.

Of course, those last “top-level” buttons would appear superimposed with the sub-buttons and that’s because until now I only talked about the position calculations, happening in the update method. On the other hand the rendering takes place elsewhere in the drawing method. Yet the logic remains around the same, I go through the “top-level” buttons and propagate the drawing method on themselves. We reaching an expanded “top-level” button I go down its own vector of sub-buttons but instead of drawing them all, if one is to be placed under the computed expansion height, it is disabled and hidden so any interaction with it is prevented and it’s not shown on screen. We’re closing on on our objective but the result is button “popping” out of the blue which is fun but visually not the best we can do. All there was left was to introduce the so convenient clipping region, the rectangle size is updated each frame and to avoid allocation each frame, the clip region objects are actually created in the constructor. Finally the rectangle is pushed before calling sub-buttons drawing methods and popped right after. We have our fancy simplistic expanding and collapsing behaviour, done.

The result was pretty convincing but floating clickable strings are not quite yet what players would expect from an AAA production. I started inserting assets to the left panel class. There was 2 main way of doing so, first the spritesheets as previously mentioned although their size is limited for efficiency reasons and the background assets for the panel were quite huge and there was absolutely no point populating spritesheets with only one asset. Assets in high definition are usually way bigger than their display size, the reason is probably about super-sampling, having more data than actually used help to achieve better visual results with eventual interpolations. The other method to insert assets was to use “stock pages”, in this legacy codebase it was a special mechanism to load huge or special assets. Rather simply, the background assets were scaled to the width of the panel and put one under the other. Though it is worth mentioning the background footer was a decoration that had to be rendered before the previous one as the bottom background were made to be tiled. This is a clever way not to tie the assets sequence size to the display size if need be to adjust the horizontal size, artists don’t have to redraw the whole assets. Final visual overhaul was introducing the highlighting logic and decorations, in order for the user to directly spot which button is expanded and which one is hovered by the cursor. The decoration sprite were small enough this time to fit in the spritesheet. Henceforth I will now describe the assets addition procedure.

First I usually receive sprites either by mail, instant messaging or if this should undergo a rather more formal process, it is directly committed in the gamedata repository by the Art team. This time I received pictures in PNG format. For the stock pages, I just converted them in TGA format using the free image editing software Gimp after carefully asking the Art team of any precaution to take. For the spritesheets we use the proprietary licensed *TexturePacker* software.

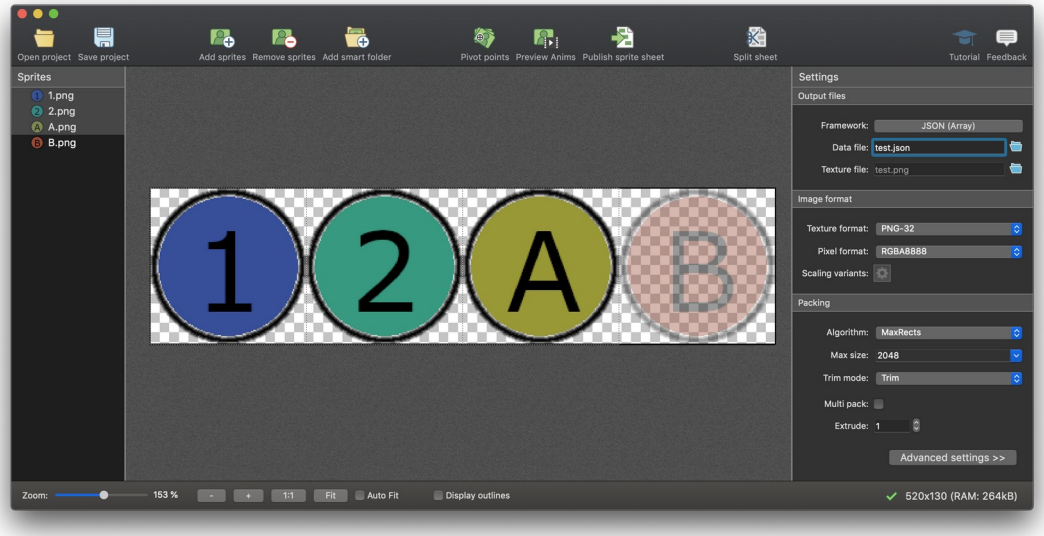


Figure 6: Texture Packer spritesheet illustration made out of 4 128x128 sprites

It simply generates a spritesheet out of the images it finds in the folder you point it at, providing control over the sorting algorithm, the dimensions, the output format and so on. There was a small issue with the exported TGA spritesheets but exporting PNG and manually converting them to TGA did just fine. Finally the we need to tell the game how to recognise all those sprites. *TexturePacker* exports a text file that references each sprite, with their name, coordinates and sizes. It can be exported in various formats, even though json was chosen here. We actually had a perl script at out disposal, written for previous versions of the game, that would convert the json file format to what would the game expect in the XML formatted famous .lnt file. It is now time to talk about the "lnt" files. They are arranged in several parts, and can unfortunately reach several dozens thousands lines. The first part is dedicated to the textures, a "*Texture*" element with a "*path*" tag and as many "*region*" tags as there sprites. The "*region*" tag, as mentioned, specifies sprites themselves by their bounding box, top, bottom, left, right coordinates. This is the first part to edit, basically copy/pasting to update all the "*region*" tags extracted and converted from the *TexturePacker* JSON file, for a given "*texture*" tag. The purpose of the second part is to declare "*UI objects*", these objects are abstract UI elements which will be created at runtime. It can plain art like pictures, buttons with one region assigned per different state (normal, disabled, pressed, highlighted), or any other UI element needed at the exception of borders, sliders, checkboxes and radio buttons and the cursor as they are "stock" elements that should all be alike, hence made out of a factory class. A factory is a design pattern to whom is delegated the role of creating many instances of an specialized objects. A "*UI object*" node has a "*texture page*" tag referring the textures in the first part, a "*type*" tag to specify its role and as many "*state*" tags needed with an

underlying "*region*" tag value. Finally the last but not the least part is the original XML files of the various layouts making up the whole main menu but regrouped in one place. It was the actual original place they were defined, and at this point I understood the prioritizing code I read several weeks earlier and that the actual XML files layouts were a custom override we did in-house. Ultimately, in the game, the feature have been there for quite a long time now along with a convenient art piece finding method to retrieve a pointer of the resource associated to the sprite ready for creating any UI element. All are piece retrieval should be check against a null pointer but other than that, that's all there is to it. With that procedure I added several decorations, I had to take them into account for the height calculations but nothing fiddly really.

When adding a new spritesheet to the "lnt" files, I face a rather odd issue. The display would flicker because of hangs and half of the sprites would render. After a quick investigation of the logs I found out that it happened after two textures allocations. Tracing back the allocation place, I found a virtual surface class. You can think about it as some kind yet another texture packing of the already packed textures. Here, a virtual surface is an arbitrarily sized texture segmented into 2048x2048 pixels hardcoded sized tiles. They are attributed whatever textures are added to the game if they fit into the tiles, and in this case huge high definition assets spritesheets. Increasing the size of this virtual surface, resulting in more tiles available allowed to successfully allocate all the new assets. This virtual surface seems to be the buffers holding the textures and which is send to the GPU, but here again once the issue fixed I don't really have the time to look into this any further while it's behaving correctly.

Finally I used the aforementioned coloured font rendering for the buttons with different colours for whether the button string was lit (expanded or highlighted state) or not (normal state). The minor challenging part here was actually bringing back smaller fonts for "sub-buttons". I sadly discovered specifying a font at text resource creation was completely useless because during the rendering, the font was always overridden the hardcoded way, using global functions to retrieve shady font class pointers incompatible with the pointless system to specify fonts at creation. Another sign the original developers might have had ambitious plans for their engine but didn't quite had the time to undertake them and hacked the font support however they could. Fortunately I could retrieve one of those shady pointer for "Times New Roman" size 14, deep copying the whole underlying font object in which I overrode the hardcoded font resource allocation to use a smaller font instead. I ended up implementing the change next to those global functions as this could benefit other parts of the menu other than just the left panel.

Week 20

With the most challenging piece actually finished, I could focus on the major and hopefully final overhaul of the main menu visuals. Indeed I received the final visual mockups from the Art team, they were significantly different from the Design mockups but I'll talk about this a bit further. Meanwhile the Game Design Documents had severed a few major changes so my previous XML layout updates felt like a waste of time actually, and there probably was some truth in this but I managed to build knowledge about a quite intricate system and this is all I was asked to do. As said, the person that would take the main menu in charge was probably meant to go through it until it's finished.

Yet I faced a rather critical bug while undertaking the overhaul of the most logic dense layout. I started improving the layout that would require the most work, in order to be confronted to delays immediately. This would allow for adjustments if it was proven too long, rather than failing the deadlines by introducing delays too late.

There was several issues regarding the resolution change. The first one was incoherent window re-sizing. Having some 16:9 aspect ratio* resolutions ending up with a square size which was against any logical explanation. I found out that below a certain width, resolution weren't used and it would use a minimum of 768 pixels width. Though there was no check against the height so we ended up with height lower than 1024 but would consistently use a width of 768. Adding the width check fixed the issue.

The second one occurred upon resolution change that would effectively change the aspect ratio, the UI element were shown "off", in other words they would be rendered displaced, and the cursor was out of sync with the visuals. It was as if the physical emplacement was updated but not the rendering emplacement. It took me quite time to find the actual behaviour as I went to investigate all 19 different resolutions (6 unused because too low resolution, 4 in 4:3, 14 in 16:9 and 1 in 5:4 thanks to the huge resolution of Retina displays). Not to mention we are later supposed resolutions of 16:10 and up to ultra-wide 21:9 aspect ratio. I was receiving mixed reports from another UI developer, mainly focused on game modes (as they use a different UI system), to whom I asked to check several resolutions things in the game modes. It turns out, he was not facing this issue, and the reason is, instead of changing the resolution in game, he changed the resolution from the Feral launcher. Thanks to him I was on track to find the cause of the issue. For the previous versions of the game, the UI element alignment and anchoring features were added, overriding the low level rendering methods to translate from a 1024x768 screen space to the actual resolution. The consequence of this change is the UI elements would keep the position they were assigned at creation, and weren't updated anymore during the drawing, hence being completely off when changing aspect ratio. To confirm my hypothesis, I tweaked the projection matrix send to the GPU. The reason for this is, the projection matrix was effectively updated after an aspect ratio change. The tweak I inserted

would simply keep the same projection matrix after an aspect ratio change. As a result, the cursor and the UI elements were synchronized again, confirming my thesis.

However, the projection matrix update upon aspect ratio change is expected, and this was in no way constituting any kind of fix whatsoever. This is a case of very intricate display bug we exposed to. I came up with 3 solutions in mind, first, the brute force, de-allocate all UI elements and re-create them so they would have a correct position. This introduced severe memory management risks and safety of this workaround couldn't be easily proven. The second solution, probably the most sense effort/result ratio taking in to account. It consisted in having an exhaustive look through all UI elements to update their positions without any memory management implication, but the maths needed to be prior well thought through. Finally the cleanest and best solution was to revert back the final position calculation at the low level rendering drawing methods. Yet doing so would break all convenient anchoring and alignment features, implying fixing back the issue and porting the features to the new (old) drawing behaviour would add a massive amount of workload.

I had spent a week wrapping my head around this and I added yet more delay to my already late schedule. So I had to move back on to the visual overhaul I started.

Week 21

For each new logic that was consistent enough to be a class I used new files. This had the unfortunate tendency to break other builds, whether it was other platforms or other versions that didn't branched yet. The internal policy was branching was only meant to freeze the environment of a project close to release as I explained it for DiRT4. I could reflect the changes to macOS, Windows and Linux, but not all supported devices and I was at several occasions asked about those changes and reminded to try to be more careful. To minimise the side effects, I choose after a rather enlightening talk to concentrate all new classes in a single file dedicated for the new main menu UI elements, excepted the quite already dense left panel.

Still, it was about time to step up in this overhaul by matching the final Art mockup visuals. The first element was frames. Re-sizing them was as quick and easy as a change in the XML files. By the past, what visually represented the right frame was, if one can even make any sense of it, an empty box list object. Literally frame that is supposed to hold successive objects, though it always stayed empty it was used for the sole purpose it conveniently allocated its own border and also have an opaque background. Indeed it could contain any kind of element as there was an abstraction class for generic list items whose role was to be inherited into a specific object, whether it was icons, buttons, entries or even files. Defining a new element restricting its feature solely to be a frame with a border and a background sounded about right. As a matter of fact, the empty list would use the ready to use border class, definitely lowering the interest of creating an object with a complex and dense logic just for its

visual. It still can't make sense about why it was like that in the first place. Shortly I had an object with a border that mimicked the visuals with very few code lines, I only had a border instance as a member, initialised it in the constructor and overrode the draw method. I decided at this point to add new borders as part of the graphical overhaul. Borders were stored and managed as stock pages I already talked about. The underlying texture was split into a 16 cells grid, with the 4 corners, the 4 lines of the 2 axes (top/bottom and left/right), the center but also the additional 8 intersections of the vertical and horizontal axes even though I never saw this case ingame.

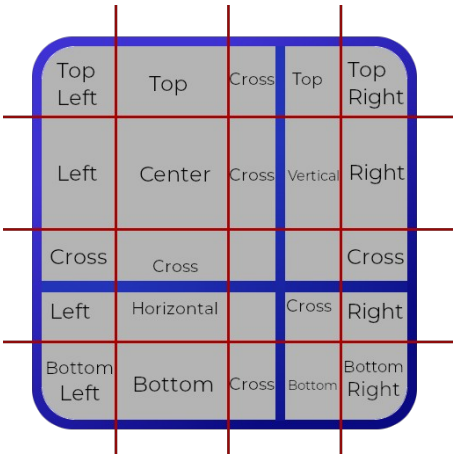


Figure 7: 4x4 Tiled border illustration

The PNG textures I was provided with, didn't match this layout nor they included those intersections, which was fine. I assembled those pieces into a single new stock page with an alpha channel. The procedure to add a stock page was pretty straightforward. First each stock page was attributed and identified with an enum, which I created. Second, that enum had to be mapped with a texture path, which I also added in the array made for this purpose. And finally the rest of logic was already implemented taking advantage of the already existing border system, all that remained was to change the enum used at border creation. It was half a surprise when I actually saw the result for the first time ingame. There was no background. The old borders didn't include any alpha channel and instead of being just borders, it included a black background along those lines and I understood at this moment that the frames background was actually the center cell tiled as many times needed to cover the entire frame. The issue was the expected new background wasn't just a plain solid colour and rather had large tiling patterns. The border system limitation was the stock page texture was equally split in 4 by 4 sub-textures and there was no way to go around this limitation. Having up to 32x32 sub-texture cells, which is already quite large for a several pixels wide border, prevented any chance of having the background included in these stock pages. So to sort this out, I implemented my own tiling mechanism using an plain art object holding sprites. I added the background sprite as an isolated stock page for my own purpose and taking advantage of the sprite drawing logic, I

computed the how many tiles were needed, store their positions in a vector and issued a draw request for all those positions with the same asset. The request were held in the sprite buffer to be rendered, mentioning which UV coordinates and which texture to use and where it was placed in the image space. I made it so the tile size was customisable, dynamically computing the required tiles positions. However, it would obviously not match the frame size perfectly unless spending extra efforts to fine tune the tile size. To circumvent that issue I simply created yet a new convenient clipping region around the frame dimensions so I could peacefully draw my tiles. Upon that, I also added another plain art object to place the decoration that would fit on the top border segment. The XML layouts defining one and only one isolated layout, it was mandatory to reference the frame in every single XML file, and to avoid the painful procedure of also adding decorations and carefully checking for the dozens of XML layout files to stay in sync at each minor change, I directly added the decoration to the frame class. As usual all, calculating the horizontally centered point and subtracting half of the decoration asset width, ended up with a perfectly aligned decoration.

There was an aftermath anyway, as the frame class was also a legitimate class which here, acted as a sub-frame for smaller elements with packed elements but not quite lists. For example it was used as a border for grouped UI elements such as icons. This time I was all set to add yet a new fancy border and override the enum wherever it was needed.

Then I went after checkboxes, also commonly known as “tickboxes” or even “toggle buttons”. Previously working on stock pages, I noticed they not only were used for large assets or borders but also for the stock elements, checkboxes, radio buttons, selection arrows and even sliders. So I decided to update all those elements I had sprites for. There’s not really much to it, all the logic stayed the same, the change was instantly reflected in game as I overrode the texture path instead of creating new stock pages. I didn’t do that for borders because the original ones were various, they had unmeaningful names and I didn’t updated all of them in one go so it still required some them. For the checkboxes and the sliders and the arrows, there was only one kind and we didn’t care much about the old ones anyway.

I thought I saved the least for the end, fonts. Well not just fonts but strings in general. The first issue was they were rendered in yellow, almost everywhere, it was part of the old colour scheme, but it didn’t blend at all with the new white lit-theme. Also most fonts had shadows underneath and the Design team wanted to discard all of them and shadows only to the left panel which lacked them at the moment. Finally, most strings were upper-cased and Design team chose to have them capitalised instead. For a time I wondered if I couldn’t just write a utility function to convert the strings in a capitalised fashion, but I remembered we were supporting several locales such as Cyrillic and finally didn’t take the risk to make too ambitious assumptions. That issue could be fixed evenly in a clean way on the in-house string database side

anyway. Regarding the shadows, there was just a few low level string rendering methods. All UI objects, even the most sophisticated ones, called either one of those methods. They basically translated strings into glyphs ant put them into the sprites buffer with the according font, size calculations and colours. Some of them were specialised for large strings to effectively render text blocs on multiple lines. I found that shadows were no more than just the same strings rendered in black before the actual ones with an offset of one pixel. I noticed there was already some kind of flag argument in those methods to obtain a way to conveniently enable or disable shadows at will. It was mandatory in a previous version of the game but remained intricate and only on a small part of the elements. With the left panel shadows requirement though there was no other way around to improve this logic and generalize it to all the strings wherever they were used. I obviously opted for having the shadows disabled by default as keeping the original behaviour would have meant tons of higher level overrides in UI objects. Finally for the left panel I add the flag to enable the shadows for base button class and the affected inherited classes. I only passed down the true flag value as argument when initialising the left panel buttons.

Ultimately on the colour issue, it was proven to be a real mess. The logic was split between a default colour in the low level strings rendering methods aforementioned, the global static sprite colour for controlling the overall actual rendering colour and custom overrides colour in some specialised UI objects. The low level part seemed easy. Rather, while taking the underlying intricate logic into account, it definitely was not. Some UI objects used higher level strings pairs objects, which is supposed to represent a left/right hand side string pair, what seemed to be a late addition was the support of sprite rendering alongside the strings. There was a close relation between text resources and art resources, sharing the states art struct that defined art resources for every each of the supported states, enabled, disabled, pressed, highlighted, focused. The underlying texture resources base art resource class was inherited into either text resources or art resources sharing common low level logic nonetheless. Fortunately the method was checking for textures first and would end up falling back to drawing text if no art resource was either set, valid or allocated. This might have been helpful during the initial development as I definitely saw similar patterns in DiRT4. If we are to think about it for a second, having text templates before getting the art resources is in the end quite convenient to progress in the development. Anyway during the fallback of strings rendering, I override the yellow-ish default colour to render black text instead. It was a mistake I only find out about several weeks after. In some the case of such strings pair in lists being sufficiently numerous to actually need a slider to browse the list, the sliders we to use the same colour as the text, I later ended up with black sprite slider and it felt very wrong. I didn't quite immediately found out about it because some lists would use colours override or strings on the right side of the pair which is yet another intricate piece of software (why not using the exactly same logic as left side of the pair was beyond my

comprehension). To this day I'm still working to sort out a list of terrains because lack of time to dedicate on this task. The terrain list goes along sprites to indicate the special property of whether it is possible to find buildings on the selected terrain. The final way to fix the low level default string colour rendering seemed obvious at last, I overrode the huge string renderer constructor I didn't pay attention to all this time. The value was there, in RGB which was a pale gold-ish colour. I subsequently set a black colour instead and low level part was finally done in a clean way but it took a lot of knowledge building. But it wasn't over yet. The global sprite colour state would define the default rendering colour of all elements, I already explained how the fade in and fade out effect worked by just updating the alpha channel of this global colour sate. What followed was a tremendous amount of back and forth between the game execution to notice wrongly coloured texts, XML layouts to identify the UI object affected and the draw methods of these UI objects. Most of them had an hardcoded colour management to finally temporarily update the global colour state. It wasn't hard but rather tedious. Not to mention places were there was an override aside the global colour state.

The following was updating the dropdown element look by inserting new assets from the artists. I started by formatting the arts assets to the format expected by the legacy border system explained before. On the code side, first thing was to override the drawing behaviour, it used to draw the "expanded" part first if the menu was expanded, then draw the "expander" on top (the button). We needed the new expanded part border to overlap the button so I reversed the order those elements are drawn. The whole task was more tricky than expected and there are 2 reasons for this.

First, the dropdown drawing method was riddled of hardcoded offsets for each and every part. In the new reversed order, we draw the button border (either with legacy system if not in focus or differently if on focus, see below), enabling a clip region and the text of the current value and the button icon, then discarding the clip region, followed by drawing the expanded menu selection with the border, enabling a new clip region, draw the background, discarding that clip region, enabling an ultimate clip region to draw the text values, draw a border which acts as a selection indicator on top of the currently hovered text value, finally drawing the slider if relevant and discarding that last clip region. Most borders having a different size, and as a matter of fact, all these steps having hardcoded offsets, I spent some time updating all of them.

Second, one of the new border was incompatible with the legacy border system. Indeed, having the vertical tiles the size of the corners prevented us from having decorations on the border sides. I first tried having a legacy border to add only the decorations on top but the seams were too obvious so I decided to add the whole vertical border sprites on top of the legacy border, if paying close attention there is a one off pixel seam but at this point I couldn't do much with the time I had. I suspect it

came from using integers rather than using floats until the very last step to minimize the cumulative approximation errors but it would have required a complete engine sprite drawing logic overhaul.

The dropdown element is finally composed of 4 different borders, one for highlighting, 1 tiled background and 2 texts, using 3 clip regions. Like previously mentioned, the text had to be updated to display a black colour rather and light yellow and before adjusting the clip regions, the text would overlap the expanding button and overshoot the bottom of the expanded part because the new border is slightly thinner.

Finally during this busy week, I eventually fixed a heap overflow I introduced several weeks prior. As I feared the menu history stack had a fixed size of them and because some buttons weren't set as "top-level" buttons, it was possible to continuously increment that stack until an ugly out of bounds issue.

Week 22 (August)

With most of the UI widgets updated, it was time to update all layouts so they match the GDD final mockup. I was advised to begin with the most complicated and dense layouts in order to precise my schedule and plan as soon as possible if and how much extra time I would need.

This challenging layout is the result from 2 merged legacy layouts. The purpose of the first one is to select the number of AI players and game options such as but not limited to different available armies and teams. The last layout was about customizing each player army.

I had to match the mockups even though I faced a small concern. The Design department mockups, whose purpose is gameplay, showed what and where the elements should be placed. Whereas the Art department mockups showed the final visual representation the game should have. But they were slightly out of sync. Dark theme for Design mockups whereas Art mockups were light-themed. Rectangle frames for Design while Art had squared frames. Elements changed place and were re-ordered. So I was told to use the Design placement with the Art visuals which turned out to be a bit confusing as elements had different sizes.

The first steps were straightforward. I started with small isolated elements, the size and location of the right hand side frame for which I previously made a dedicated class, the title, buttons, borders (already updated), labels (with fonts already sorted out).

I took some time to fix a display issue, the number of special weapons was showing billions instead of a range between 0 and 9. The issue originated in a legacy format method which would convert primitive types to UTF-16 strings. With the time I was given and already having delayed my own schedule, I couldn't tell if it was

because of the previous 32bits to 64bits conversion or even an issue introduced by the former versions of the game. My fix could have been called a hack as I deliberately tried to make it isolated and local. Upon that call to the format method, rather than passing down the 32 bits integer getter as parameter, because it seem to oddly non convert properly, I gave a final 32 bits bit wise OR expression 0x000F & value, to keep only the relevant least significant bits part. This way I limited my change impact to the sole parts I'm actually working on, making it easy to circumvent any side effect rather than introducing any regression that would hinder any other coworker's work.

The unit selection lists needed some heavy rework as well. The previous version of the second panel displayed a 5x4 cells grid, I converted back to the legacy behaviour which was 10x2. The change was simple enough, as it consisted in turning back the original for loop values. I needed to add the new art assets and update the grids sizes. The grid size is controlled by the size of one cell, a unit card slot, themselves determined by specific height and width defines. Changing this value could have introduced some regression in other part of the game where those defines were used. After a thorough check and eventually asking coworkers it the change seemed legitimate and safe. Fortunately the new assets were a bit easier to deal with than the legacy ones, conveniently getting rid of several entries in the XML layouts. Indeed there only was left, center and right parts for the first panel and a complete sprite for the second, as opposed to the cell based top-left, top-center, top-right, bottom-left, bottom-center and bottom-right original ones.

Merging layouts wasn't a trivial task to undertake and the legacy system lack some logic to handle this. I created a new class as it provided the best time over completed feature ratio rather than modifying existing classes and trying to have it working fine using global states on shared and fixed layouts. The selector panel should display clans icons and their properties. Through an icon list the player should be able to update the current selected playable army. Empty slots are represented as "plus" icons and teams are separated by "versus" icons in the shape of swords. The class had to dynamically respond and resize itself to any change involving army selection. The logic is as follow, first the number of "plus" icons are divided equally in 2 partitions for left and right, having the army icon in the middle. Then the needed spaces between each assets is summed up to compute the space size, detailed as the pseudo-code found in Appendix II – Pseudo-code of army selection icons positions calculation.

Finally the teams should be displayed in increasing order regardless of the slot. But because of the data structure being army slot driven, it turned out to be more complicated than expected to obtain the desired behaviour. The reason is, the legacy system used fixed assets that should dynamically be remapped to the first empty slot in increasing order. As a result, the class used for the icons actually uses that behaviour and it was hard to keep track of which icon instance accounted for which army slot.

```
match_icon_index(int player_index)
{
	int icon_index= -1;
	for (int i = 0; i <= player_number); i++)
		if (army[i].isValid()) { ++icon_index; }
	return icon_index;
}

match_army_pointer(int icon_index)
{
	int valid_slots = -1;
	for (auto army_ptr : armies_pointers)
	{
		if (army_ptr→isValid())
			valid_slots++;
		if (valid_slots >= icon_index)
			return army_ptr;
	}
	return nullptr;
}
```

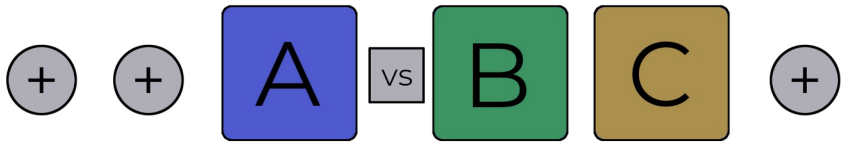


Figure 8: Player slot selection illustration

When the logic was right, I added all the UI for all properties around the icon eventually using low level string rendering methods directly. With that in mind, I've added all relevant UI elements that were present on the previous screen and supposed to make it in this panel. I faced a small challenge because relying on battling roles, the alliances system would dynamically change. All subsequent selected armies sharing parity teams number were assigned to the same alliance. In other words, team 1, 3, 5 or 7 ended up in the same alliance. Yet another hardcoded design choice to troubleshoot, not to mention to signify and test against the lack of such battling role was an explicitly check against a 0xffffffff value, very likely to be an unsigned integer negative one value. For rejoice though, the last element was used to remove player icon, which carried very few logic with it, this could have been more elaborate and clean but the logic was set the global selected slot to the player army to remove, set an invalid value and set back the previous globally selected player slot. I mean, it was the original legacy way of doing so and I didn't have time to improve the whole battle management system.

To keep track of what behaviour went in which method in the most efficient way, I initialised resources in the constructor, updated the positions and enabling states in the update method, and only rendered the objects with their pre-computed properties such as value, positions and sizes. Keeping the event behaviours in their respective methods. The usual mouse up/down/over basically only passing down the method call to underlying objects that checked themselves if the event was relevant or not. A better approach, rather that checking relevancy of events on each and every single objects would have been to use a delegates to actually send event signals to the relevant UI objects. The game modes UI system (still completely different from main menu) tried to accomplish that but we were years and years late for that suggestion.

Week 23

Finally because we were merging 2 layouts we lacked space, Design came with the solution of small sub-panels. Indeed, some elements weren't needed at the same time so the bottom left corner was used to welcome all three panels which are clans selection icons, army presets save and load and selected unit detailed information.

The clans selection icons lists was an original idea, usually sufficient clans icons placeholders would be added in the XML, being automatically initialised until we would run out of playable clans with a global static counter. Clearly not the best solution but that's how the original game worked. I didn't want to implement enabling, showing, disabling and hiding features to the clans UI elements because it meant a lot of state tracking, eventually out of synchronisation issues. Rather I created yet a new class in which I instantiated all those icons taking advantage of the legacy classes and controlled them in this very panel class. After figuring the dynamic positioning selection during the previous class, it was rather easy to position the icons on a grid of 4 by 5 given the panel dimensions at initialisation. The legacy icons elements were still used but packed in a more practical class, as a result the logic stayed the same and worked well with the selector previously implemented.

Next was the army presets panel, upon a button click, a new menu would appear to propose save and load features to the player for their army configuration. It was snapshots of the selected units lists the player had constituted, along with their properties. The original legacy menu really was just a superimposed list box with a text area to input the preset name and a couple buttons to save and load. Here again it was about concentrating those UI elements into one panel class to control their enabling at once. The UI had to be re-worked a bit, but all the art was previously added for the global overhaul so it wasn't much of a problem. Rather some odd behaviour showed up. Because those element were still enabled by the original button, half the other layout UI elements became disabled. Originally the preset menu was superimposed but on the same layer and this would prevent any element underneath

to be mistakenly messed with. I went through each of those UI elements and prevent their disabling whenever this former menu, now turned into a panel was activated.

With those panels implemented, there wasn't any more feature exclusively accessed by the former layout. I decided to merge the layouts at that point. By reworking the final layout to match the GDD, I progressively moved or built all the expected features and the former one became progressively redundant. It wasn't just linking the buttons to the final layouts rather than the former, but also to list all the logic that happened between transition between those. It turns out, the layouts use the same elements for the multiplayer game modes support, hence the buttons for the transition came along with some logic. Hopefully this wasn't much and there was very little logic to move. Finally transition button would check for the validity of settings (at least one valid player, and so on) on the intermediate layer. Those checks were re-done on the original final layout and even more in depth so there wasn't much to worry there. So with some minor changes, it was safe to link the top level buttons directly to the final reworked layout in the end.

By the end of the week, I got some help from an other developer. Initially, we thought the main menu and game modes UI systems were connected. When I found out they weren't, I also discovered I have not a single clue on how the game modes UI system worked. It would take me days if not weeks to build knowledge around it and duplicate the left panel I previously did to implement it on the pause menus. The helping developer was assigned this task as she already worked on this game previously. I spend a day trying to explain my thoughts and processes fearing my code would be messy and hard to comprehend because it sure wasn't the cleanest solution. It turns out replicating the left panel took weeks and I did no longer interact that much with the developer just after a few days as my code was intelligible enough. The systems really were proven very different and in the end there wasn't much more I could do.

Week 24

To finally finish the layout, the last panel to implement was the unit description panel. Upon right click on a unit from either the available or selected units lists, unit information should be displayed in place of the previously implemented panel. I had to match to campaign unit description panel so I already had the art at my disposal. I just needed to add the art pieces following the usual art addition process already explained. The panel description is composed of the detailed coloured properties and statistics along with a full description.

The most challenging part was the two lists have different semantics. The first one lists unit empty templates while the second actually lists units instances with underlying instanced properties. I simply created basic initialised units from the templates and could easily extract all the needed information from that. The coloured

text was already an implemented feature. For the statistics bars I was meant to add, I created a new class that held different horizontal assets representing whether the unit had a low, medium or high value in each characteristic. It was fairly straightforward as being an horizontal asset, it could be spread without any quality loss rather than being tiled and its horizontal size was a trivial interpolation of the max value and max size with the current value. The asset is updated and scaled in a dedicated method upon value change. Then trivially declared ant initialised as many instances as there were characteristics. Finally I added a flag controlled by the button clicking to show or not the description on the dynamic part instead of pros and cons and the statistic bars.

Week 25

This week was rather short as I asked for days off to effectively make the final advancements to this very report. Though, it was dedicated to get the graphics settings layout overhaul in place. All graphics settings belong to the preferences class instance mentioned earlier but the UI is hardcoded in both XML layouts and code logic behaviours. It mainly consisted of checkboxes, dropdowns and sliders. The whole process of creating the UI element classes, creating the underlying preferences entries, implementing the save and load methods into game saves and hooking those entries with the UI layout actually took the whole work week. There wasn't any major challenge other than the settings semantics. Some graphics features, such as anti-aliasing for example, weren't even implemented in the game yet. But to make sure everything is ready on time, I was asked to prepare the UI to welcome those features, meaning there will be yet a new pass to effectively hook the preferences entries to the shaders uniforms and rendering system. The graphics settings were previously agreed on by the graphics senior developer so Design and Art could make mockups and specify the actual settings values in the meantime. The vague semantics were discussed with the project lead developer and while I implemented the UI, he started thinking of setting value – actual shader configuration values translation unit, that will efficiently map a setting abstract semantic such as "Low", "Medium", "High" or "Ultra" to the desired values that Design, Artists and graphics developers will be able to easily tweak for fast feature implementation iterations.

Week 26

This was the final week before this report is due, it mostly consisted in updating the remaining XML layouts that didn't involve much new logic implementation to finally matched the Game Design Document for most layouts. By the end of the week all there was left was the multiplayer and the help related layouts which were of low priority and that will have the opportunity to finish later on.

Conclusion

This concludes the placement I have undertake at Feral Interactive Limited as a junior C++ cross platform developer, aiming at porting video games to desktop platforms. The work pace along the placement gradually built up to end up fast and demanding. From the newcomer steps meant to gather knowledge about the company policy and to actual in-depth UI development, the placement goals constantly evolved. The DiRT4 release was a perfect case to get started on. However the main project was proven more intricate to deal with and required a growing working effort. We're closing on the project which will later be announced and released.

The tasks I was assigned were as numerous as they were diverse. I started fixing some easy bugs on DiRT4 involving either UI, logic or input controllers. On the main project, it took a different turn, I started bringing back the Linux build to life, then investigated its state to finally focus on generic game port development. The major steps here were the header refactor patch, the multi-threading logic proof of concept and the main menu overhaul, specialising in UI for the latter. We should probably remember at this point that implemented solutions bring their share of caveats and eventually couldn't have been made better because firstly the lack of time to elaborate clean and optimised solutions and secondly because we are dealing with an intricate legacy code base which has its own constraints and limits.

I learned to use a broad range of new tools. First are the debugging tools, Apple's XCode *Instruments* allowed me troubleshoot intricate timing and memory management issues with convenient stack traces visualisation and browsing. I also came to understand and use the *sanitizers* family provided by modern compilers (*Undefined Behaviour*, *Memory* and so on) but especially the *AddressSanitizer* known as *ASAN*, it allowed to better find "use-after-free" pointers and other memory inconsistencies. Gdb (gnu debugger) was also proven useful to debug builds in Linux whether it was within QtCreator, directly from the terminal for troubleshooting deployed builds or remotely, not to interfere with the application window management. *Valgrind* helped me to confirm there wasn't any major alarming memory leak on Linux when I investigated the supposedly reported ones on macOS. *Renderdoc* was useful to troubleshoot graphics issues by deconstructing the render target into pieces after each consecutive API call and pipeline parametrisation. Helped me find the cause of the water shader issue. I also tried to use Nvidia's *Nsight* but the Linux and Vulkan support still were experimental and I couldn't successfully use it. "Linux users: Applications that launch child processes are not fully supported in Nsight Graphics. This will be improved in a future version." [13] I also improved my source control experience with *subversion*. Finally I also used several various tools, *HID calibrator* to troubleshoot wheel controller input, *DirectX Shader Compiler* to update the water HSLS shader, *texturePacker* to pack sprites into spritesheets for efficient use,

but also internal tools such as the build queuing system, and the numerous in-house scripts.

The Master's degree subject that was the most put into efficient use was definitely the C++ courses, as it is a daily tool in high performance real-time rendering applications such as video games, I can feel I greatly improved my understanding and skills in that topic, but there will always be room for further improvement. All the Computer Graphics related subjects, offline and real-time rendering, geometry, animation, were definitely making my work experience comfortable as I was able to follow rather technical discussions, understand their stakes and bring my own thoughts and knowledge even though I didn't get to specifically exercise them. The game design course also was a huge advantage to comprehend the work environment and position in a video game development studio. Eventually I came to discuss some audio library issues connecting the Audio and Video processing courses we had.

One of the achievements I keep in mind is coming with a procedure to confront any issues encountered: find a relevant entry point, build knowledge, understand cause, prepare and evaluate solutions, implement and finally optimise, making sure not to introduce any regression. This was proven useful all along the internship and allowed me to do about well in the most intricate situations. I could come up with other achievements thanks to this, which constituted all the major task aforementioned with a special mention to the main menu overhaul which accounted for most of my time here at the company. I came up with satisfying implementations matching the game design documents and do my part in this ambitious project.

On a more personal side, it was a thrilling adventure to come to London. I finally was part of the video game industry, working daily towards the realisation of a motivated vision I had to improve the Linux gaming ecosystem at my humble scale. I am grateful to all the Feral family that made this opportunity not even possible but also so welcoming and supporting. Moving to a new place came along with drastic lifestyle change, yet it allowed me some tourism from time to time. I also hope I could have improved my British language skills over the time. However, there were some downsides in the shape on tensions at the place I lived, fortunately sorted out by the end on the placement, but downsides also in the lack of organising and time I faced and was responsible for by the end of the placement affecting most people around me.

After this placement I will join back the team to help finish this project until its release and eventually for its post-release support lifetime. Hoping to get involved yet new interesting projects, I expressed my will to specialise in graphics development to undertake shader writing and debugging, rendering systems and graphics pipelines troubleshooting, meshes and animations related inquiries to help artists, and eventually in-house compatibility layer library Vulkan implementation maintenance. I really do wish to dedicate my future work to computer graphics development.

Bibliography

[1] Liam Dawe. (2019, March 28). DiRT 4 officially released for Linux, port from Feral Interactive. Retrieved April 2019, https://www.gamingonlinux.com/articles/dirt-4-officially-released-for-linux-port-from-feral-interactive.13844

[2] Liam Dawe. (2019, May 23). Total War: THREE KINGDOMS is out and it comes with same-day Linux support. Retrieved May 2019, https://www.gamingonlinux.com/articles/total-war-three-kingdoms-is-out-and-it-comes-with-same-day-linux-support.14190

[3] Josh "Cheese" (2018, September 11) Proton and Linux Gaming History. Retrieved May 2019, http://cheesetalks.net/proton-linux-gaming-history.php

[4] Apple (2019) Core Foundation. Retrieved May 2019, https://developer.apple.com/documentation/corefoundation

[5] Apple (2019) Core Graphics. Retrieved May 2019, https://developer.apple.com/documentation/coregraphics

[6] cppreference (2014, November 13) Storage class specifiers. Retrieved May 2019, https://en.cppreference.com/w/cpp/language/storage_duration

[7] Mark Szymczyk (2018, December 14) Measuring Your App's Memory Usage with Instruments. Retrieved May 2019, https://www.swiftdevjournal.com/measuring-your-apps-memory-usage-with-instruments/

[8] Microsoft (2017, February 8) Shader resource view (SRV) and Unordered Access view (UAV). Retrieved June 2019, https://docs.microsoft.com/en-us/windows/uwp/graphics-concepts/shader-resource-view--srv-

[9] Chuck Walbourn (2017, September 30) DirectX 11 What Is A Shader Resource View. Retrieved June 2019, https://stackoverflow.com/questions/46493782/directx-11-what-is-a-shader-resource-view

[10] Microsoft (2018, December 5) DXGI_FORMAT Enumeration. Retrieved July 2019, https://docs.microsoft.com/en-us/windows/win32/api/dxgiformat/ne-dxgiformat-dxgi_format

[11] Khronos Group (2019, August 25) VkFormat(3) Manual Page. Retrieved August 2019, https://www.khronos.org/registry/vulkan/specs/1.1-extensions/man/html/VkFormat.html

[12] cppreference (2012, November 28) new expression. Retrieved July 2019, https://en.cppreference.com/w/cpp/language/new

[13] Nvidia (2019, July 23) Nsight Graphics v2019.4 Release Notes. Retrieved August 2019, https://docs.nvidia.com/nsight-graphics/2019.4/ReleaseNotes/index.html#unique_382563166

Glossary

Splines: mathematically defined curves through functions, here it was used by DiRT4 to represent and trace the rally races tracks.

Ghost input: when an input is registered in the game while there are no hardware related event from the user.

To rasterize: to convert geometry primitives to actual pixels for rendering purposes.

Hot code: code which is very frequently called, likely to introduce overhead.

Shadow mapping: wide family of techniques to account for shadows computation in realtime rendering. From a light source point of a view, a depth map is computed. Further depth comparison from camera point of view can tell whether an area is lit.

Shader: GPU program meant to be run in parallel on hundreds of cores, the heart of rendering pipeline.

To Symbolize: symbols are extra embedded data on debug builds for error reports to show intelligible names rather than nonsense memory addresses. Symbolizing a build is the process of mapping the memory address to a names with a dedicated external symbol file as symbols can increase the size of binaries a lot.

Drawcall: query issued to the GPU along with buffers, often for rendering purposes.

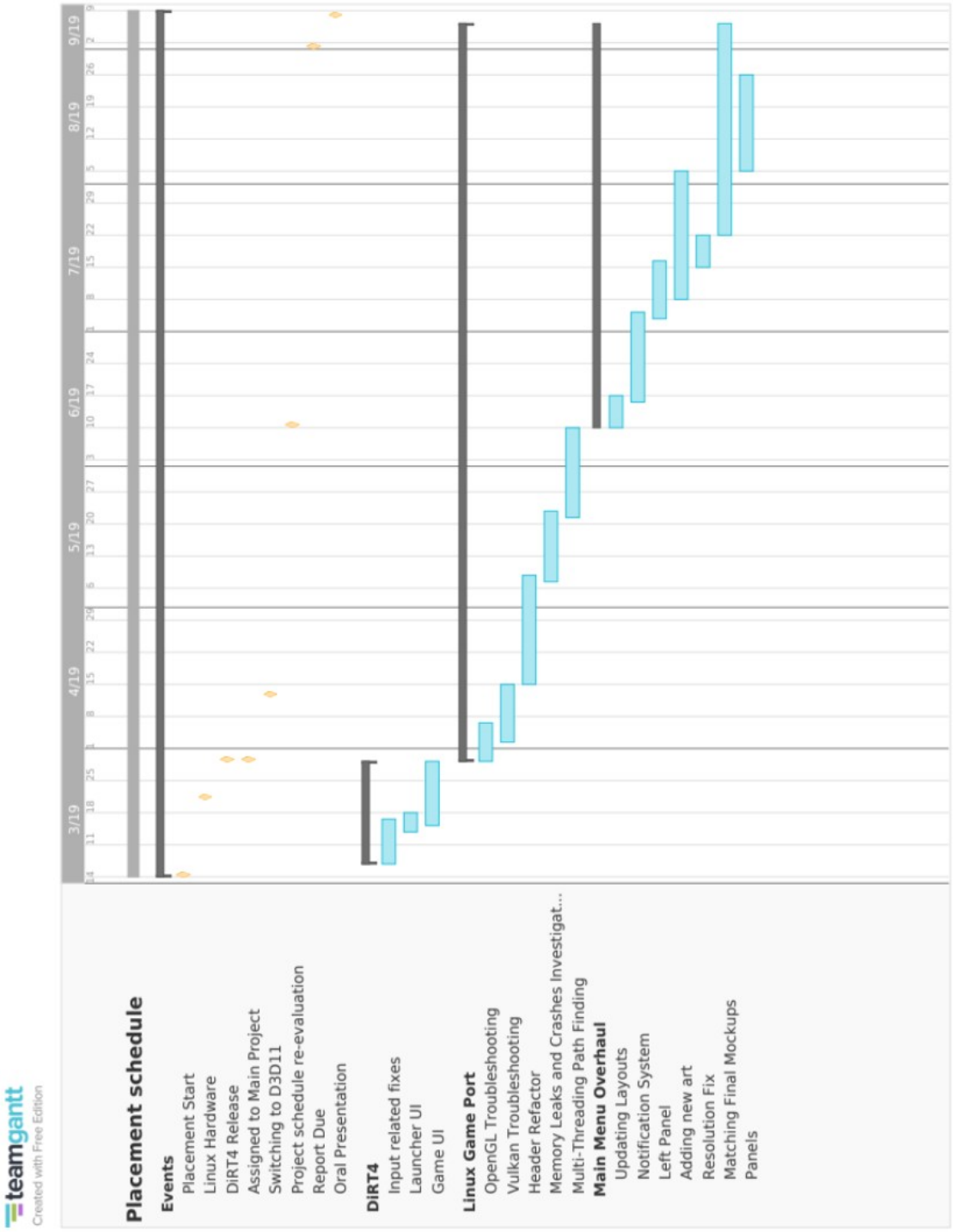
Culling: the act of discarding irrelevant part to final drawn images, speeding up rendering process.

Aspect ratio: ratio of the width by the height, used to define standard monitors dimensions along with the diagonal length in inches.

Acronym Table

API	Application Programming Interface
IGAI	Informatique Graphique et Analyse d'Images
STORM	Structural Models and Tools in Computer Graphics
MINDS	coMputational imagINg anD viSion
MSVC	Microsoft Visual C++
QA	Quality Assurance
DLC	Downloadable Content
UI	User Interface
XML	eXtended Markup Language
(USB-)HID	USB Human Interface Device class
UUID	Universally Unique Identifier
PGOW	PreGame Option Window
HTML	Hyper Text Markup Language
CSS	Cascading Style Sheet
SVG	Scalable Vector Graphics
PNG	Portable Network Graphics
LDAP	Lightweight Directory Access Protocol
IDE	Integrated Development Environment
CSM	Cascaded Shadow Mapping
GCC	GNU Compiler Collection
POD	Plain Old Data
GUI	Graphical User Interface
HLSL	High Level Shader Language
DXGI	DirectX Graphics Infrastructure
DDS	DirectDraw Surface
HDR	High Dynamic Range
FPS	Frame Per Second
LTO	Link-Time Optimisation
SIMD	Single Instruction on Multiple Data
SSE/AVX	Streaming SIMD Extensions/Advanced Vector Extensions
GDD	Game Design Document
TGA	Truevision Targa
ASAN	Address SANitizer
GDB	GNU Debugger

Appendix I – Placement Schedule



Appendix I – Events and tasks held during the internship

Appendix II – Pseudo-code of army selection icons positions calculation

```
// Find number of spaces
add_icons_number = max_players - player_number;
spaces_around_swords_icons = (alliances_number – 1) > 0 ? (alliances_number – 1) * 2 : 0;
// spaces_in_alliances is found with a loop through all alliances and
// if an alliance have at least one player, the number of player in alliance minus 1 is added
space_number = add_icons_number + spaces_around_swords_icons + spaces_in_alliances;

// Compute assets width
assets_width = add_icons_number * add_icon_width + player_number * army_icon_width +
(alliances_number -1) * sword_icon_width;
// Compute spaces width
space_width = (total_width – offset – assets_width) / space_number;

// Distribute add icons number to left and right side
left_add_icons_number = add_icons_number / 2;
right_add_icons_number = left_add_icons_number % 2;

// Init coordinates and clear vectors

// Add left icons in a loop from 0 to left_add_icons_number
// while incrementing x coordinate with the icon width + the space width

// Start counting alliance from 1
// Loop through alliances
        // Loop through players
                // if it's a valid player, add all corresponding sprites their respective vectors
                // increment x with army icon width + space width
        // check whether it's the last alliance and if it is not increment x with the sword icon
        width += space width
// Add right icons
```