h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

fbmd
FACHBEREICH MEDIA

(Faculty of Media)

Bachelor of Arts – Animation and Game

Development of a modern toolchain for 16-Bit homebrew games on Sega Mega Drive

Bachelor Thesis

Student: Hölzel, Luca | 756566

First supervisor: Prof. Dr. Martin Leissler

Second supervisor: Thomas Nickel

Date of Submission: 14.02.2022

TABLE OF CONTENTS

EXECUTIVE SUMMARY

Die Entwicklung von neuen Spielen für beliebte Computer- und Konsolenklassiker, als „Homebrew“ bezeichnet, war lange Zeit ein scheinbar gut gehütetes Geheimnis innerhalb kleiner Internetzirkel. Durch die zeitgenössische, pop-kulturelle Relevanz von Spielen aus den 8-Bit und 16-Bit Generationen der 80er und 90er Jahre kamen Spiele auf, die diesen Stil durch Pixel-Art und Chiptune Musik imitierten. Das Verlangen nach wahrer Retro-Authentizität, welches aus dieser Stilrichtung entstand, brachte schließlich neues Leben in diese verkapselten Internetzirkel, die sich mit der Entwicklung von wahrhaft authentischen Spielen für diese Platformen befassten.

Aus genau diesen Umständen wurden über das letzte Jahrzehnt neue Entwicklungs-möglichkeiten für diese Platformen erschlossen, durch welche moderne Entwickler erstmals mit Homebrew Spieleentwicklung in Berührung kamen. Eine dieser neuen Möglichkeiten ist das SGDK (Sega Genesis Development Kit), mit dem Spiele für die 16-Bit SEGA Mega Drive Konsole (1988) entwickelt werden können. Im Gegensatz zur klassischen Entwicklung für diese Platform in 68000 Assembly Sprache, kann hier eine komfortable Umgebung in der hohen C Sprache verwendet werden, was diese Art der Entwicklung für viele Interessierte Entwickler möglich macht.

Was sich allerdings auch bei all diesen Neuerungen nicht verändert hat ist der Fokus auf Entwicklungsumgebungen, die sich weiterhin an einzelne Bastler und Programmierer richten. Eine moderne, kollaborative und Netzwerk orientierte Toolchain zu erarbeiten, mit der ganze existierende Game-Teams in die gemeinsame Entwicklung für solche Platformen eintauchen können, hätte viel Potenzial. Womöglich könnte ein Schwall an hochqualitativen neuen Spielen für beispielsweise das SEGA Mega Drive, die Platform in einem neuen Licht erscheinen lassen.

Um eine solche moderne Toolchain am Beispiel des SEGA Mega Drives zu erarbeiten wird ein lokal gehostetes Server Konzept ausgearbeitet und implementiert, welches Zugang zu einem Version-Control Repository, einer Wiki Seite mit Entwicklungsinformationen und einem Zeitmanagement Tool bietet. Zudem wird die Kompilierung des aktiv Entwickelten Projekts auf Serverseite ermöglicht. Dieser Server wird auf einem erschwinglichen Raspberry Pi Computer ausgeführt.

Weiterhin werden verschiedene Programme für alle nötigen Disziplinen eines vollwertigen Game-Teams evaluiert, um eine Kompetente Reihe an anzuwendender Software in die Toolchain einarbeiten zu können. Die Bereitstellung dieser Software Programme und das Aufsetzten des Serverzugangs, um erfolgreich kollaborieren zu können, wird durch Entwicklung einer Toolbox Applikation vereinfacht. Diese Applikation ermöglicht das Wechseln zwischen Team-Rollen und Zugriff auf das Version-Control Repository mittels weniger Klicks, um die Konfigurationszeit eines interessierten Teams möglichst gering zu halten.

Um die angeführte Implementation dieser gesamten Toolchain auf die Probe zu stellen und eine Auswertung der Funktionalität zu ermöglichen, wird eine „Game-Jam“ Veranstaltung abgehalten. Durch Verwendung der Toolchain wird innerhalb von 12 Stunden darin, mit insgesamt 7 weiteren Personen, ein Spiel für die SEGA Mega Drive Konsole entwickelt, welches später auf der echten Konsole ausgeführt wird. Die Erfahrungen der Teilnehmer werden durch Fragebögen erfasst und evaluiert.

Die Stärken, sowie die Schwächen dieses Entwicklungsprozesses, insbesondere des Server Models und der angewandten Software werden ebenfalls dargelegt. So wird beispielsweise die Produktion von Musik für die Platform als stärkere Herausforderung befunden, als die Arbeit mit visuellen Elementen. Letztlich werden mögliche Verbesserungen und das große Potenzial dieser Art der Entwicklung im Kontext eines informatischen Kurrikulums hervorgehoben.

1. ─ INTRODUCING THE TOPIC

Over the last decades, the importance and relevance held by technological devices, as well as the various forms of media consumed through them, has grown exponentially. What once merely were simple distractions from life have become a large part of our generational consciousness and even inform our identities and nostalgic memories. The path which both our relationship to technology and to the numerous art forms utilizing it have taken over this time frame was an equally challenging one. Perhaps the most noteworthy form of media in this regard has been video games. From a simple toy to be bought out of novelty, to the highest earning and most universally enjoyed form of media on the planet, video games represent not only a monumental success of a young medium, but also a notable disparaging point between different generations of video game players.

Many astute publications, faculties and individuals alike have written at length about just this phenomenon, detailing the various facets of progress witnessed by the gaming landscape. This includes the rise of casual gaming, the emergence and importance of e-sports and the significance the early internet played in forming communities around video games *(Alex Hope, 2013).* Countless other important examples in the incremental journey of video game prevalence can and have been brought to table of discussion. However, this form of admirable dwelling on such a journey often fails to shine light on one of the most crucial parts: The development processes used to create video games in the first place. Just as the expectation towards video games and our understanding of them has changed drastically, the development processes, tools, team-based paradigms and industry standards have evolved accordingly, if not even faster *(Brent Cowan; Bill Kapralos, 2014)*.

Curiously, the process of the gaming landscape coming to grips with its newfound relevance coincided with the sudden popularity of retro-gaming and a wave of retro-revival games, popularizing the use of pixel-art designs and chiptune music *(Thomas Bryant David, 2017)*. While first thought to be simply a fad, trying to profit off of the nostalgia for a bygone era of the medium, the persistent popularity of this audio-visual style has led to its acceptance as a mainstay in the gaming industry. Even the highest calibers of video game studios have since embraced the style, once relegated to smaller independent studios and niche projects.

Solidifying this position, many young gamers who never experienced games from the respective time periods still find themselves enjoying the simpler visuals of retro-themed game releases *(Jack England, 2021)*, proving the continual relevance and marketability of the style.

1.1 ─ Current Problems

A large factor in the previously described adoration for retro-themed games, even among younger gamers, is reveling in the authenticity provided by this style. Just as cinema, music and even fashion often reinvent past styles and present them in a new light, the same yearning for the authenticity of cultural elements one has not experienced personally *(Andrew Higson, 2013)*, but grew up knowing of, applies to video games. It is this exact element of authenticity that forms one of the key problems around the retro-game style and its course.

Using pixel-art designs and chiptune music to essentially develop a modern game, intended to be played on modern devices, can only go so far in terms of authenticity. A modern mobile phone game, played in the vertical portrait-mode, designed around touch controls and leveraging modern game design philosophy, including for instance micro-transactions, will not be viewed as authentic in any way, even if a retro-style is chosen for its assets. This dissatisfaction with the inconsistency in retro-themed video games has given rise to an unlikely pool of developers: The “homebrew” community.

In short, homebrew development and the community that has formed around it concerns itself with the development of new games, software or graphical demo programs for old systems of the past *(Melanie Swalwell, 2021)*. Instead of recreating the look, sound or feeling of a certain era of video games, these developers instead seek to develop directly for the systems of the respective era. This radical difference provides perhaps the greatest source of retro authenticity possible, by working with the exact limitations and restrictions of specific systems, that often shaped the nature of real games released.

While being a solution to the problem of authenticity, the true problem lies within overcoming the development challenges inherent to old, restrictive target platforms. The ease

of use provided by modern game engines and the amazing power of modern processors and graphics chips allow for very streamlined, iterative development, not easily possible on decades old hardware. The previously noted advancements in team-based development paradigms equally become a challenge to implement in relation to such target platforms. As such, overcoming the previous need for old, clunky development kits *(Retro Reversing, 2020)* and restrictive assembly language, all in a modern team-based context, certainly presents a large problem in the field of homebrew development.

1.2 ─ Incentive for solutions

Ultimately the benefit of establishing such a described toolchain lies in lowering the barrier to entry for team-based homebrew development. This would allow existing teams of developers to quickly begin development of homebrew projects without spending much time researching a cohesive method for collaboration, or evaluating individual tools to be used. The challenging nature of development for limited systems holds great educational potential as well, as decisions need to be well thought out and structured before jumping into implementation. Comparatively, the ease of development offered by the typical modern game engines often encourages quick, unstructured bursts of development *(Neurosys, 2020)*, failing to educate on resource sensitive development practices. These missteps are often only realized too late, for instance when porting the game to a less powerful platform, resulting in much additional time spent rewriting existing game systems.

The desire to work within such restrictions to sharpen ones skills is made quite clear with the popularity of fantasy consoles, such as the PICO-8 *(Lexaloffle Games Ltd, 2015)*. While designed to be developed with modern languages in mind, these restrictive hypothetical systems allow for problem solving to become the main focus of development, much like homebrew development for real retro systems.

Whichever way one approaches the topic, the ever-increasing popularity of authentic homebrew games and modifications of classic games from the time period, known as “rom-hacks”, facilitates a demand for an approachable, modern team-based development toolchain, opening previously locked doors for many creative teams.

2. ─ CURRENT STATE OF THE ART

Before beginning to hypothesize any solution to such a multifaceted topic as homebrew game development, it is vital to familiarize oneself with the current standards available and the course both the craft itself and its community have taken over the last decades. Gaining an understanding of the origin of the homebrew designation, the influence this style of development has had and which innovators and projects propelled homebrew development to where it is now, will help to guide ones perception and approach for the topic. The characteristics of both the cultural and technical aspects of this community, as well as the strides made in the direction of team-based homebrew development work will be explored in the following sections.

2.1 ─ Brief history of homebrew development

To begin discussing the modern context of homebrew development, it is important to first shine a light onto the nature of the homebrew community and even the very origin of the term "homebrew" itself. The vernacularly defined origin of the term can be obviously deducted from its components "home" and "brew", referring to the act of brewing alcoholic beverages domestically in small quantities. While the legal implications of this action do not typically reflect on the "homebrew" abstraction discussed in context of game development, some of the frugalities that come with a self-taught approach to a highly technical activity do hold true to the topic in a metaphorical sense, which will be examined further along in this chapter.

Beyond the historical word origin, perhaps the most pivotal use of the term was in the titular "Homebrew Computer Club", founded in the San Francisco bay area in 1975 *(Ashley Haugen, 2021)*. This legendary congregation of early computer enthusiasts would, as popularly known by this point, see the early stages of the "Apple I" computer being discussed and built by Steve Wozniak and Steve Jobs of later Apple Inc. fame. Many other members would equally go on to leave large impressions on the early personal computer industry. Within this club the term "homebrew" referred to both hardware and software created by individuals to be presented and discussed with other members.

From the discussions and advancements of groups, clubs and corporations alike the late 1970s and early 1980s saw the rise of the early commercial home computer, typically referred to as 8-Bit "micro computers" *(Computer History Museum, 2020)* as to be distinguished from the larger (cabinet sized) "mini computers" and the even larger mainframe computers employed in many government bodies and institutions by this point. During this early phase, one could make the argument that development was simply the act of using a computer, as most of the earliest models required basic programs or even machine code to be input directly into RAM, for any desired program, before the advent of removable floppy disk or tape storage for program loading. The colliding of highly technical requirements for the operation of early computers, with younger individuals apt and eager to learn forged the basis of what would later become the modern definition of the term homebrew.

The homebrew definition emerging from this newfound technological prevalence in society is perhaps best categorized and defined in the previously cited book "Homebrew Gaming and the Beginnings of Vernacular Digitally" (2021) by Melanie Swalwell. This definition employs four main parameters to be met for the classification. (1) The development takes place in a domestic setting, not a corporate or otherwise institutional context. (2) The project is developed by a very small group or a single individual. (3) The development skills employed for creation are self-taught, as a result of the clashing between new cutting edge technology of the time, with the uninformed public having to learn everything themselves, before computer focused curriculums were introduced. (4) No large scale publishing of the resulting game or software, rather spreading through word of mouth and personal floppy copies. While this definition certainly characterizes the circumstances of the time period well, it struggles to account for the influences present in a modern context. The aspects of domestic development and self-taught skillset faulter under the modern relevance of game development in an educational context, while the aspect of publishing does not apply to the possibilities afforded to software distribution on the modern internet. Nonetheless the provided and well justified definition presented by Melanie Swalwell informs the still present spirit of the homebrew community to this day, making it important to understand this origin point.

2.2 ─ Major innovations, important projects for SEGA Mega Drive®

In order to form a cohesive understanding of the homebrew concept and its origin, a general approach to the topic, its background and path to the present has been laid out in the previous sections. Going into specifics on such a topic as homebrew game development necessitates more technical specialization however. As described previously, this thesis and the related research paper cover the development of a toolchain and the research into programming performance on the 16-Bit Mega Drive console, initially released in late 1988 by SEGA Ltd. The state of development tools and methodologies varies wildly from system to system, making any further technical explanation difficult in general terms. As such all presented information from this point forward relates in particular to the SEGA Mega Drive console. It should be noted however that general information about limitations and restrictions of old hardware can still be useful in a broader context and are possibly applicable to other systems of the same time period.

One common misconception about homebrew games development is its perceived relation to the retro-gaming boom of the 2010s. These are in fact intertwined, influencing one another, but homebrew development reaches back a surprisingly long way, before any kind of nostalgia would have prompted development of a game. For the SEGA Mega Drive for instance, the earliest noted homebrew projects emerged as far back as 1991 with “Super Ping-Pong” by developer “Pure-Byte” *(see source of figure below)*. Much like many early homebrew games, the individuals or teams creating them did so mainly as a technical challenge. The above mentioned game follows this trajectory, as it simply implements a PONG style game, with a few added features, which was still impressive considering there were no tools or publicly available documentation to facilitate development at that time. Other similar projects emerged throughout the later 1990s and early 2000s, most of which equally implementing an existing arcade game or other simple game concept. (“SegGala” - Galaga clone, Bill Eubanks 1998; “Ultimate Tetris” - Tetris clone, Haroldo OK! 2001). The purely technical motivation behind these projects was clearly apparent, with many following Mega Drive homebrew projects being implemented for demo-scene contests, competing in clever programming on limited hardware. (“Space Pixy”, Sarang 2010; “Overdrive”, Titan 2013).

Original games

1990 - 2009

#	Game	Developer	Year	Genre	External links	Cartridge version	Digital version	Note
1	*Super Ping-Pong*	Pure-Byte	1991	Pong	Information Download	None	Free Download	A simple pong game.
2	*SegGala*	Bill Eubanks	1998	Arcade Shooter Game	Information Download	None	Free Download	A Galaga clone.
3	*Ultimate Tetris*	Haroldo OK!	2001	Puzzle	Website	None	Free Download	A Tetris clone.
4	*Frog Feast*	RasterSoft	March 31, 2006	Arcade	Website	None	Demo (File lost)	A tribute to the classic game "Frog Bog".
5	*Tubes* (V. 0.06)	Bruce Tomlin	February 6, 2007	Puzzle	Discussion Article	None	Free Download	Make a completely capped group of pipes to score.
6	*Tetrex*	Mairtrus	Aug 10, 2008	Puzzle	Discussion	None	Free Download	A tetris clone.
7	*Airtstriker*	Electrokinesis Studios	November 30, 2008	Vertical Shooter	Website	None	Free Download	Airstriker is a 1980's-like arcade vertical shooter with 2 player support and an endurance mode.
8	*260bSnake*	Sonic 65	September 12, 2009	Arcade	Discussion Download	None	Free Download	A Snake clone in 260 bytes.
								A small game of

Fig. 01: Listing of early homebrew games for SEGA Mega Drive
(source: https://segaretro.org/List_of_Sega_Mega_Drive_homebrew_games)

Perhaps the greatest breakthrough in the popularity of homebrew games for the SEGA Mega Drive came with the widely acclaimed “Tanglewood”, developed and released by Matt Phillips *(Big Evil Corporation)* in 2018. This project, covered in slightly more detail as part of the related research paper as well, was developed using a period correct SEGA Mega Drive development kit and Windows 95 computer for debugger attachment. This true to the time period development style raised interest from many developers across the internet, prompting

an appearance of Matt Phillip on the “Computerphile” video web series *(Matt Phillips| | Computerfile, 2020)*, explaining the development kit and the assembly language used to program the Mega Drives main processor.

Fig. 02: Tanglewood game physical release

Fig. 03: Tanglewood developer Matt Phillips, pictured with original dev kit

So far, all the mentioned projects and individuals approached the development of games for the Mega Drive system the same is it would have been during its period in active development, using the aforementioned low-level assembly language. The difficulties, differences and hinderances regarding this approach are illuminated in detail as one part of the related research paper as well. The other part of this paper regards similar questions about the new higher-level language frameworks which have sprung up over the last decade. In particular, the largest break from the previous assembly focused efforts in Mega Drive homebrew development, is represented by the acclaimed SGDK (Sega Genesis Development Kit) framework by Stephane Dallongeville. In short, this C language based programming framework implements an API, containing abstracted functions for all relevant tasks to be programmed on the SEGA Mega Drive. Instead of working with memory addresses and direct hardware operations, comfortable and easily understandable function calls can be used to program new games for the Mega Drive console, using SGDK. Another vital part of this development kit is the resource compiler component, allowing for modern file formats to be

automatically converted into binary formats usable by the system, forgoing the previous need for custom asset tools to be programmed for each game.

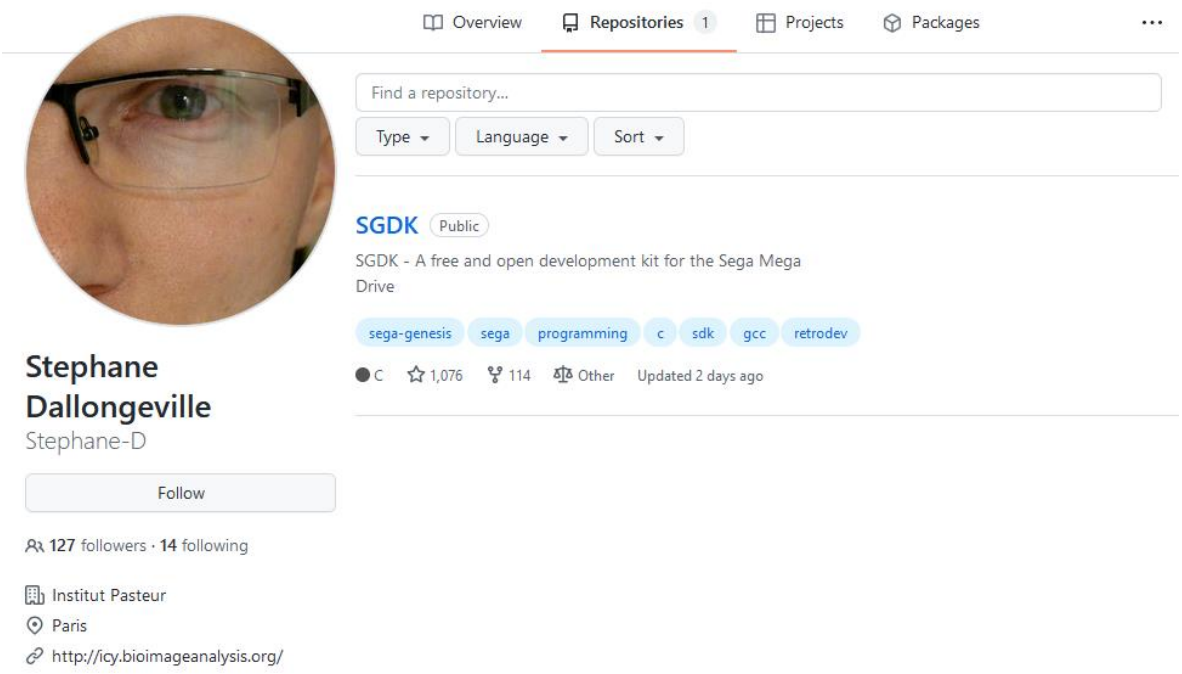


Fig. 04: Stephane Dallongeville GitHub, SGDK

This style of high-level framework has emerged for many consoles of the time period to allow more people entry into the development for these beloved systems. To facilitate more integrated development with these new tools, ports to different operating systems like GenDev *(Kubilus1, 2015)* and MarsDev *(Andre DeRosier, 2017)*, integration scripts for various design and asset tools and plugins for various text editors and development environments *(Zerasul, 2020)* have been created by the homebrew community as well.

As it stands, the high-level programming for the SEGA Mega Drive afforded by SGDK and the related communal efforts represent a breakthrough in the ease of development for the system, prompting the presently conducted work to adapt it into a usable, team-focused toolchain, to propel these efforts into more peoples minds and hands.

2.3 – Current state of team-based development for SEGA Mega Drive®

As described, the initial approach to homebrew development on the SEGA Mega Drive employed the same low-level assembly Motorola 68000 language *(TigerNT, 2009)* used to develop for the system originally. Besides the difficulties in learning this approach from a modern perspective, the collaboration with a team of other disciplines is equally negatively impacted. When creating a game from the ground up in assembly language, many types pre-defined in higher-level frameworks, such as SGDK, have to be implemented manually. Besides basic representation of data for graphics, most other storage formats for levels, audio and more will be custom to the implementation, necessitating custom graphics tools, level tools and audio tools to be implemented for the project. For most tasks, creation of plugins for existing programs will suffice, but depending on the specifics of the game this can wildly differ. This results in an antiquated team constellation, in which programmers more often than not have to act as integrators of the created assets, on top of their development tasks. This was indeed common place during the era of the Mega Drive system, but is simply not a feasible team-based paradigm to be carried forward into the modern day of homebrew development for the system.

The emergence of higher-level C programming for the Mega Drive via the mentioned SGDK framework has certainly stirred up the preconceived notions about a team-based development process for the system. While most developers leveraging these innovations still tend to be single individuals working on their own homebrew projects, the SGDK and its ports do offer new possibilities to structure the development within a team much closer to a modern paradigm. Specifically the clear folder and file structure native to SGDK (see research paper) and the flexibility of the resource compiler for different kinds of assets should theoretically allow for compartmentalization of tasks over a network. These possibilities seem to not have been made use of yet however, as there is no documentation or field report about such an undertaking available on the internet at the time of writing, prompting a comprehensive approach to be developed as part of this thesis.

3. ─ PROPOSITION OF WORK TO BE CONDUCTED

Now that the intricacies of homebrew development, its history and current progression have been thoroughly laid out, it is time to constructively devise a plan of action to tackle the discusses problem. Firstly, the requirements to facilitate the desired complexity of a team-based development approach will be made clear, functioning as a reference point for wanted features and needed functionality throughout the later implementation. Based on these requirements, the individual elements and in turn their requirements will be presented and visualized, to put emphasis on keeping a clear structure throughout later chapters. Lastly a method to test the anticipated results for real-world usage needs to be defined, from which an assessment of the applicability of the resulting work can be conducted as part of the later chapters of this thesis.

3.1 ─ Requirements

Before jumping into ideas for the possible execution of specific tasks, it is important to first set a baseline of requirements that the proposed toolchain needs to adhere to going forward. For modern team-based development multiple aspects of the workflow, and necessities to facilitate the workflow in the first place, have to be observed and taken into account. To clear the way for collaboration over the network, the need for some type of server to manage the project becomes clearly apparent. Depending on the desired level of professionalism, multiple avenues could be evaluated for use in this scenario. Since the question to be answered is whether a development process akin to the current day standards is feasible for homebrew development on the SEGA Mega Drive, a comparable approach with modern version control software is to be used. Furthermore, the server-model should allow for individual work of the different disciplines to be integrated asynchronously to avoid file conflicts, as is standard nowadays. Another important step towards a modern, network-centric development cycle for such homebrew projects is build-automation. By devising a setup that allows the server, not just local development machines, to compile and distribute the current project incrementally, more rapid testing of the project could be enabled.

Beyond just the handling of file upload, version control and compilation, there are other aspects that should be implemented on the server-side to allow for a more streamlined development and learning process. For instance, a common burden of small teams, especially in non-standard projects such as homebrew development, is the many emerging questions of team-members on tools and development processes to be used. These questions very often end up being answered by the programmers, leading to an overhead of development and explanatory tasks. To remedy this problem at least to a point, all necessary information about the workflow of each discipline and the tools to be used should be made available to the team. There are multiple possibilities for how the information could be shared, but a common and useful one is the construction of a wiki page, hosted on the same server as above.

With the development processes and equally the ability to study these processes now covered, there is one last server-side task to be considered. As one should be familiar with in the realm of collaborative development work, time management is equally crucial to a structured, modern development cycle. Without a firm grip on the time taken to develop a feature or create an asset, the available time to finish a project can quickly slip away at the hands of an unstructured paradigm. To allow for time management to become part of the proposed workflow, a time management system needs to be deployed. For this, the same server should also host a simple time management system, preferably accessible via a browser to ease access for all members of the team.

With the necessary elements of the server-side now clear, it is time to turn to the other side. What is required or even desirable for the client machines, to be connected to the proposed server, is less apparent at first. After pondering the required tasks on the client-side, two main elements emerge. First, the tools required by each individual discipline need to be distributed and configured correctly on the client machines and secondly, access to the server-side repository and the credential setup that comes with this needs to be taken care of smoothly. To simplify these elements and make them accessible to all team-members, a toolbox application should be developed and installed on all development machines. Operation of this application should be self-explanatory and require very little prior knowledge with the server-side setup itself.

Specifics on the desired execution of the hereby established requirements, as well as the evaluation to be conducted for the resulting implementation, will be described in the next sections of this chapter.

3.2 – Basic visualization of intended client-server toolchain

After more conducted research into the execution of the previously proposed requirements for both the server-side and client-side implementation, the path forward for how to execute these desired elements became clearer. Visualizing the relationship between the abstract top-level elements present on the client-side and server-side respectively should give a much clearer understanding for how these two sides are intended to interact.

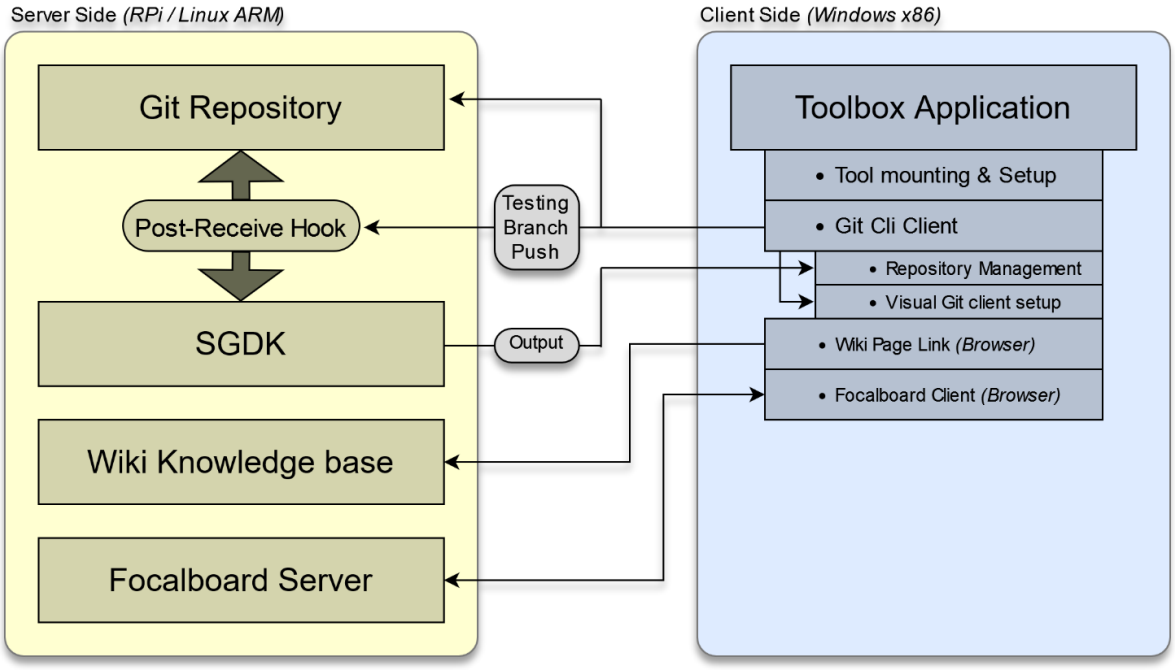


Fig. 05: Toolchain visualization, operating between client and server

The above diagram shows the simplified interaction of the client-side toolbox application and its components with the individual server-side components hosted on the raspberry pi server. Of note is the relation between the local git client and the repository / SGDK installation on the server. More thorough explanation, also focusing on the specific tools

chosen to implement the visualized elements, will be presented in the next chapter, as part of the tool designation.

3.3 ─ Development server for network-centric team model

It is indeed rather difficult to discuss the desired structure or setup of something as specific and technical as a server without possibly defining parts of the implementation to a degree. While all of these technical details will be reiterated and explained in the next chapter, it is important to define a few key specifics to explain the approach for the setup of the development server.

As a whole, the implementation of the above described development server should be conducted in a general-purpose manner, increasing the adaptability to more specialized deployment circumstances later. To allow for easily interchangeable components and transparently configured elements on the server, a reliance on free open-source software (FOSS) is key. The advantage of this kind of software is the ubiquity that comes with open code *(Steven Weber, 2004)*, usually being available completely free for any desired hardware. Speaking of hardware, the same approach to accessible deployment and recreation of the desired server extends to hardware selection as well. With how simple games for the SEGA Mega Drive are, the machine running the server elements is not required to be very fast or have vast amounts of storage. As such, using server hardware that is very accessible to interested parties both monetarily and topically becomes a clear goal. The chosen hardware for the implementation of the server to be conducted is a "Raspberry Pi 4" (Model B) *(Raspberry Pi Foundation, 2022)*. This single-board computer (SBC) is a widely popular choice for small server projects and other interactive projects across networks. More technical information about this choice will be presented as part of the implementation.

To chronologically cover the previously mentioned parts of the server setup, the collaboration and file sharing needs to be addressed first. With the simplicity of code bases for SEGA Mega Drive games, a few simplified options for collaboration over the network do exist. Using a simple local file or folder sharing option would most likely work well enough, but as team-sizes increase this approach will presumably become less reliable, as more people need to

access the same files at once. Keeping both with the requirement of modern team-based solutions and the reliance on FOSS software, the ubiquitous “Git” version control software was chosen, as it is the most widely used version control software available and comes standard with the debian linux based “RaspianOS” for the chosen raspberry pi computer. The collaboration requirements above also listed the use of server-side build-automation. With the overall simplicity of the build process using the mentioned SGDK framework, parts of the git version control setup will be configured to allow for automated building. The necessary configuration and specifics for a git sever setup with SEGA Mega Drive development in mind will be covered in the implementation chapter.

Upholding the requirement of an adaptable FOSS workflow for server-side tasks, all other described elements should equally consist of FOSS software running locally on the raspberry pi server. This is especially applicable to the mentioned wiki website to be hosted via the server. The free and unencumbered access to information is perhaps the most vital part of the internet, as such many FOSS solutions to host information in wiki-page format have emerged over the years. Considering the relative power of the raspberry pi, the most modern wiki engines may not be the choice, as the system requirements target equally more modern systems in terms of power. The wiki engine of choice here is the venerable “Mediawiki” *(https://www.mediawiki.org/wiki/MediaWiki)*. Being originally developed in 2002 for the modern mother of all wiki sites, Wikipedia, this wiki engine clearly should have all necessary features needed as well as, considering its age, a modest system footprint. The contents of the wiki will consist of formatted versions of all the thorough notes and documentation taken during the learning process for this thesis and the related research paper. By offering all the knowledge gathered during this process to a team, the amount of verbal questions is possibly reduced. This first iteration of the wiki might not be as effective as possible, seeing as it consists of formatted notes and not formally written guides for each discipline, but it should still be a decent help and leave room for future improvements.

The tool of choice for time management was picked by the same criteria of FOSS software and ability to locally host and adapt for future iteration. The tool settled on was “Focalboard” *(Mattermost Inc, 2021)*, a standalone project by the “Mattermost” (Inc) team, previously known for their equally FOSS online chat service of the same name. Being originally

developed as an in-house chat tool for the “SpinPunch” game development studio, the mattermost code was open-sourced in 2015 and the team assigned to it has added functionality ever since, including the mentioned focalboard software. While mattermost is a FOSS alternative to the likes of “Slack” and “Teams”, focalboard aims to provide the same FOSS implementation as a competitor for “Trello” *(Atlassian Corporation, 2011)*. Trello in particular is a very popular choice for simple task and time tracking, which was also used during many semester projects as part of the Animation & Game degree. Having the ability to self-host a comparative tool should help make completely locally contained time management possible for the project.

After all the configuration and implementation discussed in the following chapter are conducted, the resulting working raspberry pi server will be archived. To achieve this, the micro-SD card containing the operating system and all implemented server components will be imaged into a single image file. This image file can then be shared and written to another empty micro-SD card for redeployment in multiple development sessions and scenarios. While all used server elements are open-source, making sure the rights of each individual server component are reserved is still important. A detailed description of how to setup the server should be made public, containing references to all used software and repositories, to avoid possible problems with publicly sharing only the final server image.

3.4 ─ Toolbox program to manage tool and server access for all disciplines

With the intended setup for the server now being largely defined for later implementation, the focus should shift to the local machines that will be used by the team-members for their work to be contributed to the given project. As previously mentioned, each machine needs to be fitted with the correct software to be used by each discipline and needs to be setup for connection to the version control server. Quick reference URL shortcuts to the described wiki and time management tool should equally be present for ease-of-access. The tool selection and more specific details are discussed in the coming implementation chapter. Beyond the choice of tools however, it is how they are distributed and managed on the client machines that affects consistency and replicability in future development sessions. Setting up all

development environments for each discipline manually on each machine is very time consuming and can lead to inaccuracies with desired program versions or folder paths. The access to the git version control repository in particular shows another weakness of this simple approach, as most team-members will likely have little understanding of the software and would need help setting up the repository themselves.

To remedy these issues, a toolbox application automating the process of program distribution and repository setup should be developed. The requirements for such a toolbox are not immediately apparent, as this type of application is not very standardized and highly dependent on its context. Firstly, the automatic assigning and mounting of needed programs for each discipline should be implemented, triggered by a simple button press or other easily understandable user interface element. This aspect should also be quickly reversible, allowing for the same machine to be shared between team-members by quickly mounting a different set of tools to be used. Simplified entry of the necessary credentials for the version control server should be provided in the program, then setting up the repository with only the required elements presented for the actively selected discipline. These simple abstractions via a friendlier user interface should streamline the setup of all required development environments for the respective disciplines.

The program itself is to be implemented for computers running the “Windows” series of operating systems, with versions from “Windows 7” to “Windows 11” as targets to be supported. The overall structure and implementation will be discussed in the coming chapter, but another important requirement to be noted for the program should be the externalization of its actions to script files. Allowing the editing of functionality outside of the compiled program will aid to make the program more adaptable to future changes, without needing recompilation. Details about this process will follow in the appropriate chapter / section.

4. — IMPLEMENTATION OF PROPOSED WORK

Before jumping into the following specific subsections on the establishment and implementation of the toolchain, it is worth noting how the term “toolchain” is used in the general context of the implementation. Firstly, it is typically used as an umbrella term for a multitude of interconnected and causally dependent tasks, executed in a linear fashion. The term can refer to entirely programming and compilation related tasks, in which one programs output is immediately used as another program or commands input, forming an immediately traveled chain of tasks to result in a single desired output at the end. In the context of collaborative development approaches, such as the one presently discussed for the SEGA Mega Drive console, the term toolchain is also used, referring instead to a chain of both end-user tools and compilation tools with which to create a single project. In this manner, this definition of toolchain is also often used synonymously with “pipeline”, equally deploying a metaphor of individual streams of information meeting and forming a cohesive result at the end.

The presented streams of information flowing between user tools, server components and the resulting compilation process, is broken up into its individual pieces, each explained and later visualized in the following sections.

4.1 — Choice of tools, frameworks & environments

To begin implementing the individual elements of the described toolchain, it is important to first define all chosen elements themselves and the reasoning behind their selection. Alternatives that could be considered in place of the finally chosen elements should be named to broaden the understanding of available tools and material. Different development scenarios might very well benefit from a different selection or constellation of tools, as such both reasons for and against the respective choice need to be discussed.

4.1.1 — Operating systems for server & client machines

Starting out with the choice of operating system to be used for the sake of development, both the noted client-side development machines and the server these machines will interface with

need to be outfitted with a suitable operating system. Since the server machine chosen is a raspberry pi 4 model b, an ARM architecture based single board computer, the choice of operating system is rather obvious. The debian linux based “RaspianOS” designed specifically for the machine comes with great stability and a large repository of industry standard networking and compilation tools to boot. While other linux distributions like “Ubuntu Linux” and “Arch Linux” (among others) and even variants of the BSD (Berkley Software Distribution) line of unix-like operating systems are equally valid choices, sticking with the thoroughly documented and supported “RaspianOS” typically avoids unforeseen problems in the long run. Which software packages are used and how they are configured on the server-side is covered in the following sections.

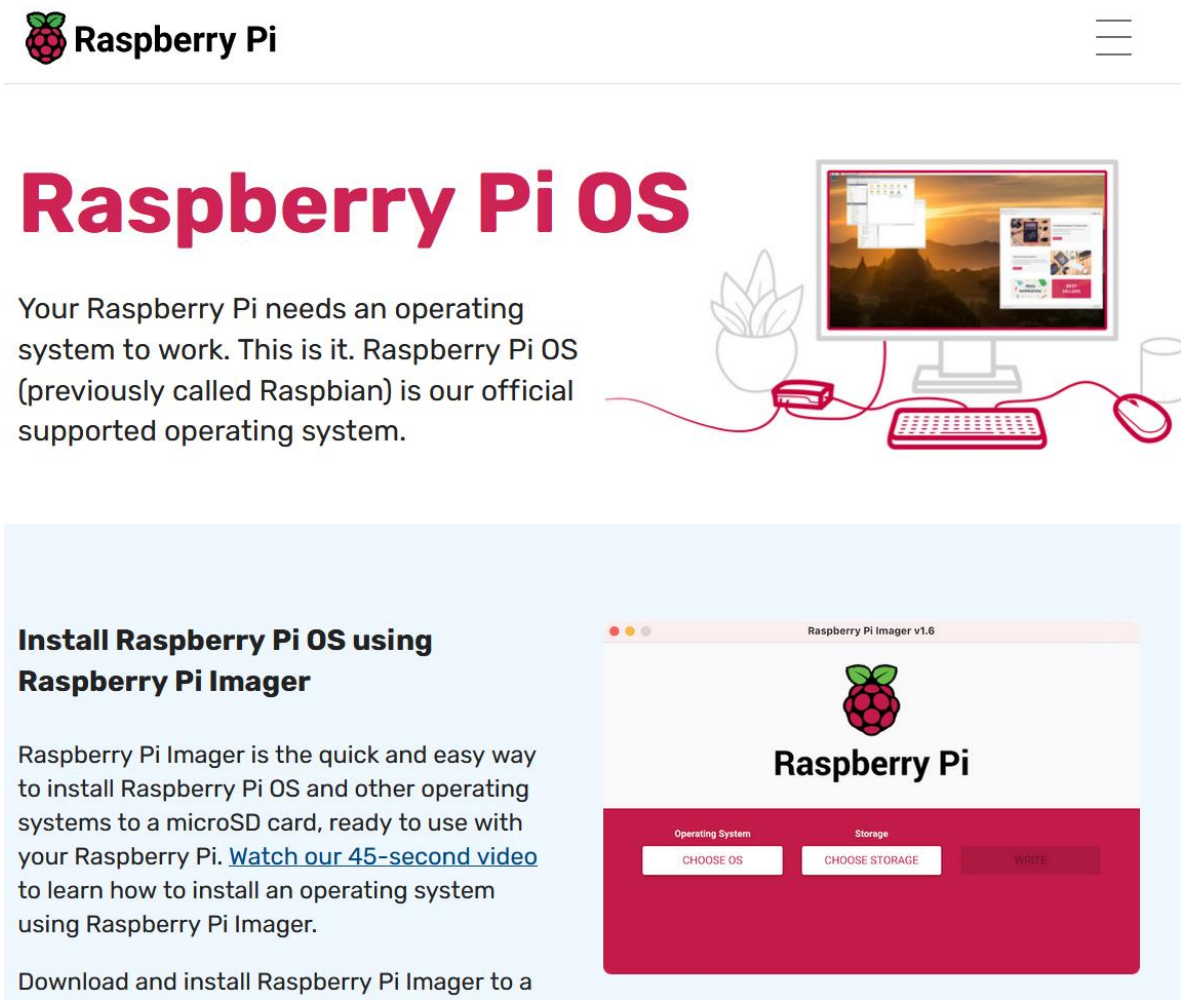


Fig. 06: Raspberry Pi OS download page

On the client-side a whole different set of requirements and restrictions do apply. The hardware chosen here obviously plays an equal role but could be much more diverse depending on deployment circumstances. While the linux operating system for use on personal computers has come an amazingly long way over the years, especially for development tasks, the compatibility and software selection does still often lack, not to mention the historic problems between linux and gaming related software. Additionally, the user interfaces shipped with various distributions of the operating system can vary wildly, which might be interesting to certain users, but could potentially be hard to get used to for less tech-savvy team-members. In the end, both for the just described reasons and the shier ubiquity of its largest closed-source competitor, the linux opertating system was reserved for use on the server only, while the development machines are to run a Microsoft Windows operating system. Which version chosen should depend entirely on the hardware available. Lower-end machines should be outfitted with Windows 7, for its resource-sensitivity and still modern enough feature set, while more modern machines should receive an installation of Windows 10 or Windows 11.

4.1.2 — SGDK framework & alternatives

Now that the environments to be present on all machines are cleared up, the method of developing and compiling games for the SEGA Mega Drive console within these environments needs to come into focus. Naturally, the backbone of development in this case is the chosen framework to facilitate development for such a specific target platform. The logical choice for a modern development framework here is the previously mentioned SGDK (Sega Genesis Development Kit) or one of its supporting forks, such as GenDev or MarsDev. Not only did the sudden emergence and quality of this framework prompt the recent wave of new Mega Drive homebrew projects, but it provides the needed balance between performance, capability and adaptability to team-centric work through its resource compiler. This is not to say it is truly the only option to be considered for every project moving forward. The so called Second Basic Studio *(Second Dimension, 2022)* allows for SEGA Mega Drive development using a custom implementation of a BASIC style language, reminiscent of the simple programming interfaces for early home computers. Additionally Second Basic comes with its very own integrated development environment, fir to structure development tasks

adequately. While possibly a great choice for a programming beginner, the reliance on an interpreted language, which in turn gets compiled down to executable code leaves deep optimizations on the table, which become rather important on limited hardware. It still represents a highly interesting project to keep in mind for a different development scenario.

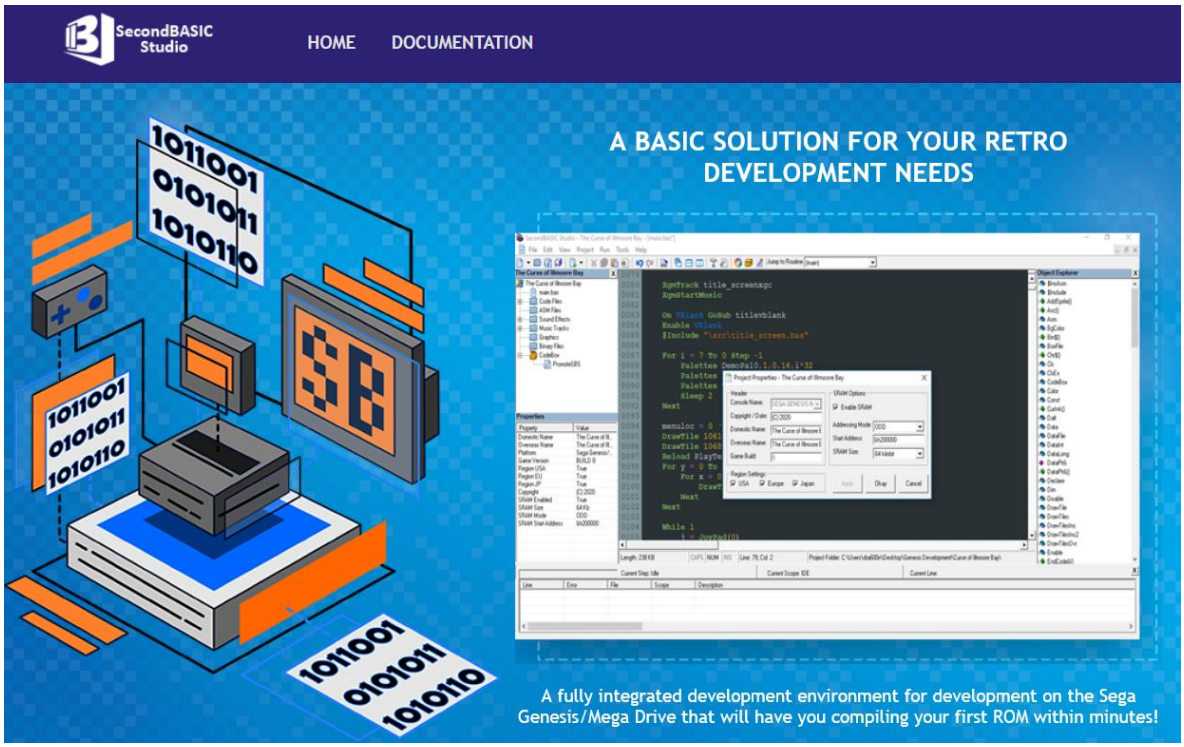


Fig. 07: Second Basic Studio download page

As such, in the field of high-level, optimized programming for the SEGA Mega Drive platform, SGDK has no true competition at the time of writing. That is not to say competition is completely mute. While the high-level programming in the C language, offered by SGDK, is the obvious choice for a team-centric, modern development model, the original development style utilizing the low-level assembly language native to the Motorola 68000 CPU is still very much an option *(Hugues Johnson, 2022)*. While very cumbersome to learn for modern audiences, there are possible performance gains to be profited from, as well as the assurance that all code is written entirely by the development team itself, without needing to credit the framework creators. Specifics of how implementation in such a style works and the notable differences between performance and development of both styles is discussed thoroughly in the related research paper.

One final note on the SGDK framework chosen is its compatibility with the server and client operating systems. Developed with the Windows operating system in mind, all development machines will be able to locally compile any given project utilizing the default SGDK release. Unix-like Linux and Mac operating systems however need to use one of the described forks of the SGDK repository, namely GenDev or MarsDev most commonly. While these are designed for linux and cross-platform SGDK development respectively, both presume the usage of an x86(_64) processor architecture, posing a problem for the chosen ARM based raspberry pi machine. After struggling with the library compilation process for both named forks on the raspberry pi, a user by the name of Doragasu in the noteworthy SGDK discord messaging server, recommended two of their personal repositories. These contained scripts and docker container definitions for building a 68000 compatible GNU C compiler (GCC v6.3.0) and the latest SGDK framework release for the ARM architecture. *(Doragasu, 2022)* The usage and adaption of these repositories will be discussed in the later section on server setup.

4.1.3 — Integrated development environments

After the surrounding technical necessities are now well defined, the concrete development tools to be used on the development machines and with the described framework need to be discussed. The possible alternatives and reasoning of choice should equally be of concern here. Starting with the tool directly responsible for instructing the framework, the chosen code editor or development environment is to be carefully considered, as it will act as the programmers interface. SGDK supports multiple editors by default and can be easily configured to work in a multitude of others, through its simple reliance on standardized “makefiles”, instructing compilation.

The “CodeBlocks” IDE (Integrated Development Environment) seems to be among the most popular choices. As an open-source, fully featured example of an IDE, it brings many qualities to the table. On the flipside however, it is a rather old project and is starting to show its age, both visually and feature-wise. While other similar IDEs like “Eclipse” and “QTCreator” can be configured for development work with SGDK, one more popular modern choice exists in “Visual Studio Code”. Being by definition an extensible text editor,

not a full IDE, this widespread tool comes in both a small memory footprint and a sleek, modern visual style. While less feature-rich by design, its extension database can be used to only add the desired functionality. As such, the “C/C++” extension by Microsoft and the “Genesis Code” extension by Zerasul can be used in conjunction to turn this text editor into a lean SEGA Mega Drive development power tool. Adding to this, a completely open-source version of “Visual Studio Code” exists, named “VsCodium”, even bringing peace to the desire for open-source tool usage, wherever possible. After evaluating all approaches this tool clearly stuck out in its approach, usability and timely design, quickly becoming the chosen development tool for the toolchain.

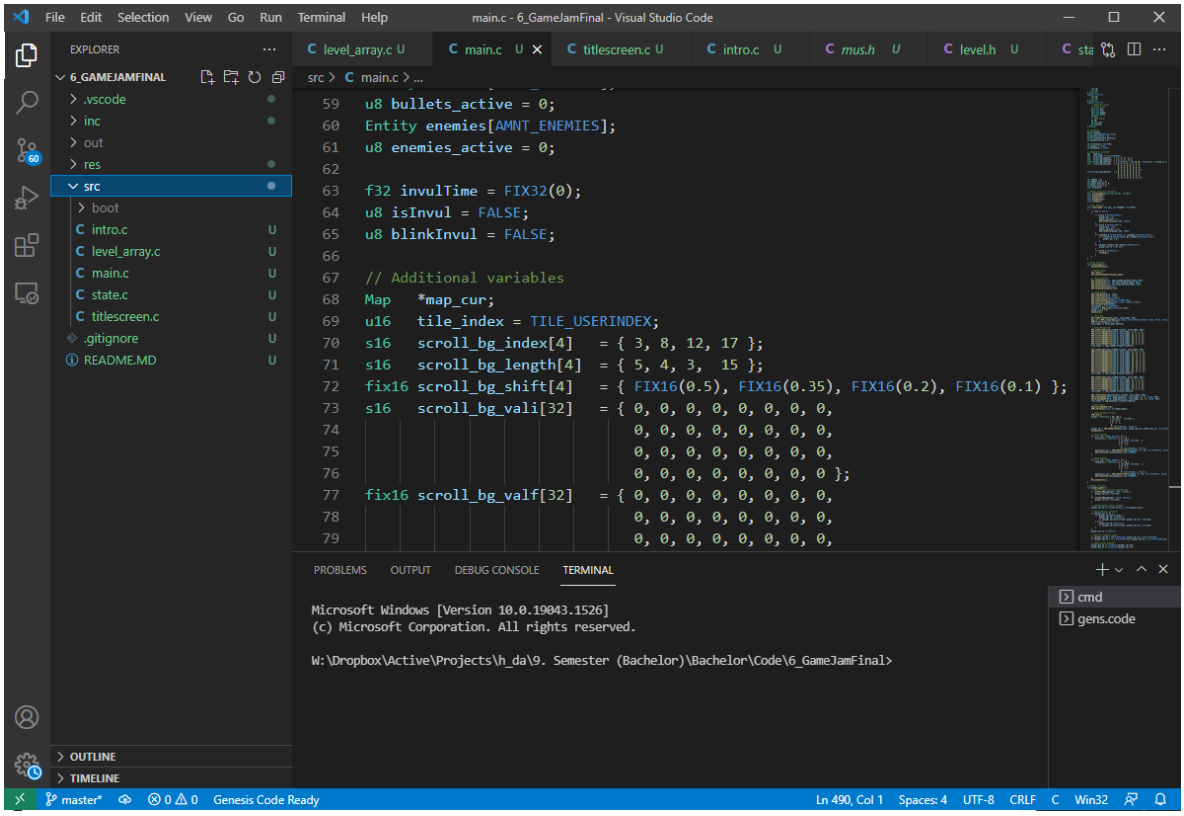


Fig. 08: Visual Studio Code with an SGDK project opened

4.1.4 — Graphical art tools

Shifting the attention to tools for the creative pursuits next, the tool choice for art creation equally has great implications for the quality of collaboration. While entering text and executing compilation functionality is rather similar between development environments, the

approaches to art creation are more abstract between different pieces of software. Hence, artists typically like sticking to tools they are familiar with. With the clear topical focus on pixel-art, tightly restricted by palette usage, a more specific choice of art software is important however.

With Adobe Photoshop being the industry standard art tool, it can be configured for usage in pixel art and even export in indexed color / low bits-per-pixel modes. Much like the previous IDEs however, the amount of additional, unnecessary features and inability to be deployed portably (without registration and uplink to adobe servers) make the tool an unjustifiable burden for a concise set of tools, despite its universality. Much like the mentioned “Visual Studio Code” a simple art tool that only does what is needed in the current circumstance should be evaluated. For pixel-art specifically, many options exist. From ports of classic art software, originally used back in the 16-Bit era, such as “GrafX2” *(Chez, 2001)* to the modern web-based pixel-art solutions like “Piskel” *(Piskelapp, 2022)* and “Pixilart” *(Bryan Ware, 2022)*, multiple options seemed adequate.

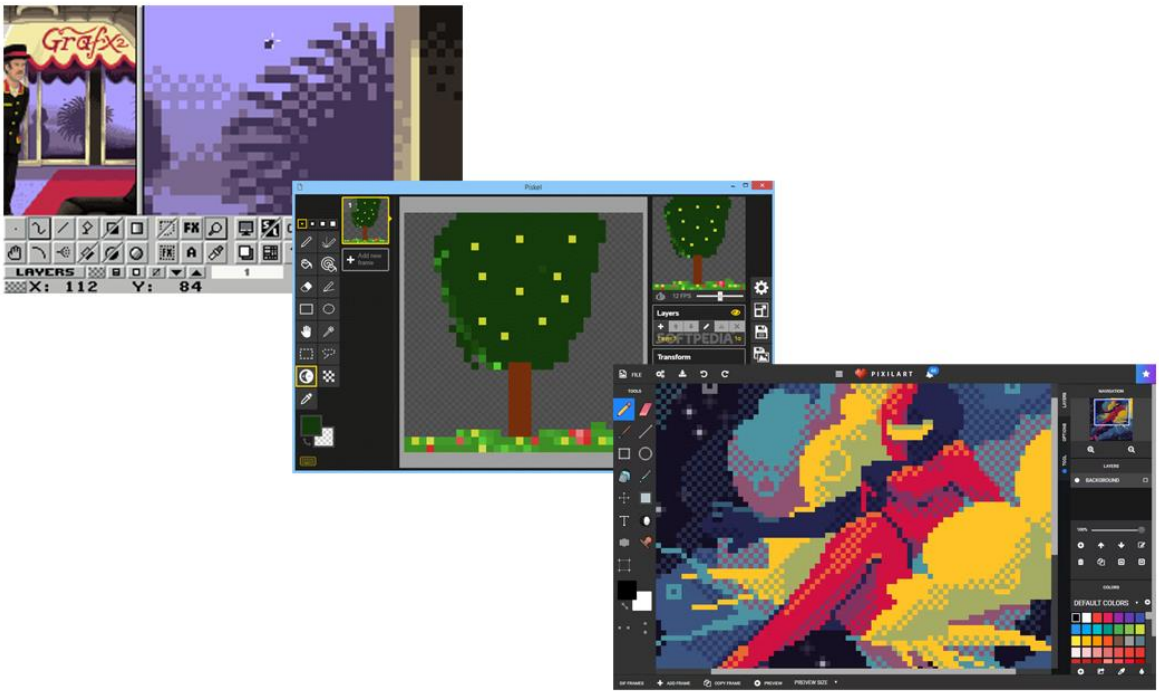


Fig. 09: Grafx2, Piskel, Pixilart - Alternative pixel art software

One of the most popular solutions in this space however is the venerable “Aseprite” *(Igara Studio, 2022)*, a stand-alone portable pixel-art tool for modern PCs. Being designed from the ground up around pixel art, palette usage and indexed color output, not to mention many helpful features missing in more main-stream art software, such as pixel-perfect line smoothing, this tool packs a great punch at a low resource footprint. While the project is available under a less permissive license and official versions are indeed paid, the source code is actually openly available and can be built manually to obtain the full version. As stated in its license however, commercial products created with the tool need a license to be legally viable. Despite this minor gripe, it is largely recognized as one of the best pixel-art solutions and thereby is to occupy the slot for art creation tool in the present toolchain.

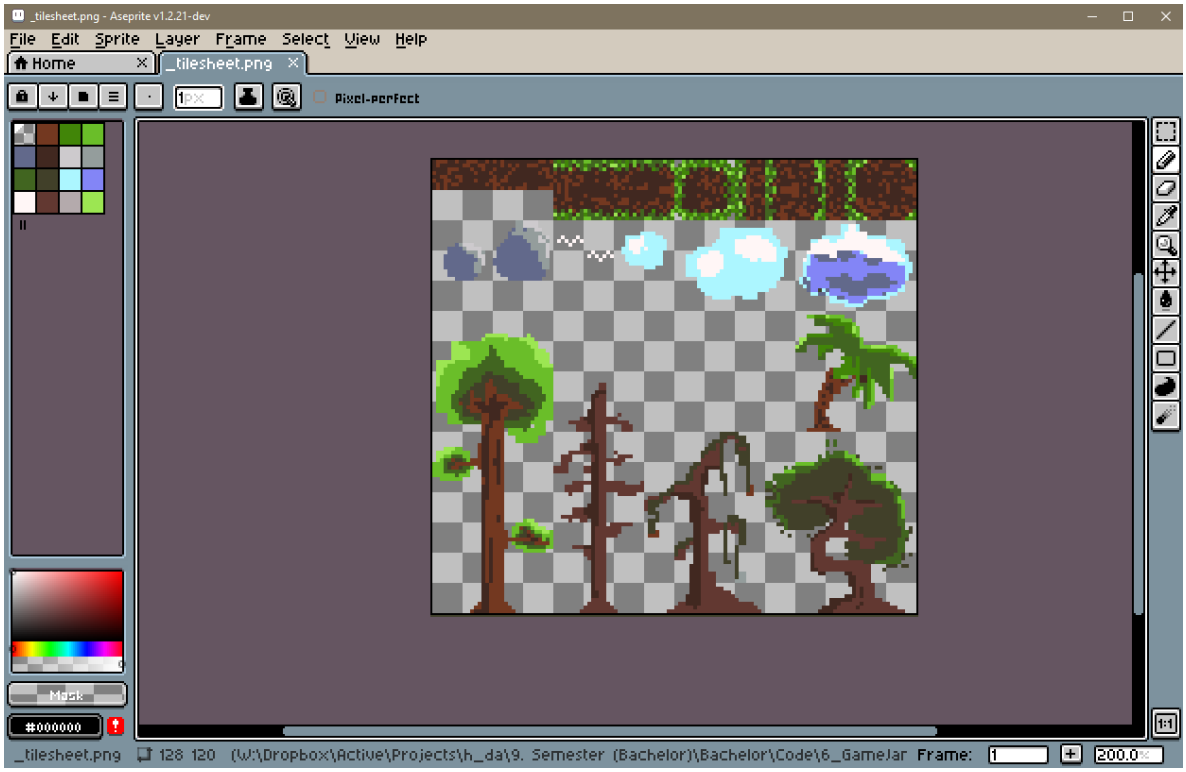


Fig. 10: Aseprite pixel art tool, showing custom tilesheet

4.1.5 — Music production & sound tools

The topic of music production on such a target platform comes with even greater restrictions. The YM2612 and TI-SN76489 sound chips of the SEGA Mega Drive, providing FM and PSG sound generation respectively, have drastically limited and fundamentally different

functionality from what is expected on any modern computing system. With 4 simple PSG channels and 6 more capable FM channels, all music and sound effects have to be represented through this limited output. The sixth FM channel has the built-in functionality of allowing low bitrate sample playback, often being used for drums or voices in iconic game soundtracks of the system. The creation of simple WAV sample files for music accentuation or sound effects could be handled by just about any piece of audio software. The one settled on for the toolchain is “Tenacity” *(Tenacity Team, 2022)*, a recent fork of the popular and efficient open-source “Audacity” recording software. This fork stemmed from controversies surrounding the “Audacity” project, regarding undocumented data collection and telemetry, uncompliant with the GPL-3 public license *(Tom Nardi, 2021)*. The fork not only removes potentially malicious code, but also modernizes the user interface a bit. Being a simple audio recording software, there is not much else to be said about the tool.

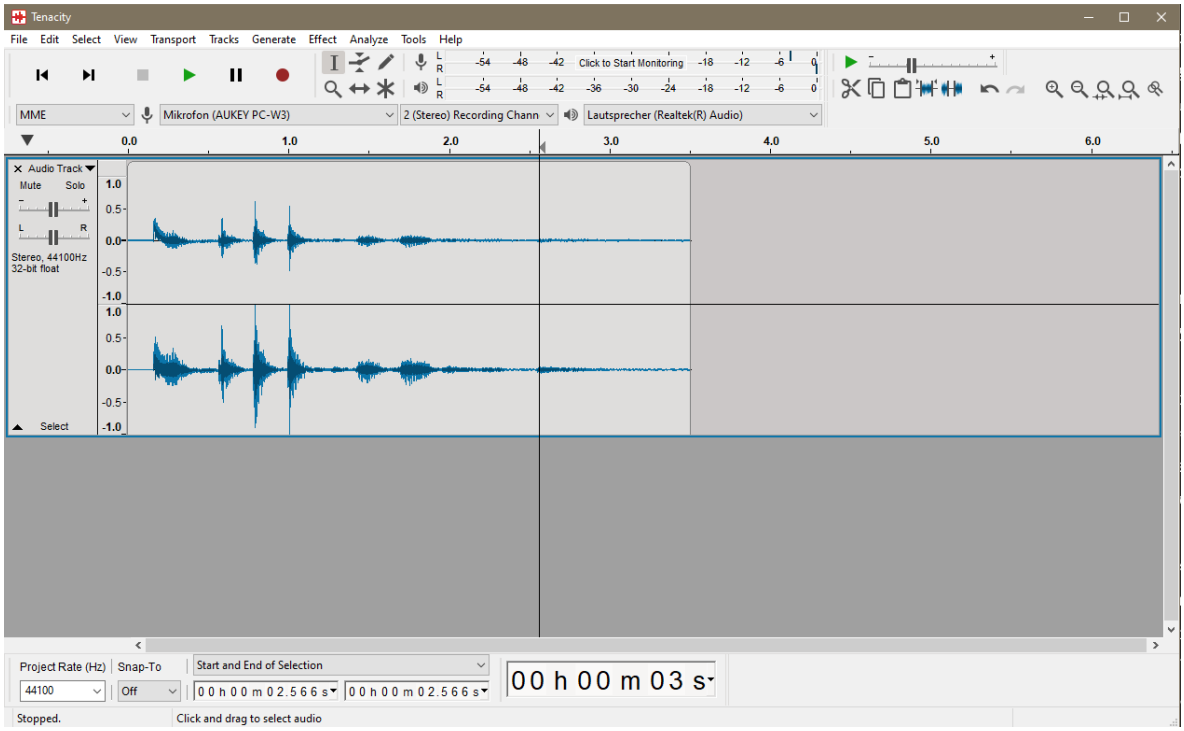


Fig. 11: Tenacity open-source audio editor

The music production software for the system however is a very different story. With the limited amount of sound generation and reproduction capabilities any modern music production approach fails to support such a restrictive sound environment. Additionally, the

SEGA Mega Drive as stated does not have an inherent operating system that could expose some sort of sound API. Very specialized software for the express purpose of outputting for this console is in order. Upon release of the console, there was no standardized audio workstation, each studio had to develop their own methodologies and software. In 1991 the project “GEMS” (Genesis Editor for Music and Sound Effects) *(Video Game Music Preservation Foundation, 2020)* was first released, being a MS-Dos based graphical audio workstation, to be attached to a SEGA Mega Drive console via a debugger. Many famous soundtracks of the console were indeed composed using this software. Some even go as far as noting this software for creating the “typical” Mega Drive sound, as many composers simply used the template instrument presets bundled with the software. Being a truly ancient tool, it is clearly not the best choice for a modern workflow.

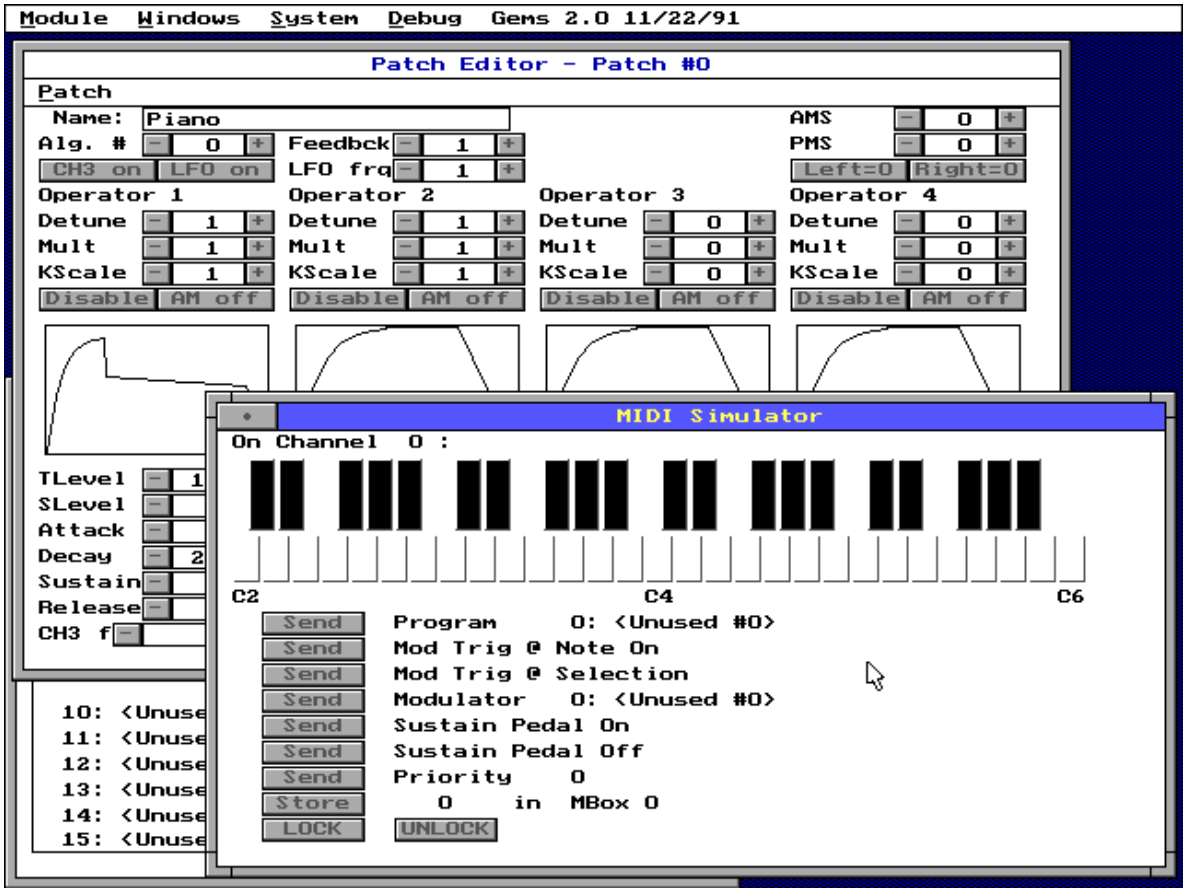


Fig. 12: GEMS historic music creation software

A newer option is the “VGM Music Maker” by Shiru. This basic windows application implements a tracker-style composition environment and comes with many features missing from previous comparable projects. The operation and user interface layout is indeed quite similar to the general-purpose music tracker “OpenMPT” for Windows as well. The name also implies the output format: .VGM (Video Game Music). This format was specifically designed by homebrew developers of the SEGA Mega Drive, SEGA Game Gear and NEO GEO platforms, incorporating the FM instrumentation, PSG capabilities, and compressed sample storage. While still an available and capable enough application, VGM music maker is no longer actively developed.

The clear music production software to choose however for the creation of SEGA Mega Drive music has to be “Deflemask” *(Leonardo Demartino, 2011)*. This modern tracker-style music application supports a wide variety of consoles from the SEGA Mega Drive, SEGA Master System, NEO GEO, Nintendo NES to the Commodore 64 and more. Multiple output formats, a concise yet capable selection of effect commands and a modern user interface, plus a very active development cycle make this tool the obvious modern choice for music creation on the platform. After over a decade on the market, the team behind the tool have not slowed down their continued development either, as multiple updates to the interface and feature set have been rolled out over recent years. A version of “Deflemask” is even available for mobile devices, bringing a usable tracker style audio software to a touch-screen dominated landscape successfully. Being a paid, close-source application, some might still want to look elsewhere, if this is more of a concern. From a standpoint of advancements and usability however, there are no close competitors currently on the market.

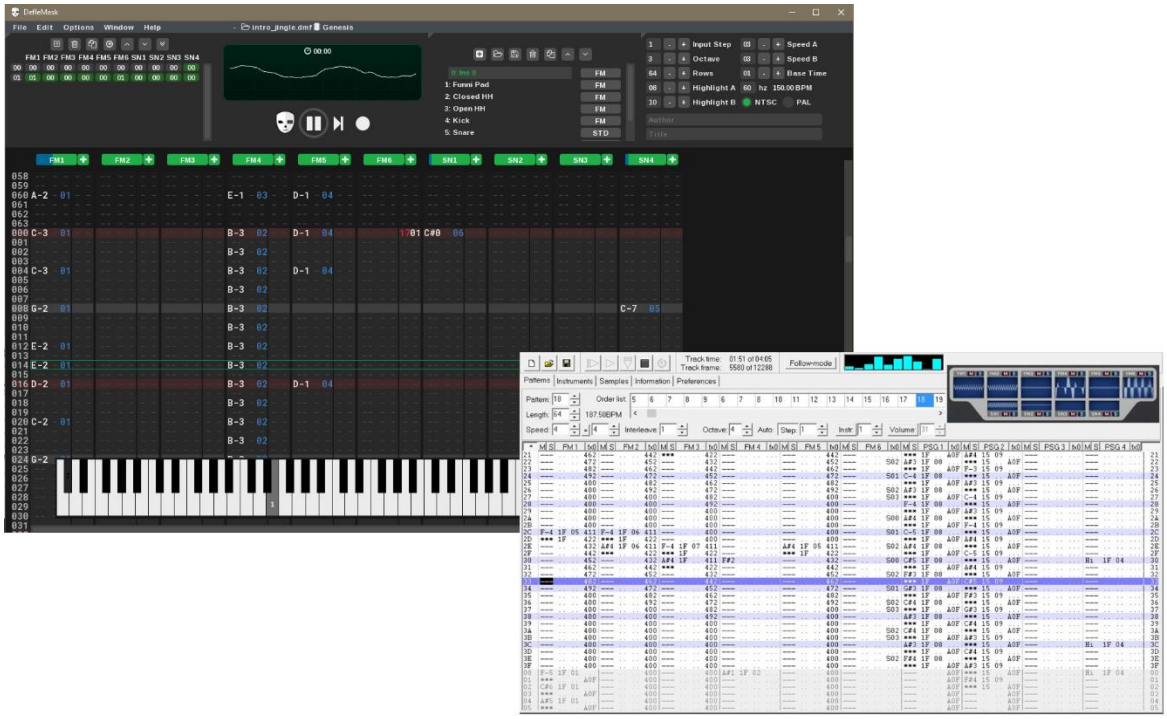


Fig. 13: Deflemask and VGM Music Maker playing tracked music

4.1.6 — Level design tools & restrictions

The last creative discipline to be evaluated for tool selection is level design. By its nature, level design is based entirely on the style of game developed and the assets with which to design levels. As such it is both highly dependent on the chosen implementation to represent levels within the games logic and the created or desired assets for said game. Because of this highly interconnected nature, the level design process inherently needs to make assumptions about the games nature. Within the SGDK framework, certain assumptions of how levels act and are defined are already made, accepting an image file of the level, to then remove duplicate tiles and reduce the level into a grid of indexes into the created tileset. As such, any image editor could theoretically be used as a level design tool. This data structure is referred to as “Map” in SGDK and the imported level asset is defined as a “Map Definition”. The problem with the pure image based approach however, is that interactions with the level and objects / sprites. While functions to check a map location for a tile index do exist in SGDK, they are rather slow. Having a separate integer index array for only objects to be collided or otherwise interacted with is vital for execution speed.

While many platform-agnostic level editors exist, one popular modern choice is “LDTK” (Level Designer Toolkit) *(Sébastien Bénard, 2018)*. With a simple modern user interface, variable tile sizes and layers it brings a lot to the table. One important aspect is one of its layer-types “Integer Grid”, allowing to assign an integer value to each position in the map. This grid can be exported as a JSON format, but clever conversion utilities *(Under-Prog, 2021)* have been written, allowing the export as a text file, containing a C language array definition, ready to be included in any SGDK project immediately. A truly worthwhile tool, allowing a very comfortable, modern level design utility to be used in the current homebrew context.

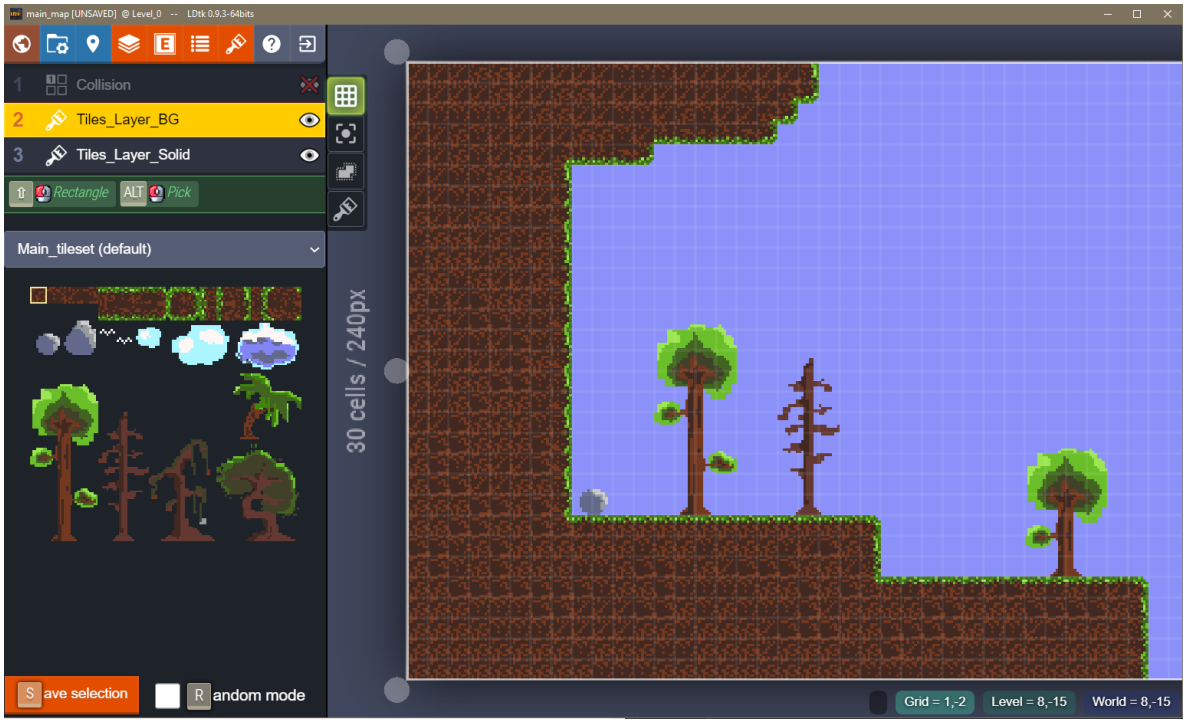


Fig. 14: LDTK, editing levels with tilesheet

4.1.7 — Emulation software for development

Wrapping this section up, the last elements to be discussed are the emulation programs used during development. While the resulting rom files can absolutely run on real SEGA Mega Drive hardware, during testing the usage of emulators is almost a requirement, as manually copying each locally compiled rom file to a flash cart is a truly unnecessary hurdle. Which emulators to choose is an important question however, as many programmers and teams have

attempted their own emulation implementation for the SEGA Mega Drive over the years. During the evaluation chapter of the related research paper, the performance, feature set and accuracy of the four most common emulators for the console have been thoroughly examined. In short, the results showed that perhaps the most commonly used emulator “Kega Fusion” *(Steve Palmer, 2010)* had strong inaccuracies when compared to both the real hardware and even other emulators, making it not desirable for homebrew development.

The “GensKmod” *(Kaneda, 2006)* emulator is a particularly popular choice in the homebrew scene for the console, as it provides robust debugging features. The ability to output arbitrary debug strings from the rom file into a message window is a great tool during development, even being implemented as a simple function call into SGDK itself. The accuracy here is much better, but being a debugging emulator it does permit certain executions that would result in crashes on original hardware, especially around memory allocation and deallocation. Still for its unmatched debugging features it is a vital tool of the development process.

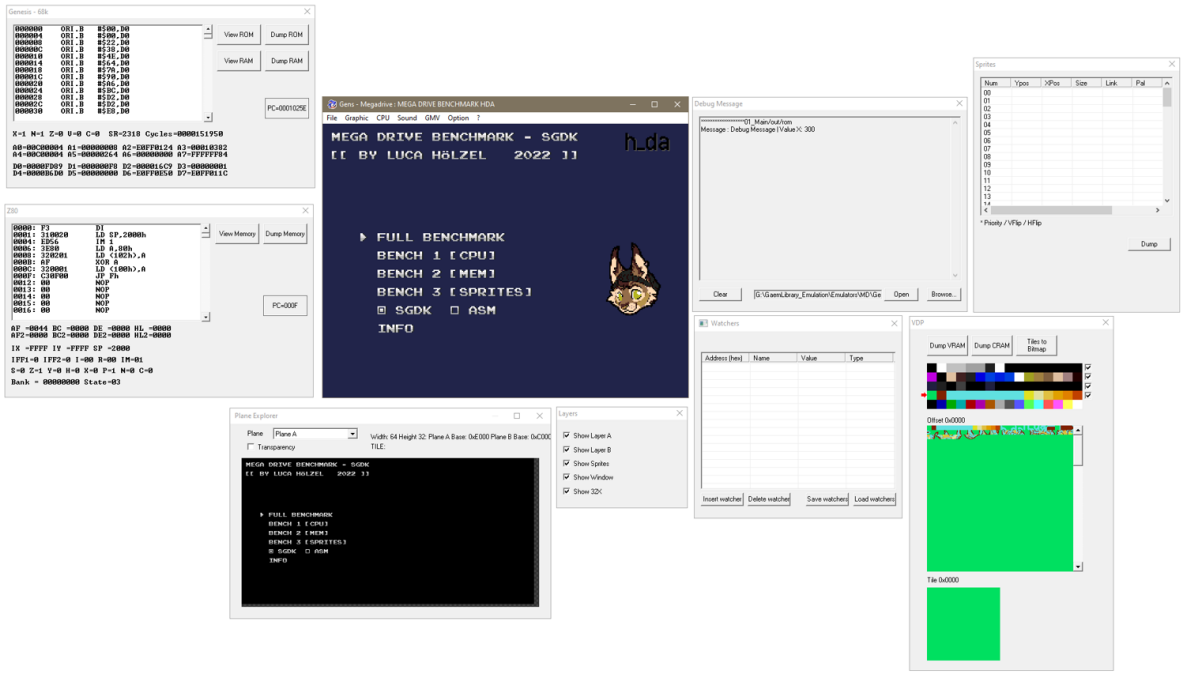


Fig. 15: GensKMod emulator, displaying multiple debugging windows

To test for hardware accuracy without need for the real hardware, the “BlastEM” *(Michael Pavone, 2019)* emulator is referenced by many in the SEGA Mega Drive homebrew

community as the most accurate emulator. The findings produced during the mentioned research paper evaluation mirror this sentiment. Not only is performance very close to real hardware, but all encountered execution errors were handled the exact same way here, making this emulator the ideal sanity-check during development of experimental features. As such the emulators “GensKMod” and “BlastEM” are designated to be part of the toolchain.

4.1.8 — Visualization of chosen tools within toolchain

Within the toolchain constructed and thoroughly documented in the following sections, the tools evaluated and chosen as part of this section each serve a specific purpose. As that purpose is rather obvious based on the type of software it is, the interactivity between these tools and the server-side tasks is much more important to successfully convey. Referencing back to the third chapter of this thesis, the individual jurisdiction and role these tools play in the overall toolchain construction can now be expressed concretely, revealing more of the inner workings behind client-server interaction.

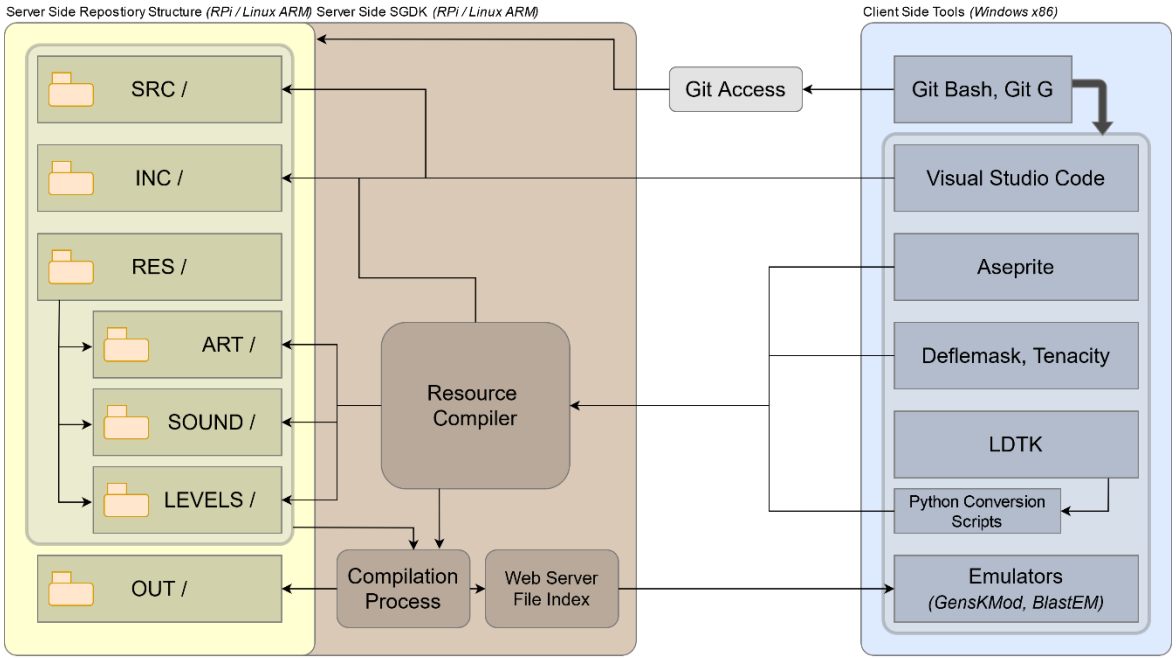


Fig. 16: Toolchain visualization, individual tools and jurisdictions

With git clients acting as the main point of interaction between the server-side repository and the client-side tools, they essentially facilitate the collaborative development to take place. While the chosen development editor interacts exclusively with the source code and include files for development, the creative tools all output their files into the resource folder, converted and handled by the SGDK resource compiler. For level editing in particular, an extra step involving the mentioned conversion script is prefaced before this process. Through build-automation, the server can build certain branches of the repository and distribute them around the network, for use with the mentioned emulators.

How these visualized processes are implemented and how these interactions are facilitated between the client and server is discussed in great detail in the following section, making it useful to reference back to the above diagram for clarity.

4.2 — Constructing chosen elements into a toolchain & server

Now that all client-side tools to be used as part of the toolchain and the reasoning behind their choosing have been discussed in-depth, the construction of the toolchain around them will be discussed in the following section and its sub-sections. As described, the implemented toolchain is highly server-centric, leading to the importance of the discussion around intricacies of the server setup itself. After clarification of the server functionality and jurisdiction, the implementation of the toolbox program will be explained. The configuration of the previously evaluated tools, to be managed by the toolbox, will equally be discussed, reinforcing the desired role each tool is to play in the toolchain.

4.2.1 — Setup process for raspberry pi server

As previously discussed, the hardware chosen to act as the development server for the purposes of the toolchain, is a raspberry pi 4 model b single-board computer. Running a quad core ARMv8 based CPU at 1.5GHz, with 4GB of RAM available on the chosen model, this tiny computer should have all the required power for the simple implementations of the desired tasks. A storage medium in form of a micro-SD card is required as the systems boot drive. Other than the computer itself and a storage medium, a power supply and an ethernet

networking cable is required for operation. Configuration of wireless network access is possible, but never recommended for any server application. A protective casing and even a passive (heatsinks) or active (fan) cooling solution are great to have, but not a requirement by any means.

Fig. 17: Raspberry Pi Components with passive & active cooling

Being a simple machine, the basic installation process to get the chosen operating system up and running is rather straight forward. The latest version of the mentioned RaspianOS linux operating system can be downloaded from the official raspberry pi website, either as a packaged .ISO drive image, or through their custom imaging software. After inserting the micro-SD card into the actively used computer, the custom imaging software can be used to flash the operating system onto the medium. In case of the .ISO image being used, a different

third party imaging software such as “Rufus”, “Balena Etcher” or “Win32 Disk Imager” can be used to transfer the bootable system to the micro-SD card. After successfully completing this step however, one vital thing for configuration is to use the present computers file manager to add an empty file named “ssh” into the “boot” folder found on the newly flashed micro-SD card. This empty file simply instructs the boot process to automatically launch the secure shell access service on system boot.

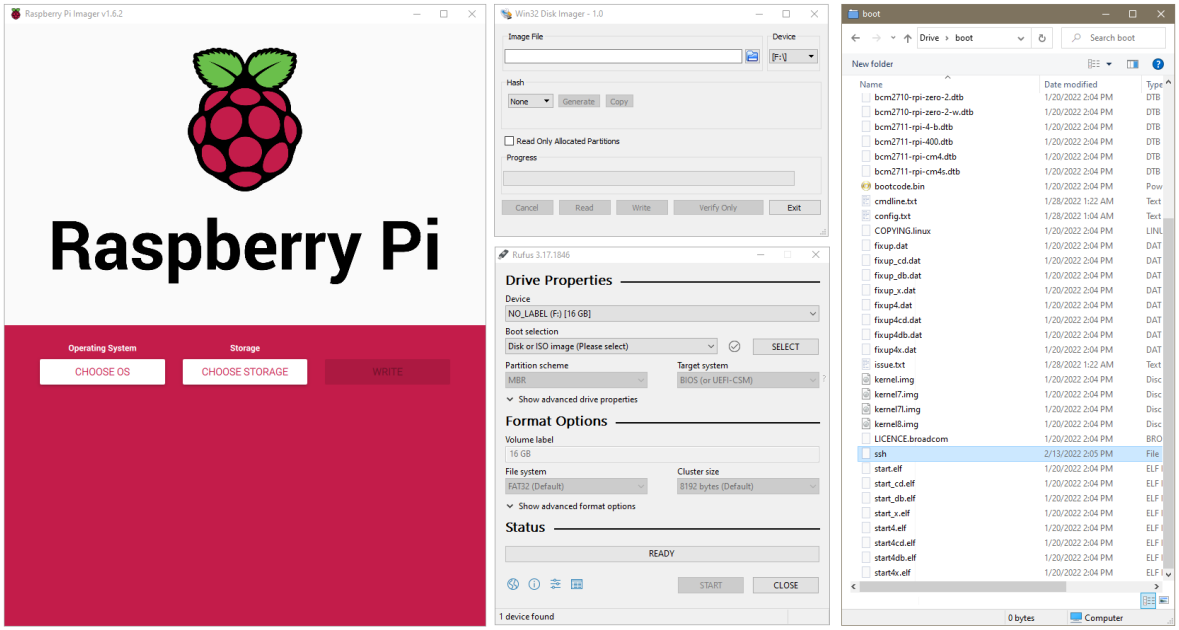


Fig. 18: Raspberry Pi Imaging and SSH activation

Finally, the medium can be inserted into the raspberry pi computer. After connecting the ethernet cable and the USB-C (micro USB on older raspberry pi revisions) power supply to the machine, the system should boot up immediately. From this point onwards, the machine itself can be left alone, as all access and configuration is now done remotely. From the same computer as before (or any computer on the network for that matter), a terminal or command prompt window can be opened, with which shell access to the raspberry pi machine is gained via the use of the “ssh” command. To access the machine, either the local IP address of the systems hostname needs to be known. For simplicity sake, the hostname should be easiest to remember. By default the hostname should be “raspberrypi”, the default username “pi” and the default password “raspberry”. By executing this command, a successful secure shell session should be established.

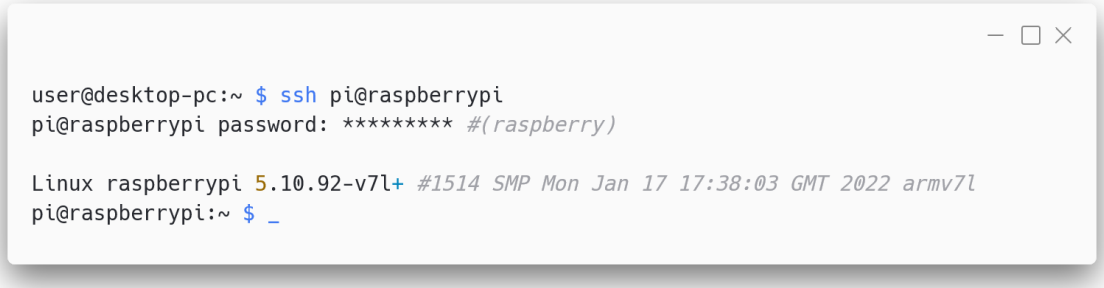


Fig. 19: Initial SSH login

Beyond the obvious security measures of changing the username, password, hostname and RSA-key based ssh authentication to be used, more thorough security measures can be set in place. On the machine itself, an easily configurable firewall, such as “UFW”, can be installed, blocking all requests on undefined ports. On the network, the router itself can be configured to act as a firewall for the machine, which however is entirely dependent on the model of router or other networking equipment used. For the sake of this simple implementation, these additional security measures were omitted. During a later chapter, these options will be reflected upon once more.

Now that the basics are covered, the setup process for getting a simple version control system with secure, user-credential based network access up and running, needs to be discussed next.

4.2.2 — Git version control setup and network access

As previously noted, a server-centric model of collaboration is a clear requirement for a workflow and toolchain to be considered contemporary. Especially in current times, in which in-person meeting and local collaboration are often not possible, structuring the tool usage around a central version control server, might that be a local or a remote one, has become more than just modern, it is truly required in many cases.

With version control being a very important tool, but also highly dependent on its deployment scenario, many solutions do exist. The main distinguishing quality to divide the available solutions into are centralized and distributed (or decentralized) solutions. Centralized

solutions rely entirely on a single central server, from and to which all versioning information is transmitted. Typically, these solutions are intended for corporate and enterprise use, as a centralized approach is much better adapted to a closed-source development approach. Distributed version control systems rely on each contributor having their own version of a repository, equal in status to all other repositories. The changes made can be synchronized through a server to other contributors, but the local repository of each contributor is not tied exclusively to the incoming changes from a central server.

A common example of a centralized approach taken is the popular “Helix Core / P4V” *(Perforce, 2022)* version control system, while the most popular distributed version control system is “Git”. Git is in fact the most popular version control system world-wide, by far. It’s open-source nature and close relation to the linux project, both developments being spear-headed by Linus Torvals, propelled it to the true industry standard it is today.

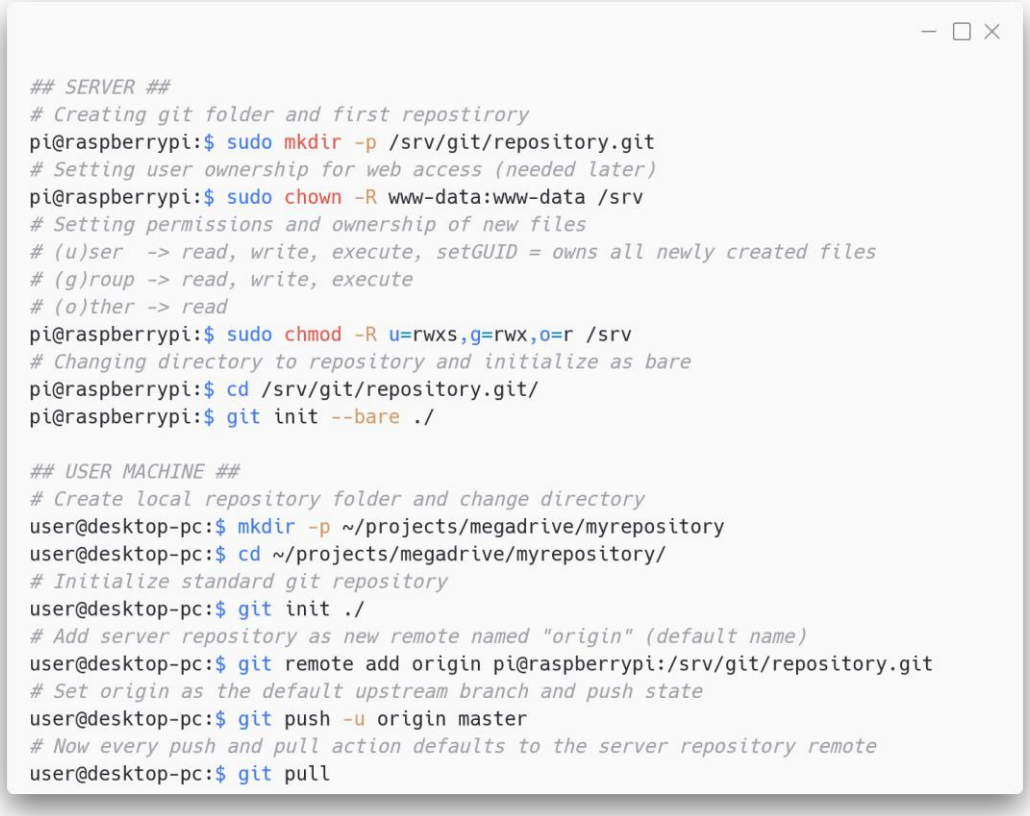
With the ubiquity of Git in almost all corners of development work, the basic usage of Git, or any version control client for that matter, is out of the scope of this thesis. Why git was chosen however was explained to establish context in the realm of version control systems themselves.

With the mentioned relation to the linux project, the underlying simplicity of the unix philoshopy is a corner-stone of Gits deployment. Instead of configuring a central server application, which then manages a set of folders, utilizing system-wide configuration files, Git operates entirely self-contained within any desired folder, locally or on a server. To initialize a Git repository on the raspberry pi server, the presence of Git should be ensured, by installing it via the apt package manager.

Fig. 20: Package manager system update and git installation

Once installed, the desired folder or folders to contain the Git repositories should be designated. Since the internal file structure of the machine will become part of the later accessed sever path, keeping the folder shortly named and close to the root of the filesystem is a best practice. Once created, new folders can be created within this folder for each desired repository to be hosted. Just by mere existence, these folders will do nothing at all. To initialize one of these folders as a repository, it needs to be initialized as a bare repository. Bare in this case refers to the fact, that the repository will not structure its internal files necessary for history and commit relation into the common ".git" hidden folder, but instead will lay out these files bare within the folder itself. This is important, as the server-side repository should not be initialized as a normal repository. A normal initialization or cloning command in git does not only create or synchronize the versioning information, but will also build the actual working directory (work tree) from that versioning information. This will become necessary for a different purpose on the server-side later however.

Once these steps are completed, the just created server-side repository can be accessed from any local machine in the network. Somewhat counter-intuitively however, to begin using the repository, a user still needs to initialize a local repository on their machine, then adding the server-side repository as a "remote" for the local repository, thereby synchronizing the states between the two. This methodology is simply an artifact of the distributed style of version control implementation, as the server repository is in essence just another repository on another computer, holding no direct authority over the changes of the local repository.



```
## SERVER ##
# Creating git folder and first repostirory
pi@raspberrypi:$ sudo mkdir -p /srv/git/repository.git
# Setting user ownership for web access (needed later)
pi@raspberrypi:$ sudo chown -R www-data:www-data /srv
# Setting permissions and ownership of new files
# (u)ser  -> read, write, execute, setGUID = owns all newly created files
# (g)roup -> read, write, execute
# (o)ther -> read
pi@raspberrypi:$ sudo chmod -R u=rwxs,g=rwx,o=r /srv
# Changing directory to repository and initialize as bare
pi@raspberrypi:$ cd /srv/git/repository.git/
pi@raspberrypi:$ git init --bare ./

## USER MACHINE ##
# Create local repository folder and change directory
user@desktop-pc:$ mkdir -p ~/projects/megadrive/myrepository
user@desktop-pc:$ cd ~/projects/megadrive/myrepository/
# Initialize standard git repository
user@desktop-pc:$ git init ./
# Add server repository as new remote named "origin" (default name)
user@desktop-pc:$ git remote add origin pi@raspberrypi:/srv/git/repository.git
# Set origin as the default upstream branch and push state
user@desktop-pc:$ git push -u origin master
# Now every push and pull action defaults to the server repository remote
user@desktop-pc:$ git pull
```

Fig. 21: Git server and local initialization, configuration

Among the basic terminology and functionality omitted during the beginning explanation, the nature and usage of “Branches” in Git does hold rather high importance to later toolchain deployment. In short, branches allow for separate paths of commits to be defined. With each branch essentially just being a pointer to a set of commits issued to the repository, different features or elements can be added on one branch, ignoring all changes and new commits happening on the remaining branches. This allows for different versions of a repository to be worked on at once and then later being merged back together when the implemented element in a given branch is accepted.

Different styles and usages of branching form different development strategies, like “Trunk” style development, where all work is done in one branch, forgoing mege conflicts and poblems, or “Feature Branching” in which each new feature is implemented in an isolated

branch, then merged as necessary. The ins and outs of these industry-wide approaches are beyond the scope of this thesis however. The branching strategy adapted for the toolchain is a discipline based one. As different disciplines will work on separate elements and in separate directories, each discipline working inside their own branch should provide good separation of work and induce few merge conflicts, since jurisdiction over files is quite clear. Since certain elements of creative work in art, level design and music overlap one-another (level designer needing artist tilesets, …), the creative disciplines work in one conjoined creative branch, whereas programmers work in a separate code branch.

All accepted changes into these branches are then iteratively merged into the master branch, representing the current state of development. When at any given point the current state of the game (or experimental features) should be tested not just locally, but by all game testers as well, the state of the master branch will be merged into the testing branch.

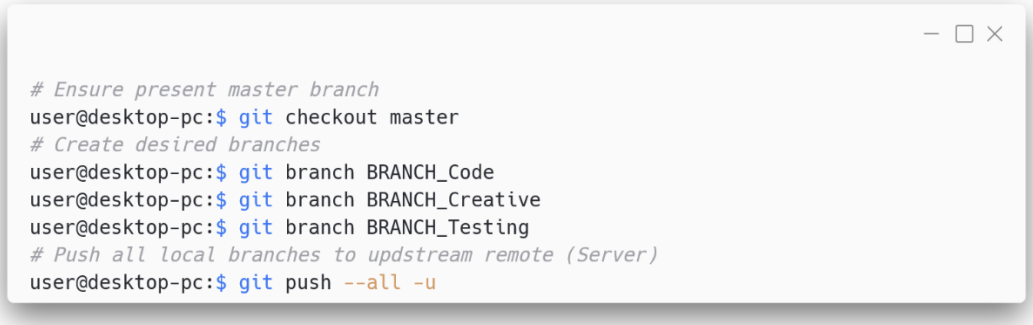


Fig. 22: Git Branch configuration

In order to facilitate data exchange between the local repository and the newly added remote repository on the server, via the push, pull and fetch commands, the protocol with which to send and receive the data has to be discussed. Multiple protocols are available, each coming with their own advantages and caveats, depending on deployment situation.

The basic protocol is the native “Git” protocol. Typically the fastest protocol available, it is only ever intended for local deployment without user-based authentication, as it does not provide any authentication features whatsoever. For group-based work with credentials

required, this approach can still work, but is less secure and will most likely become a risk as team sizes increase. The most commonly used modern protocol is the “SSH” protocol. With secure shell access being a widely used and secure standard, it is no wonder it takes the crown as the most common option. The security here is key. Like mentioned during the last section, secure authentication with remote servers is typically achieved with public and private encryption keys, serving as a replacement for classic passwords. For the sake of the initial implementation of the server and toolchain for this thesis however, the creation of key files for all development machines and the following registration of these keys on the purely local server was unnecessary. Instead, another common protocol was used, the “HTTP” protocol. Being the driving protocol of the web, http obviously has robust features for authentication and user management as well. Moreover, http is the only protocol in the context of Git, that allows both username and password to be specified within the access command (useful for local script automation):

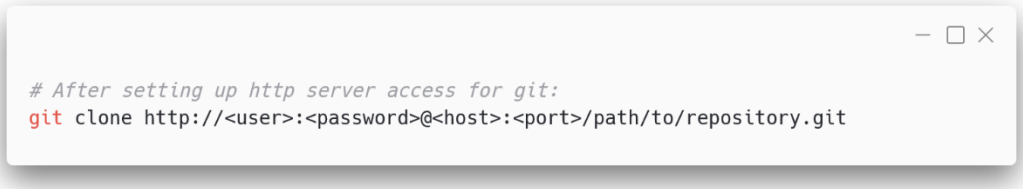
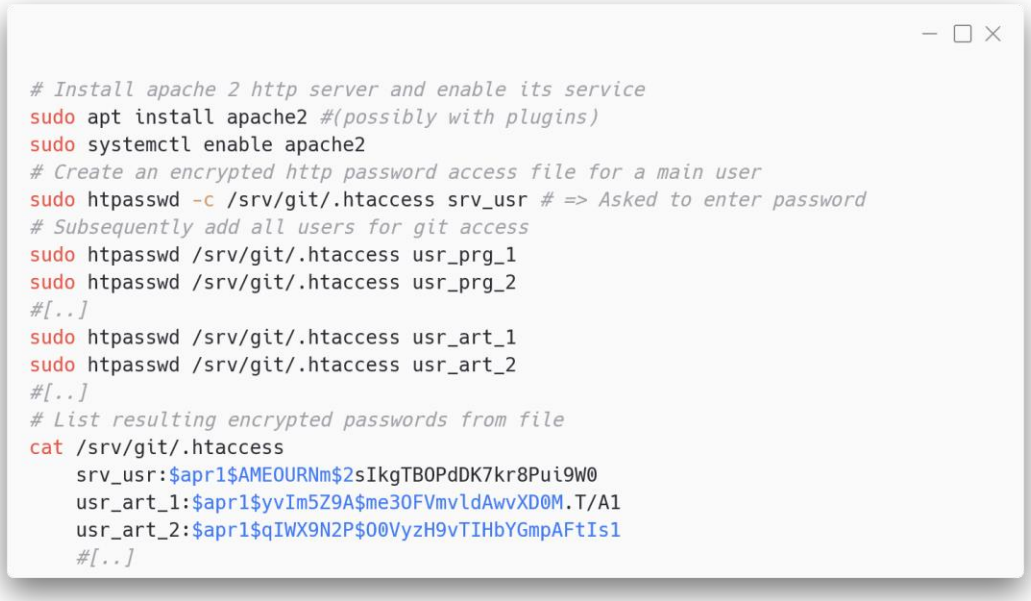


Fig. 23: HTTP accessing of a Git repository

In order to use the http protocol for Git however, a http web server has to be installed on the server machine, managing the access over the protocol. This might seem like a burden, but for the following server tasks, a web-server will be required as will be explained later. As such configuring it during this step simply prepares important system aspects for later iteration.

Which http web server to install typically comes down to two choices nowadays, mainly between “Apache 2” *(Apache Project, 2021)* and “Nginx” *(Nginx Inc, 2021)*. Both have advantages and disadvantages, but apache 2 was chosen mainly because of familiarity. After installing it from the available apt repository, credential-based access for the desired git users

can be setup. Utilizing an apache command to set and subsequently encrypt passwords for all desired users, the resulting access file can be placed in the established git folder. Of note here is that the usernames used for http access do not have to exist as user accounts on the linux server. These are two separate types of credentials.



```
# Install apache 2 http server and enable its service
sudo apt install apache2 #(possibly with plugins)
sudo systemctl enable apache2
# Create an encrypted http password access file for a main user
sudo htpasswd -c /srv/git/.htaccess srv_usr # => Asked to enter password
# Subsequently add all users for git access
sudo htpasswd /srv/git/.htaccess usr_prg_1
sudo htpasswd /srv/git/.htaccess usr_prg_2
#[..]
sudo htpasswd /srv/git/.htaccess usr_art_1
sudo htpasswd /srv/git/.htaccess usr_art_2
#[..]
# List resulting encrypted passwords from file
cat /srv/git/.htaccess
    srv_usr:$apr1$AMEOURNm$2sIkgTBOPdDK7kr8Pui9W0
    usr_art_1:$apr1$yvIm5Z9A$me3OFVmvldAwvXD0M.T/A1
    usr_art_2:$apr1$qIWX9N2P$O0VyzH9vTIHbYGmpAFtIs1
    #[..]
```

Fig. 24: Apache2 installation and user password creation

After this, the ports configuration file of the apache server can be updated to listen to a new port we wish to facilitate http driven git access under. Keeping this desired port in mind, the next step is to create a virtual host file for the desired git access. A virtual host in apache acts as a desired local folder to be made available (or hosted) under a given port, with given environment settings. The needed virtual host settings required for git were adapted from the official git documentation on the subject of http access. Once this virtual host file is present in the folder for available sites, it can be enabled for hosting either through the apache enable command, or through a manual symbolic link of the file into the enabled folder. To test the configuration, the apache 2 service can be reloaded (or completely restarted). The reported status of the service can then provide information on possible problems with the implemented

settings. If all clear, the http access to the previously initialized and then remotely added Git repository can be tested from the previously used computer.

```
#>> /etc/apache2/ports.conf
Listen 80
Listen 8080  # Newly added port
#[..]

#>> /etc/apache2/sites-available/git.conf
<VirtualHost *:8080>
        ServerAdmin webmaster@localhost
        SetEnv GIT_PROJECT_ROOT /srv/git   # Git repository locations
        SetEnv GIT_HTTP_EXPORT_ALL
        ScriptAlias /git/ /usr/lib/git-core/git-http-backend/

        Alias /git /srv/git
        <Directory /usr/lib/git-core>
                Options +ExecCGI -MultiViews +SymLinksIfOwnerMatch
                AllowOverride None
                Require all granted
        </Directory>

        DocumentRoot /var/www/html
        <Directory /var/www>
                Options Indexes FollowSymLinks MultiViews
                AllowOverride None
                Require all granted
        </Directory>

        <LocationMatch /git/.*\.git>
                AuthType Basic
                AuthName "Git Verification"
                AuthUserFile /srv/git/.htaccess  # HTTP access users / passwords
                Require valid-user
        </LocationMatch>

        ErrorLog ${APACHE_LOG_DIR}/error.log
        LogLevel warn
        CustomLog ${APACHE_LOG_DIR}/access.log combined
</VirtualHost>

# Enable virtual host via apache command
pi@raspberrypi:$ sudo a2ensite git
# or via direct symlink
pi@raspberrypi:$ sudo ln -s /etc/apache2/sites-available/git.conf /etc/apache2/sites-
enabled/git.conf
# Update Apache2 service
pi@raspberrypi:$ sudo systemctl reload apache2
```

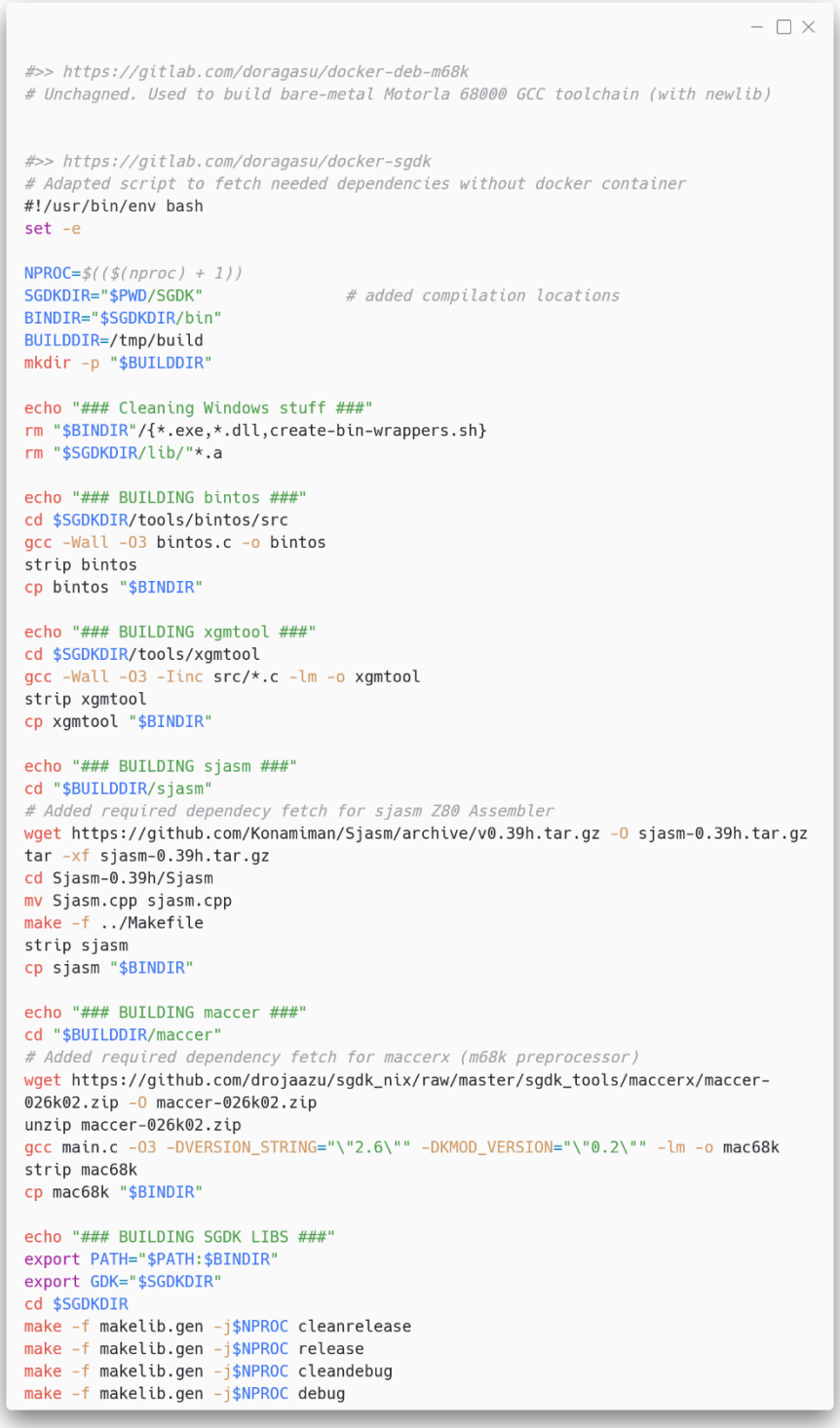
Fig. 25: Port and Virtual Host configuration

4.2.3 — Server-side building and framework compilation

Now that a working version control setup has been deployed for use with multiple contributors, facilitating server-side compiling of the actively used repository is an important step to a modern workflow and toolchain. Local building during active iteration is still absolutely necessary. But deploying a set of implementations and assets for testing should be built on the server and distributed to anyone on the network, allowing for rapid testing, without the need for game testers to be actively using the repository itself.

Before such a build-automation can be implemented, building of SGDK projects themselves needs to be evaluated for the hardware. As stated previously, SGDK is intended for Windows machines and the available cross-platform forks are still designed around the x86(_64) architecture. Meaning compilation of both the SGDK libraries and the subsequent compilation of SGDK projects using those libraries on the ARM based raspberry pi is not officially supported.

Luckily the mentioned discord server member Doragasu recommended their script repositories for creation of a necessary build environment and GCC compiler setup for ARM based SGDK deployment. These scripts were intended to be deployed using the docker container format, but simply encapsulated standard bash scripts. For the sake of transparency during implementation, containerization of such important parts of the toolchain as SGDK itself was to be avoided. As such the provided scripts for the build environment were used with little modification, but the SGDK build scripts were modified to fix certain dependencies and setting the correct build paths.



```
#>> https://gitlab.com/doragasu/docker-deb-m68k
# Unchagned. Used to build bare-metal Motorla 68000 GCC toolchain (with newlib)


#>> https://gitlab.com/doragasu/docker-sgdk
# Adapted script to fetch needed dependencies without docker container
#!/usr/bin/env bash
set -e

NPROC=$($(nproc) + 1))
SGDKDIR="$PWD/SGDK"               # added compilation locations
BINDIR="$SGDKDIR/bin"
BUILDDIR=/tmp/build
mkdir -p "$BUILDDIR"

echo "### Cleaning Windows stuff ###"
rm "$BINDIR"/{*.exe,*.dll,create-bin-wrappers.sh}
rm "$SGDKDIR/lib/"*.a

echo "### BUILDING bintos ###"
cd $SGDKDIR/tools/bintos/src
gcc -Wall -O3 bintos.c -o bintos
strip bintos
cp bintos "$BINDIR"

echo "### BUILDING xgmtool ###"
cd $SGDKDIR/tools/xgmtool
gcc -Wall -O3 -Iinc src/*.c -lm -o xgmtool
strip xgmtool
cp xgmtool "$BINDIR"

echo "### BUILDING sjasm ###"
cd "$BUILDDIR/sjasm"
# Added required dependecy fetch for sjasm Z80 Assembler
wget https://github.com/Konamiman/Sjasm/archive/v0.39h.tar.gz -O sjasm-0.39h.tar.gz
tar -xf sjasm-0.39h.tar.gz
cd Sjasm-0.39h/Sjasm
mv Sjasm.cpp sjasm.cpp
make -f ../Makefile
strip sjasm
cp sjasm "$BINDIR"

echo "### BUILDING maccer ###"
cd "$BUILDDIR/maccer"
# Added required dependency fetch for maccerx (m68k preprocessor)
wget https://github.com/drojaazu/sgdk_nix/raw/master/sgdk_tools/maccerx/maccer-
026k02.zip -O maccer-026k02.zip
unzip maccer-026k02.zip
gcc main.c -O3 -DVERSION_STRING="\"2.6\"" -DKMOD_VERSION="\"0.2\"" -lm -o mac68k
strip mac68k
cp mac68k "$BINDIR"

echo "### BUILDING SGDK LIBS ###"
export PATH="$PATH:$BINDIR"
export GDK="$SGDKDIR"
cd $SGDKDIR
make -f makelib.gen -j$NPROC cleanrelease
make -f makelib.gen -j$NPROC release
make -f makelib.gen -j$NPROC cleandebug
make -f makelib.gen -j$NPROC debug
```

Fig. 26: ARM M68K GCC and SGDK build scripts

After this, the resulting locations of the GCC 68000 header files, as well as the resulting library and include directories of the SGDK build were added to the systems environment variables.

```
#>> /etc/environment
if [ "$(id -u)" -eq 0 ]; then
  PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/opt/SGDK/bin"
else
  PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/local
/games:/opt/SGDK/bin"
fi
export PATH
export GDK="/opt/SGDK"
```

Fig. 27: Environment settings for SGDK usage

After testing the build process with sample projects included in the SGDK repository, the compilation requirements for server-side building were deemed to be met. Finally, the automation of this process for the actively used repository needs implementation. Automated building is generally a common part of modern development toolchains. With web-based continuous integration (CI) organization tools, such as “Circle-CI” or the built-in integration tools of GitHub and GitLab, the feature of continuous build automation is an important standardized process in the software development industry. The web-based interconnectivity and shier scale of these implementations however is truly unnecessary for the purposes of the currently deployed toolchain. As SGDK uses basic “makefiles” for building, simple scripts for building and output management will facilitate all the build-automation necessary. The automated execution of these scripts however might not be so clear.

One detail about Git, omitted from the previous section, is the utilization of “Git Hooks”. One of the folders present in either the local “.git” folder or the bare structure within the servers git directories is labeled “hooks”. Keeping with the simple, entirely file-based implementation of git, these hooks are nothing but executable bash scripts, reading

information from the standard-input given to them via the git program. Git hooks typically act as “Pre”-actions or “Post”-actions. The first referring to scripts executed on the local machine, before the respective action is performed, whereas the latter are executed on the remote server repository, after the action has completed. For the present case, in which one of the repositories branches (testing branch) is supposed to be compiled with SGDK on the server itself, a “Post-Receive” hook is implemented as a simple bash script.

```
#>> /srv/git/md_dev_prj.git/hooks/post-receive
#!/bin/bash
TARGET="/srv/git/md_dev_automation/md_dev_prj_aut"
GIT_DIR="/srv/git/md_dev_prj.git"

while read oldrev newrev ref
do
        # Check if testing branch was pushed
        BRANCH=$(echo "$ref" | cut -d/ -f3)
        if [ "$BRANCH" != "BRANCH_Testing" ]; then
                exit 0;
        fi
        # Assemble git work tree in automation folder
        git --work-tree=$TARGET --git-dir=$GIT_DIR checkout -f "$BRANCH"
        # Change to work tree directory and build with SGDK
        cd "$TARGET" && make -f $GDK/makefile.gen release
        # Copy resulting rom file to web-server location, rename accordingly
        cp "$TARGET/out/rom.bin" "/var/www/html/sgdk_testing_build/"
        cd "/var/www/html/sgdk_testing_build/" && mv "rom.bin" "rom_$(date +%H:%M--%m-%d).bin"
done
```

Fig. 28: Git Post-Receive script for build automation

After defining the full path to the git repository and the target folder for the building process, multiple iterations of three variables are read from standard input. Each of these respectively concerning themselves with the old and new repository version, as well as the reference string, containing information about the repository head and the branch relevant during the operation. Since build actions are only supposed to be executed on the desired testing branch, the ref string is given to the cut command with the division character of slash (“/”). This leaves only the branch name, being stored in the “BRANCH” variable. This variable is then

checked against the desired branch name and execution is exited if a different branch was pushed to the repository.

If the testing branch indeed has been updated, the script begins assembling the repository and build environment. First, the previously mentioned work tree of the repository, normally not present in the server-side repository, is assembled in the designated target folder and the desired testing branch is checked out. At this point, the desired state of project to be tested has been assembled in the target folder, ready to be built. After changing the current directory into the target folder, the simple linux “make” command with the specified SGDK “makefile” and build configuration (release / debug) is executed. This will build the project present in its testing state and output it to the “out” folder, as “rom.bin”. Finally this resulting rom file is then copied to a folder within the *“/var/www/html/..”* folder, the typical location for data relating to web hosting services. As a last step, this file is renamed to append the build date to the filename.

These last steps might seem confusing, especially the copying action performed after building. As defined, the compiled testing version of the game is to be distributed across the network, not just to collaborators accessing the git server. The location the rom file is copied to is exposed as a local directory listing (index) web page via a separate virtual host file in the apache 2 configuration. The port this listing is hosted under can be accessed from any machine in the network, to get as many game test iterations done as possible, forgoing the need for game testers to waste time accessing most of the toolchain.

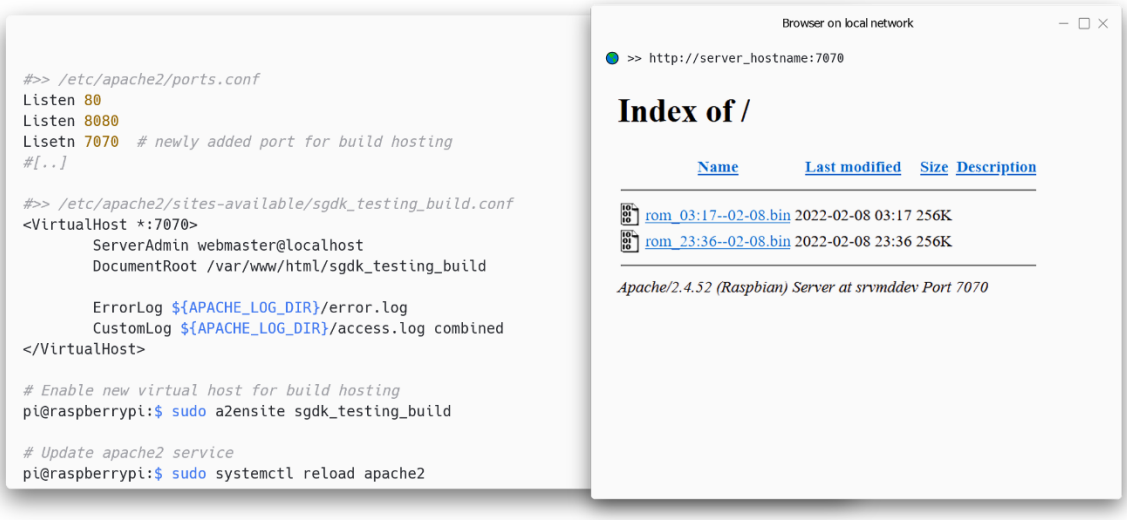


Fig. 29: Simple build-hosting in apache2

While a rather crude implementation of build-automation with network-wide build distribution, the provided setup absolutely works and has minimal dependencies to any third-party software, that is not open-source or otherwise not transparently understandable in an educational context. While better solutions and more thorough build-automation processes definitely exist, this basic model should serve both as an example of an effective, yet simple implementation and as a stepping stone for further advancements to be made.

4.2.4 — Server-side collaboration tools

First it is useful to reiterate the chosen tools for management and knowledge aggregation. The mediawiki framework is used to locally host a small expandable wiki containing information about the workflow, disciplines and individual tools. The focalboard task tracking software is used to manage tasks across disciplines and for a potential vision keeper to conduct time management and project management with. Both of these tools are implemented and hosted on the same raspberry pi server running the git and http server already.

The implementation of the mediawiki framework was conducted by the specifications designated in its documentation. After installing the SQL database and necessary PHP plugins for the apache 2 http server, already running on the system, setting up could begin.

Following the mentioned specifications, the SQL server was setup with a database to be later managed by the mediawiki php engine. The PHP plugins for the apache 2 server were enabled, as well as the provided virtual host file. Lastly the mediawiki installation unpacked into the *"/var/lib/mediawiki"* was symbolically linked into the *"/var/www/html"* folder, pointed to by the virtual host file. After a restart of the apache 2 server, accessing the designated port in the virtual host file (9090 in this case) from a web-browser revealed the interactive, web-based setup wizard. From there, credentials for the database and all other settings were applied, before being presented with an empty wiki, ready to be filled with information.

The information itself stemmed largely from the notes taken during the learning process both for the bachelor thesis program and the research paper. While not nearly complete, the presented information gives a great look into SGDK usage for programmers and basic tool usage for creatives, substituting additional knowledge with links to informative sites and tutorials.

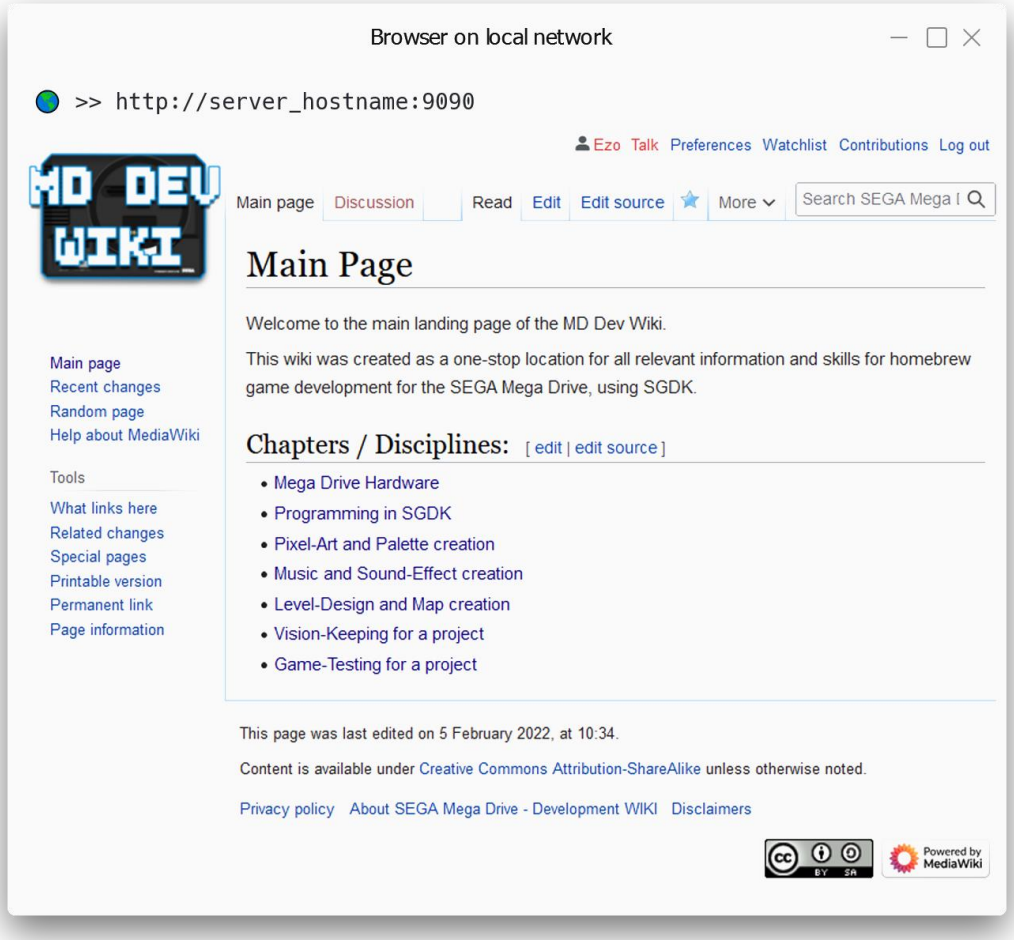


Fig. 30: Wiki hosted on local server

One of the main differences in deployment is the standard LAMP (Linux, Apache, MySQL, PHP) implementation for the mediawiki, running directly within the environment of the server, while the focalboard releases are only distributed for use within docker containers. Containerizing single server tasks into easily replaceable building blocks has the benefit of preventing program interconnected dependencies. This keeps the removal or change of one software package from effecting the stability or function of other software. The detriments however are that every container will often pull the same dependencies, resulting in larger storage and memory footprint, and the lack of choice of how pre-packaged docker containers operate.

In the case of focalboard for instance, the released docker container uses Nginx for http hosting, instead of apache 2. This caused certain ports on the main server to not receive data, instead being used by Nginx in the focalboard docker container. This process was not very transparent and can lead to hard to identify issues down the line, when containerized and standard approaches are mixed. For an educational context, the most transparent and understandable method of implementation should always be preferred. As such the usage of another task tracking tool, or a custom focalboard build might be of value.

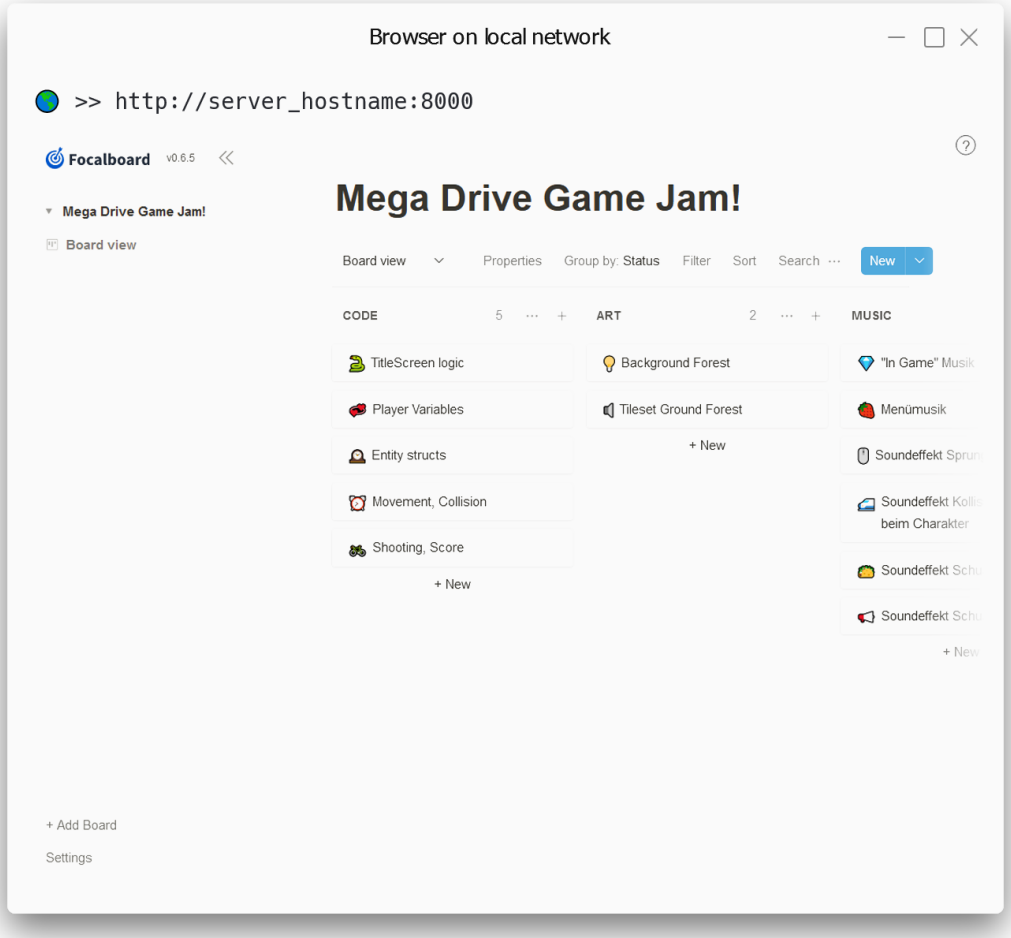


Fig. 31 Focalboard hosted on local server

The basic usage of focalboard is quite self-explanatory, especially for users of comparable web-based tools, such as “Trello”. Each user can create a locally stored account to log their changes with a username. The vertical containers to be filled with tasks and managed via simple drag&drop actions provide a good divisor between different disciplines and activities. As such, it facilitates an entirely locally hosted, tried and true method of task tracking, even if the setup has a few issues to be worked out.

4.2.5 — Creation of toolbox application

In order for the specific usage of the chosen toolchain programs to be simplified and streamlined, a toolbox program was implemented. The development environment and user interface toolkit chosen for such a task is not inconsequential, especially for later porting efforts. In this case the community edition of “C++ Builder” 10.4 by Embarcadero Technologies Inc. was chosen, utilizing the Windows VCL (Visual Component Library) user interface toolkit.

It is noteworthy that this RAD (Rapid Application Development) style development environments, the likes of “C++ Builder” and “Delphi”, have fallen out of favor over the years, either in favor of completely open-source frameworks or those offered as part of Microsoft’s “Visual Studio” or Apple’s “XCode”. Nonetheless, the personal familiarity with this toolkit for user interface development, stemming from school assignments in computer science classes, outweighed the necessity to learn an entirely new approach. Especially considering the relative compactness of the proposed feature set and scope of the application.

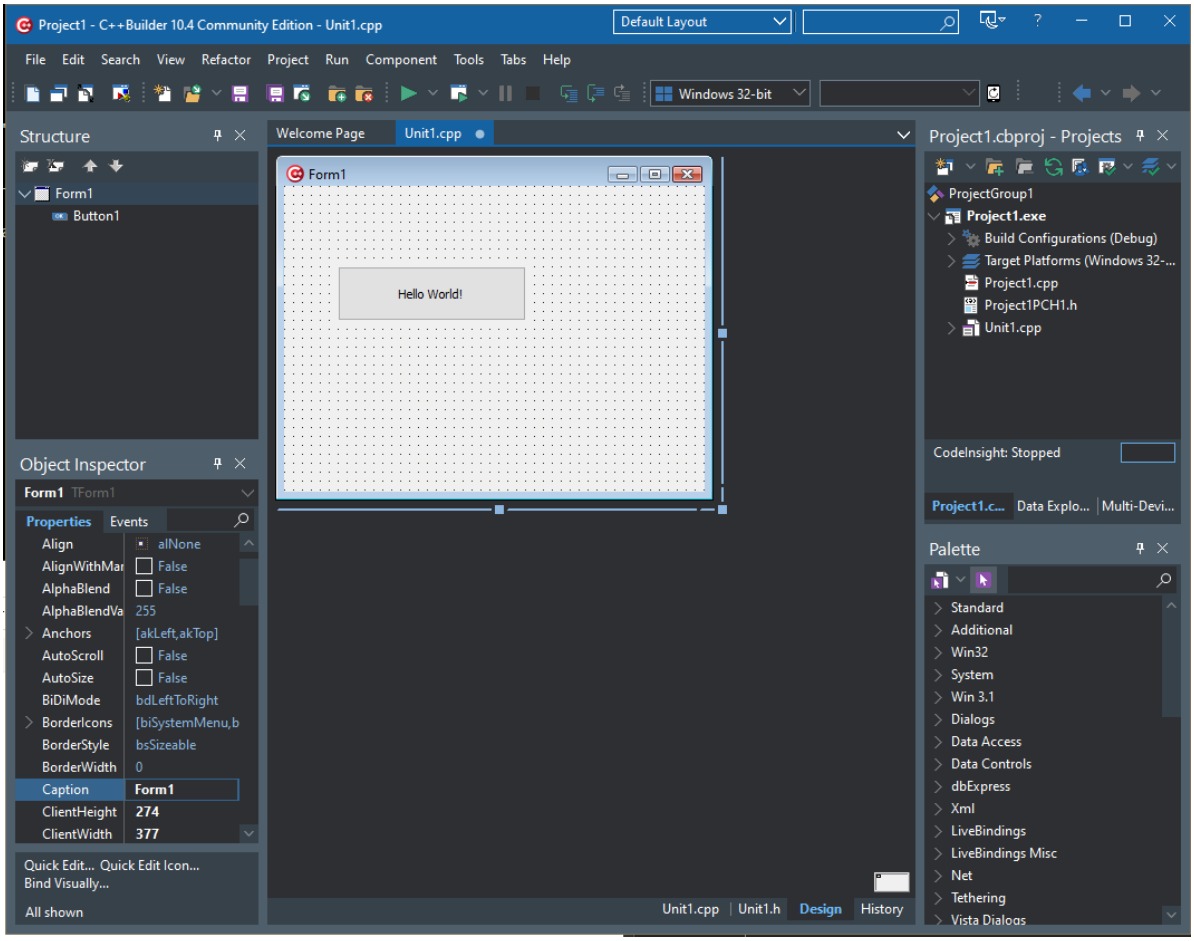


Fig. 32: C++ Builder 10.4 CE

As described in the previous requirements, the programs main goal is to mount required programs and folders under a virtual drive, ensuring that all absolute paths to software and folder structures are consistent across all machines. This process of mounting the required data is dependent on the team-role chosen by the user. Programmers for instance will receive a wider range of compilation scripts, emulator setups and folder references, in addition to their development tools. Artists and musicians on the other hand do not require deeper access into the repository than the resource folder in which to place their created assets. They also only require a simple release compilation script and have no use for the debugging emulator, reducing the amount of shortcuts presented to them.

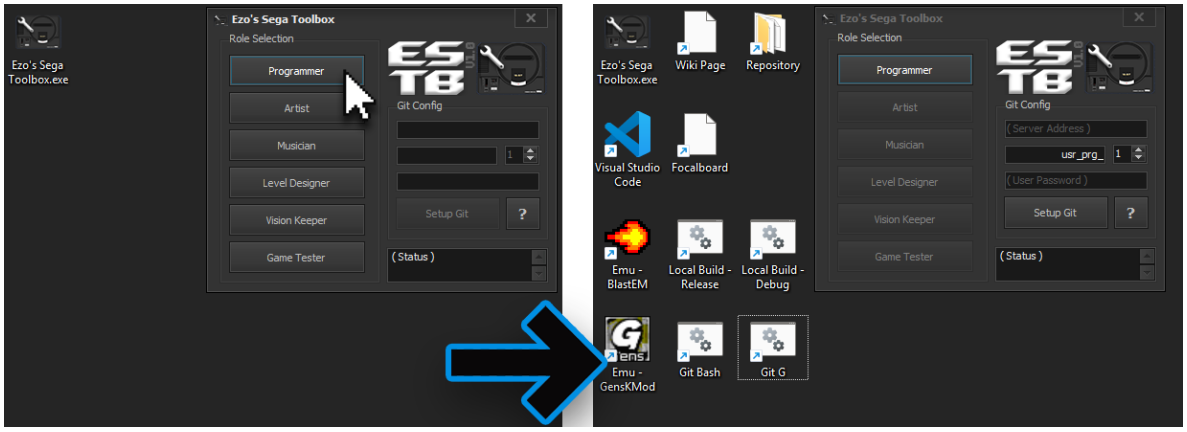


Fig. 33: Role selection and tool mounting inside toolbox

Additionally, the toolbox also acts as a front-end for the Git version control configuration on the client machines. Upon choosing one of the team-roles, the Git configuration user interface becomes available, after the respective team-role's programs are mounted and linked to the desktop. A read-only username is automatically chosen through their role and a number can then be appended to it. This way, the numbered (http) Git access usernames for programmers, artists, musicians (..) can only be chosen if the needed tools for the respective role are actually mounted on the system. This helps with keeping accidental commits from wrong usernames for the respective role to a minimum, leading to a more reliable commit history for later inspection.

The user interface itself was kept rather simplistic. A few group boxes for organization were used, dividing the role selection area from the other elements relating to the Git configuration. The layout was settled on from the start and had very literal additional iteration. The theming capability offered by the VCL toolkit allowed for a dark grey skin to be applied to the application, dubbed "Carbon", which fit in nicely with the overall Mega Drive themed project. Beyond the main application window and the simple, object-oriented actions between them, an information window and a settings window are also part of the application. The latter in particular implements a few useful settings for keeping the main window on-top of all active windows, turning off the use of system notifications, disabling the creation of desktop shortcuts for mounted tools and specifying a different drive letter to be used for tool mounting.

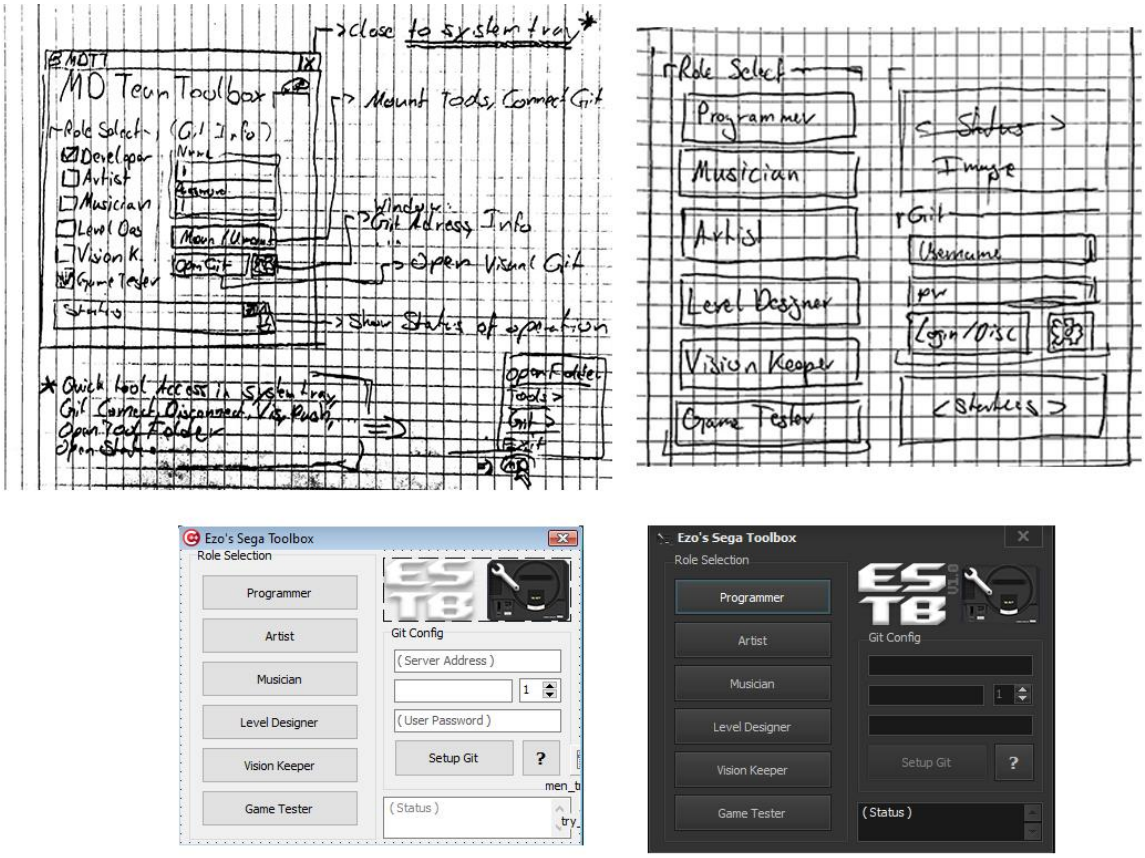


Fig. 34: User interface planning & iteration

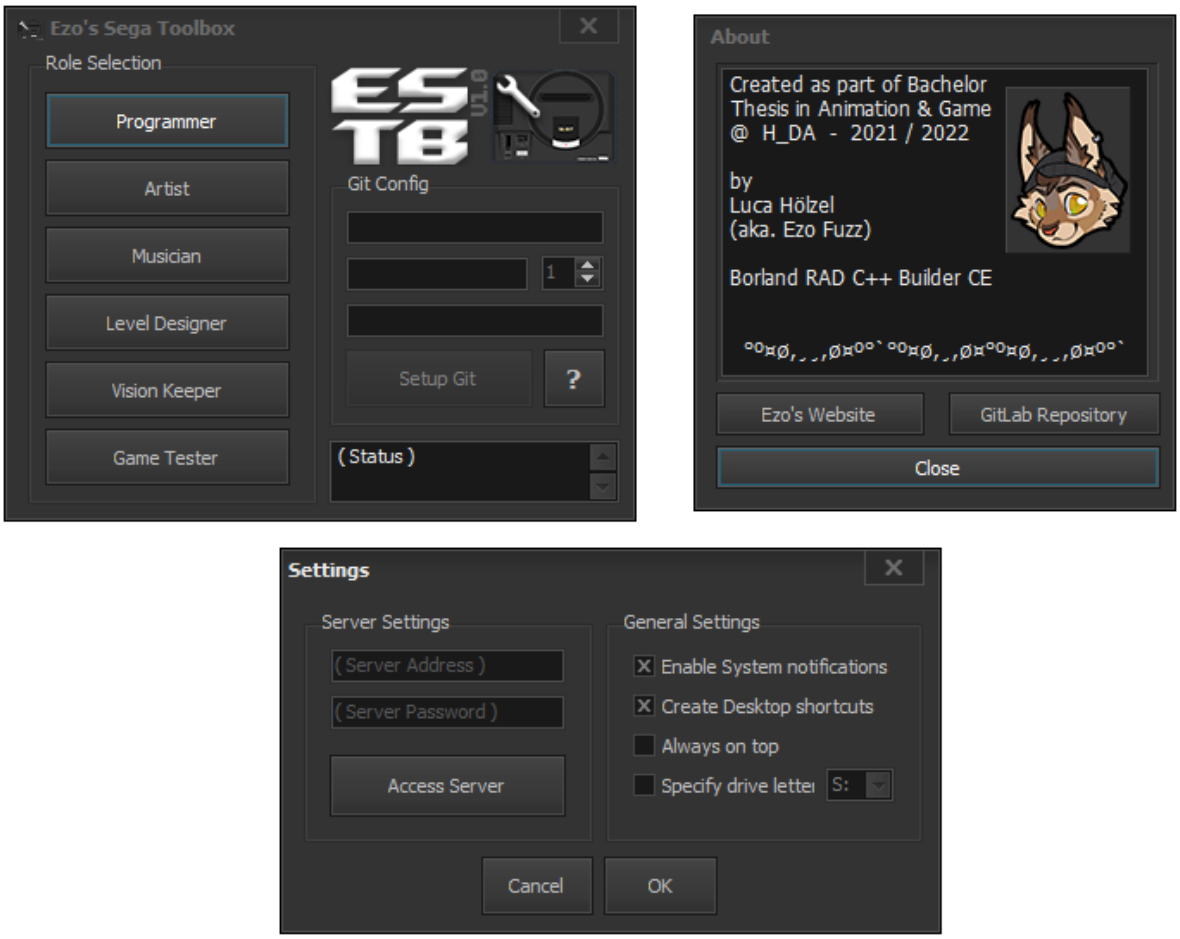


Fig. 35: Toolbox settings and information windows

Through the usage of C++ mutex objects, the application ensures that only one instance can run at the same time. This becomes necessary, as closing the main window only hides it, leaving an icon in the system tray to interact with the toolchain program more discretely. Selection and deselection of the active role and access to the settings window can all be controlled with a right-click onto this icon, making the program operate more like a daemon in the taskbar, if the user deems it more convenient.

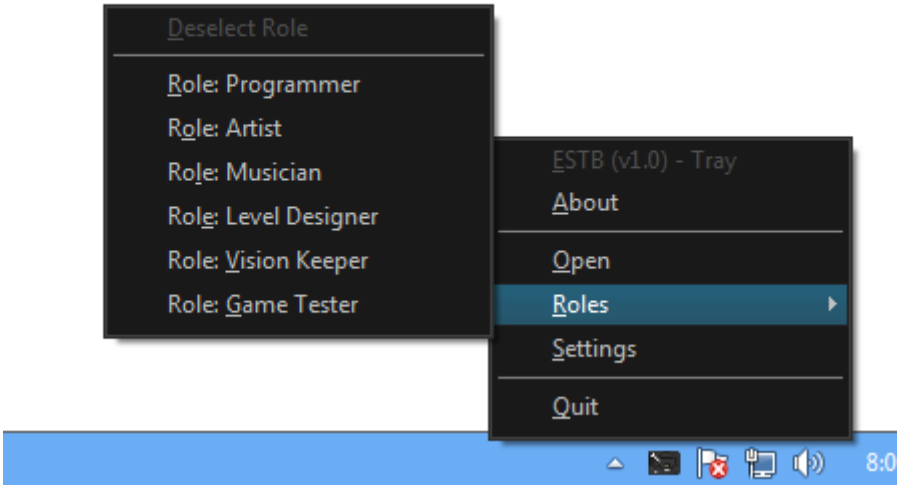


Fig. 36: Toolbox system tray menus

As also stated in the requirements, a toolbox program of this style would be amiss to implement all locations and names of individual tools and folders into the application itself. Having to recompile the application for the smallest change to tool selection or folder structure would be a large waste of time. Instead, a different approach was chosen. All functionality within the program that is supposed to interact with the mounting of tools, environment settings or Git server access executes an interpreted ".bat" (windows batch) or ".sh" (unix shell) script.

The specific relative paths to the desired storage locations, names and order of tools, settings for git credentials and any other desired functionality for the preparation of a team-roles work environment, are all defined in these scripts. As such the final version of the compiled program can continue to be used for its convenient user interface, while the selection or location of tools can change. Simply editing the scripts with any text editor allows changes for any deployment scenario, without recompilation.

```
// Bat execute functions (exec strings shortened for example)
void Tfrm_ESTB_main::ExecuteMount() {
    AnsiString exec = frm_ESTB_settings->bat_prefix+".. args");
    ExecuteCMD(exec, false);
}

void Tfrm_ESTB_main::ExecuteUnmount() {
    AnsiString exec = frm_ESTB_settings->bat_prefix+".. args");
    ExecuteCMD(exec, false);
}

void Tfrm_ESTB_main::ExecuteGitConf() {
    AnsiString exec = frm_ESTB_settings->bat_prefix+".. args");
    ExecuteCMD(exec, false);
}

void Tfrm_ESTB_main::ExecuteCMD(AnsiString exec, bool showWindow) {
    if (!system(NULL)) { AddStatus("Error:", "Could not access CMD.", true);
                         ShowMessage("Error: Could not access CMD.");
                         return; }

    exec = "cmd.exe /k "+exec;
    char buf[exec.Length()+1];
    strcpy(buf, exec.c_str());

    WinExec(uf, (showWindow ? SW_SHOW : SW_HIDE));
}
```

Fig. 37: Calling script files from C++ implementation

```
#>> C:\ESTB\_bat\ESTB_mount.bat
@echo off
:: Reading input variables
set DRV=%1
set DAT=%2
set ROL=%3
set DSK=%4

:: Checking drive existence
if exist %DRV%\_bat\ goto notneeded
if not exist %DRV%\nul goto domount
if exist %DRV%\* goto error

:: Mounting folder as virtual drive, call shortcut script
:domount
subst %DRV% %DAT%
powershell -Command "Start-Process cmd -WindowStyle hidden -ArgumentList '/k subst %DRV%
%DAT% && %DAT%\_bat\ESTB_shortcuts.bat %DRV% true %ROL% %DSK% && exit' -Verb RunAs"

:: Open mounted folder
if "%DSK%"=="false" (explorer.exe %DRV%)
#[..]

#>> C:\ESTB\_bat\ESTB_git_exec.sh
#!/bin/bash
DRV="$1"
REP="$2"
USR="$3"
EML="$4"
PWR="$5"
ROL="$6"

# Sanitizing input
REP=$(echo $REP | sed 's/https\?:\/\///')

# Removing Previous
rm -rf "$DRV/_rep/*"
rm -rf "$DRV/_rep/.*"

# Setting up repository
cd "$DRV/_rep"
git clone http://$USR:$PWR@$REP ./
if [ $? -eq 0 ]; then
    msg /time:2 $USERNAME "SUCCESS: Repository cloned."
else
    msg /time:10 $USERNAME "ERROR: Could not clone repository over HTTP!"
    exit 1
fi
#[..]
# Setting up remote settings
git remote rm origin
git remote add origin http://$USR:$PWR@$REP
git fetch origin "$WNT_BRANCH":"$WNT_BRANCH"
git pull origin "$WNT_BRANCH"
git push --set-upstream origin "$WNT_BRANCH"
git config --global user.name  "$USR"
git config --global user.email "$EML"
git config user.name  "$USR"
git config user.email "$EML"

if [ "$ROL" == "0" ]; then
    # Fetch all branches for programmers
    git fetch origin BRANCH_Creative:BRANCH_Creative
    git fetch origin BRANCH_Testing:BRANCH_Testing
fi
#[..]
```

Fig. 38: Example of script elements

Only the information passed to the scripts as arguments is determined at compilation time. Should additional information for the script execution be necessary, it would need to be added and recompiled, but for the purposes of the toolbox scope, no additional information should technically be needed. Administrative privileges needed for certain tasks can be acquired entirely through the scripts as well, removing the need for specification at compile time. Nested scripts, in which a windows batch script sets up the shell or bash environment for a unix script (useful for Git) are easily configurable as well.

4.3 ─ Deployment of toolchain for team-based development

To begin deploying the described toolchain implementation for use in a development scenario, the server needs to be outfitted with the provided image of the working implementation, or a custom implementation of the described elements for use in altered circumstances. After the server has been setup, the networking should be taken care of next. Ensuring that all desired development machines are connected to the same local network as the hosting server (raspberry pi or otherwise) should first be cleared up. The ability to access all of the hosted services on the server needs to be ensured as well. Depending on router and firewall settings, some ports might not be accessible on all machines. Additional equipment, such as network switches, might also be quickly evaluated before development should commence.

For the development machines available, the appropriate Microsoft Windows version should be picked and installed. As a rule of thumb, machines with 4GB of RAM (or less) should always be setup with Windows 7, as newer versions like Windows 10 and Windows 11 consume a notorious amount of memory on lower-end systems, often slowing down or stifling the pace of work to a noticeable degree.

The main data folder for the toolbox application should be placed on the root of the main C: drive. As many of the thoroughly evaluated tools that are used as part of the toolchain are paid software or simply not open-source, the toolbox data folder distributed with this thesis is not legally allowed to contain them. Instead, text files linking to either official download or purchase pages for these tools are included. The tools should simply be extracted or

installed directly into the folder referencing their link. The toolbox executable itself can be placed in the Programs Folder, then linked to the desktop, or simply put on the desktop for the duration of development.

Before development can begin, environment variables needed by the toolbox and its scripts are required to be set within Windows. As the programmatical setting of environment variables proofed to be unreliable within scripts and the C++ Builder environment, the setting of these variables has not been automated yet. Both the installation process and the setting of environment variables could be automated for a future release with the promising “Inno Setup” *(Jordan Russel, 2021)* installer creator, implementing a scripting language to define locations of files to be places and system variables to be set. For the meantime, these still have to be set manually (Settings > System > Advanced System Settings > Environment Variables).

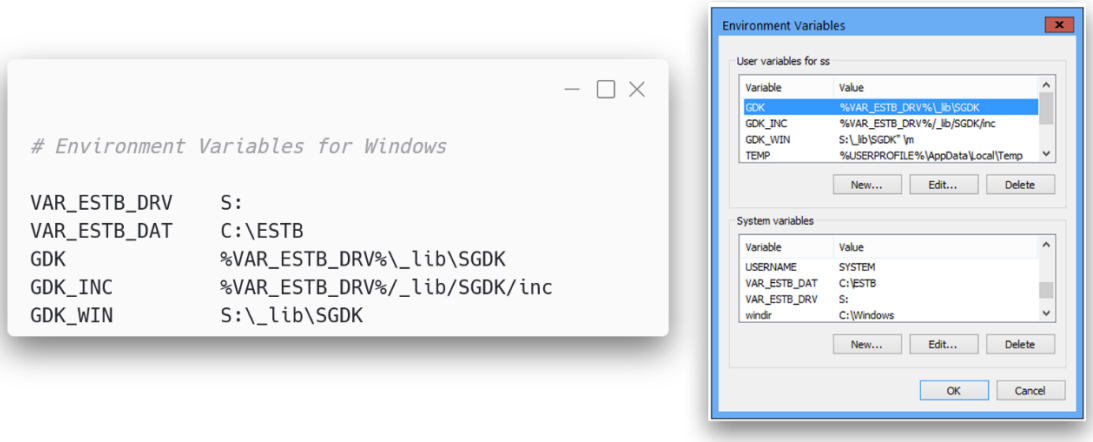


Fig. 39: Environment variables for toolbox deployment

Once this process has been done once, it is advisable to save time by cloning the finished Windows setup to other SSDs or HDDs for the rest of the development machines to boot into. This avoids having to manually place the tools and set the variables multiple times.

After all cloned operating systems are confirmed to work, the team-based development process can begin. Team members should be instructed to manage their time using the hosted focalboard and to use the hosted wiki to read up on their tools specific workflow. As the

leading developer, also make sure to clean up the repository and regularly merge all finished branch work into the master branch. Once milestones of development are hit, the testing branch should also be merged into from the master branch to initiate the server-side building and distribution of the rom file, for game testing purposes. Beyond this, the development work should be able to begin smoothly without further distractions.

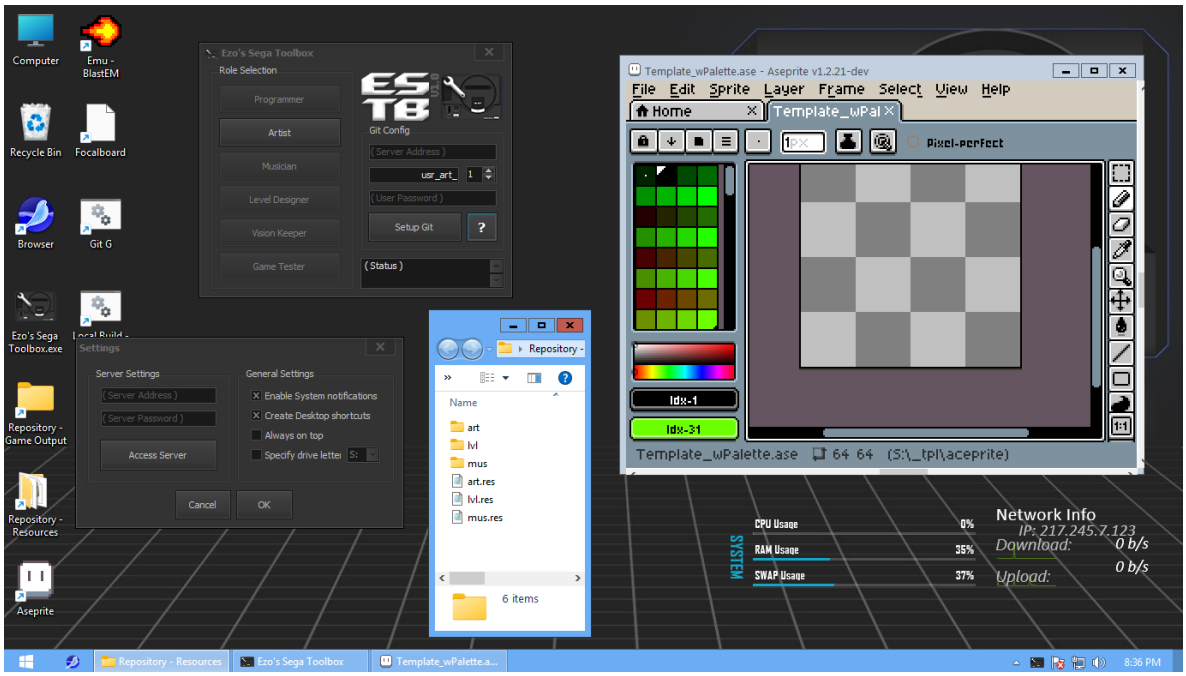


Fig. 40: Screenshot of development machine configured

5. ─ ASSESSMENT OF RESULTS

The presented execution of the toolchain implementation might be informative on its own, but it would not be conclusive, as no empirical data about its functionality would be available. A toolchain intended to be used actively by multiple people, working in a multitude of disciplines, at the same time, especially in the context of a relatively undocumented target platform, certainly needs proof of applicability. Without a clear result on the adequacy of the conducted work, the intended educational findings on the subject would remain rather mute.

This begs the question how both the devised toolchain itself, as well as the usability of the tools within, could be put to the test in a scientifically relevant manner. While more thorough evaluation of the toolchain would be possible, constructing a small "Game Jam" event around its usage seemed to result in adequate findings. This event entails assembling a team of individuals apt at their respective discipline (art, music, level design, programming ..) with little to no prior knowledge of the SEGA Mega Drive console or the specific development tools and programming framework to be used. These individuals are then presented the subject via a short presentation, establishing the history of the console, as well as the limitations imposed by the old hardware.

Given access to development machines (Windows laptops) with the installed toolbox software and access to the described development server with all elements of the toolchain implemented, the first development session took place. Within a timeframe of roughly 12 hours, a small Mega Drive game was developed by this previously rather uninitiated team. After development ceased, surveys were filled in by all participants, collecting important data about their expectations, experience and satisfaction with the result. The game developed as part of this event is equally included and discussed within the following sections.

5.1 ─ Team constellation & expectations

The individual team members who took part in this game jam event were all comprised of personal friends of the author, partly with semi-related professions or aspirations in technical

or creative fields. Being all avid gamers, the willingness to help generate findings for this thesis, by way of jumping into retro game development, was expressed by all.

Beyond being interested in video games in their spare time, the individual professions and assigned roles during the game jam event are of essence to later evaluate the gathered results from each team member. As programming is quite frankly the discipline needing the most effort to begin constructive work for most game projects, programming was to be handled by the author first, with one team-member serving as the secondary programmer. This was done mainly to ensure a certain quality of code and implementation, as to not detract from the work conducted by other team members. Taking the role of second programmer, Timo Riber joined the team, bringing considerable experience in programming through his studies in cyber-security.

For art, Alvin Hölzenbein and Mark Behn joined the team. The former being a long time gamer and twitch streamer, pursuing a career in youth tutoring and the latter being an engineering student with previous experience in pixel-art. The two are close friends in particular, with frequent creative projects between them. Bouncing ideas for visuals off of one each other is typically a strength in such a befriended art team.

Gabriel Rother, a systems integrator currently in training, has had previous experience with the creation of game mods and custom maps for various first-person games. With this experience and general technological savviness, the role of level designer seemed to fit rather well. As all team members knew one another, the important communication between for instance the level designer and the previously discussed duo of artists was open and not restricted through personal unfamiliarity, intended to lead to a more creatively free environment.

For music and sound, Felix Lucchesi and Aaron Schnee joined the team. Both being avid hobbyist musicians, respectively in electronic and rock music, their differing technical views on music production was intended to provide multiple angles to tackle the rather restrictive sound environment for the SEGA Mega Drive.

Lastly, Noah Böckling rounded out the team. Being an active student of psychology, his perception of the of the other team members and their styles of work seemed like an

interesting factor to consider. As such he received the role of vision keeper, mainly observing the creative process of all individuals and managing the time taken for the various tasks of all disciplines. The management on his part was carried out through use of the described focalboard task tracking tool in particular.

As most team members brought important skills and motivation to the table, the general expectation towards the development process and potential result were positive. All disciplines were comprised of individuals decently apt at their craft, leaving negative expectations stacked rather against the ability to communicate workflows and intended collaboration models effectively.

On one hand, the rather large pool of functions provided by the SGDK framework and the general importance of hardware understanding still in place for SEGA Mega Drive development, lead to slim expectations for collaboration with the second programmer. Even though the toolbox software was designed around ease of use, the relatively clunky process of git access through a manually typed URL, lead to expectations of frustrations being voiced about the tool. One other expected concern was the artists ability to work with the restrictive 4x16-Color palette limitation in place for the SEGA Mega Drive. Beyond just working with this palette, the importance to export in indexed color mode, with consistent colors across all artwork, also lead to concerns.

Nonetheless, working with an excitable group of befriended gamers and creatives was sure to be a rewarding and hopefully conclusive experience about the work conducted during implementation of the toolchain.

5.2 ─ Game jam event, resulting product & experiences of team members

On the fifth of February 2022 the game jam took place. With participants arriving at roughly noon, the presentation of the event, selected games to be played to gain familiarity with the console and getting set up with the provided development machines, was taken care of. After a baseline understanding of the historically popular and technically possible caliber of games

on the SEGA Mega Drive was established to the team, a brainstorming session was launched, discussing the genre, goal and subject matter of the game to be developed.

Quickly, the majority of participants voiced their interest in a run-and-gun style platforming game. While certainly a more complex type of game, compared to other genres, the technical foundation both of programming knowledge and flexibility of the SGDK based toolchain was deemed up to the task. The game itself was to feature a cyborg hunter as a main character, hunting various mutated wildlife out in nature. A futuristic laser gun was decided to be his arsenal of choice. Beyond these points, it was decided not to dive too deep into the specifics, rather figuring out the process and path forward as development began.

From roughly 02:00 PM to 11:30 PM, the game jam was conducted, with additional clean up and bug fixes a few hours later. During the preparation for the event, multiple desks were setup in the available space to act as development stations for the individual disciplines. A separate table was used as the networking center and a place for the original console to be stationed. The networking infrastructure was comprised of a managed network switch, attaching all development client machines and the raspberry pi server to the same local network.

Fig. 41: Networking and console setup for game jam

Throughout the experience, multiple automated builds through the servers testing branch were copied to the available flash cartridge for testing. Each time a new build was tested, the whole team gathered rather excitedly around the CRT TV and grabbed a controller. These testing interludes noticeably upped the team morale and reinstated motivation for the next tasks to be worked on.

Coming back to the expectations laid out during the previous section, the event certainly proved some right, but also lead to a few surprising and positives discoveries as well. Firstly, the expectation to spend a lot of time communicating the workflows and aiding in technical support did come true, despite the best efforts with the hosted wiki. Once on track, with most questions answered, surprisingly quick progress in most disciplines was made however. Programming collaboratively also came close to original expectations. Implementing the majority oft the game, while also aiding in technical issues, left less than desirable time to instruct the second programmer on the team. Despite this, his experience with programming allowed him to pick up the pieces by reading the wiki and the actively written code for the project. In the end, the basic logic for title screen and intro text display were indeed implemented by him, without additional help required.

Some unexpected lessons were also learned from the event however. The expected struggle of the art team with the restrictive color workflow remained minor. Dividing all their color choices into 4 palettes of 16 colors, utilizing the first color as transparency for their art, and exporting all tiles in indexed color format, all were done admirably after first instruction. The communication and exchange between artists and the designated level designer lacked however, with tile-sheets being delayed in favor of other tasks, often leaving the level designer without new elements to implement into the level. Perhaps most surprisingly, the tracker style music production approach turned out to be a major problem for both designated musicians. Even with concise tutorials and personal help, both the modern DAW experience and the general analog musical prowess shared between both musicians, failed to breach the restrictive nature of pure FM synthesis and the tracker interface. Despite this, three simple tracks for the title screen, gameplay and win condition were produced and implemented.

Lastly one more surprising element was the repository management. With frequent merges resulting in small merge-conflicts, a bit of time was wasted resolving and cleaning up the branches multiple times. A different style of branch architecture, possibly even a simple “Trunk” style single branch approach could have suited the rather simple nature of SGDK projects better.

After the dust had settled, the final game rom was tested on the previously pictured SEGA Mega Drive console. Despite initial hurdles, the game which was able to be produced through usage of the implemented toolchain stuck rather close to the original vision, also thanks in part to the adequate time management provided by the designated vision keeper. Beyond just the games features, the artwork produced and the simple music to express them were also adequate. The difference in art-style between tiles and sprites and the simplicity of the music certainly speak for the experimental nature of the project, but represent a clear road forward for what could be creatively expressed given more time and experience. The quality of written code and stability of the game logic was also a positive surprise, considering the amount of time spent explaining and aiding the other team members, along with additional repository cleanup duty.

After a few runs of the game and a collective high-five, the game jam was concluded. The final state of the game jam repository and the compiled game rom file is included within the file structure of the enclosed USB storage medium. Alternatively, download links to all produced elements as part of this thesis are also listed in the table of references.

Fig. 42: Game project achieved during game jam

Conversing with the team members after filling out their surveys, an overall appreciation of the event and the resulting product was voiced. While the second programmer, artists and the level designers were very satisfied with seeing their work in palyable fashion, the musicians in particular voiced dissatisfaction with their provided efforts and the difficulties faced with the chosen music tool.

The survey results were categorized and used as part of the following section on applicability. Additional testimony of team members, filled out as part of the survey, can equally be accessed within the enclosed USB storage medium or via the download links.

5.3 – Applicability of solution to the proposed problem

To form conclusive analysis on the applicability of the implemented toolchain, used during the game jam event, the valuable survey data was structured into a table and subsequently visualized with a diagram for visual inspection. The survey was structured with three main parts, each concerning themselves with a different quality of the provided elements ant the project which was conducted with them.

Survey Results collected during SEGA Mega Drive Game Jam (05.02.2022)

Collected results:

Participant	Role	Usability: Toolbox	Usability: Repository	Usability: Tools	Quality: Collaboration	Quality: Wiki	Quality: Time Management	Satisfaction: Own Work	Satisfaction: Results	Satisfaction: Experience
Aaron Schnee	Musician	3	5	2	4	3	3	4	4	5
Alvin Hölzenbein	Artist	4	3	4	5	4	5	5	5	5
Felix Lucchesi	Musician	4	2	3	4	4	3	4	4	4
Gabriel Rother	Level Designer	5	5	5	4	5	3	5	4	4
Noah Böckling	Vision Keeper	3	4	5	5	4	2	4	4	3
Mark Behn	Artist	5	5	5	4	5	5	5	5	5
Timo Riber	Programmer	5	5	4	3	5	2	2	3	3
Average results :	Programmer	5	5	4	3	5	2	2	3	3
Average results :	Level Designer	5	5	5	4	5	3	5	4	4
Average results :	Artist	4.5	4	4.5	4.5	4.5	5	5	5	5
Average results :	Musician	3.5	3.5	2.5	4	3.5	3	4	4	4.5
Average results :	Vision Keeper	3	4	5	5	4	2	4	4	3

Fig. 43: Table survey results

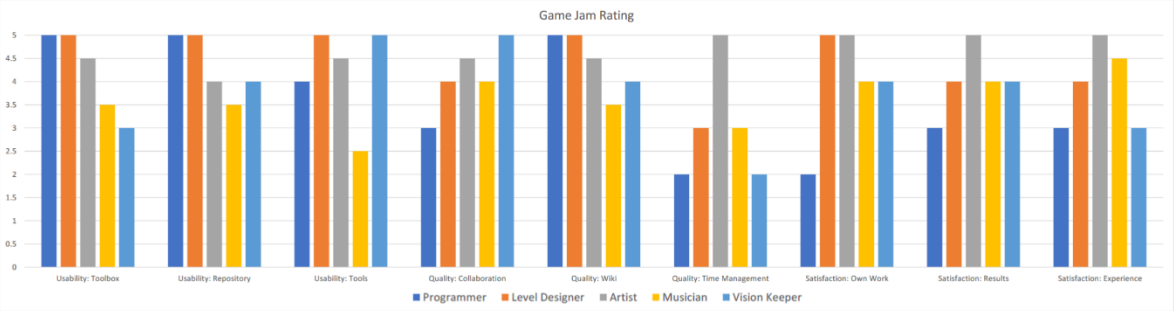


Fig. 44: Diagram survey results

Firstly, the usability of the toolbox application, the repository access and the tools relevant to the respective discipline were called into question. The different viewpoints of the participants, based on their team role, can be observed and compared as such. Unsurprisingly the secondary programmer found the usability of the toolbox and repository access to be very positive, as working with such tools and version control systems is a common task in his professional environment as well. The usability of the programming tool, Visual Studio Code, was ranked slightly lower, as a fully-fledged IDE might have suited his usual style of development better. Artists found the toolbox and their designated art tool very usable as well, struggling slightly more with the usage of the repository. As mentioned, the musicians struggled harder with their tasks, largely because of the nature of the designated music production tool. As such, the usability of the tool faired poorly in their regard, while the usage of the toolbox and repository evened out to a roughly mediocre rating. Lastly for this part, the vision keeper found the usage of the task & time tracking tool to be very usable, while struggling with the repository and the toolbox a bit more, especially during initial setup.

The second part of questioning concerned itself with the perceived quality of collaboration, information acquisition through the hosted wiki and the time management during development. As described, the collaboration was rather meager for the second programmer, as frequent issues demanded time that could have helped him gain a better understanding of the SGDK framework. The level designer had a slightly more positive view of collaboration, frequently interacting with the artists, but did fall short from a perfect rating for the waiting periods describe previously. The vision keeper collaborated admirably and frequently communicated with all team members, rating the interaction highly. The remainder of the quality ratings expressed a very positive view on the provided wiki and a noticeably negative or mediocre view on the time management. Despite the vision keepers best efforts, the inability to judge time, when utilizing new tools did create a slippery slope and feature creep towards the end of development, resulting in lower ratings.

The last of the questioned sections revolved around the topic of personal satisfaction with the individual work conducted, the final results and the general view of the experience. While the level designer, artists, vision keeper and even the previously somewhat disappointed musicians were all quite satisfied with their personal work, the second programmer expressed disappointment with the amount of implementation possible to learn and execute during the event. The adjacent ratings for satisfaction with the results and overall experience ended up very positive, all things considered. Despite the understandable frustration of the secondary programmer, mainly due to limited time, all other team members enjoyed the resulting game and the experience along the way, rating it from 4 to 5 out of a highest possible rating of 5. The stressful nature of keeping time in check for all team members seemed to degrade the vision keepers experience rating down to a rating of 3, same as the programmer.

Now that the resulting game project and the survey results have been presented and interpreted, it is time to shift focus back onto the initially proposed problem. The feasibility of implementing a team-based toolchain to facilitate collaborative development on a platform as restrictive and specialized as the SEGA Mega Drive was no easy question to answer. Especially considering the vast differences in approaches, tool selection, team constellation and work environment possible, a completely universal solution is hard to implement, if possible at all. The specific implementation created as part of this thesis had to concretely

settle on these decisions to arrive at a usable result. Oriented towards low-cost deployment with an accessible raspberry pi computer, large focus on good onboarding quality with the development of the toolbox and simple implementation of build-automation, the presented implementation fulfills most of what it set out to achieve.

Despite the described multitude of hick-ups and unexpected hurdles, the development of the final game and quality of collaboration possible during it, achieved with the present implementation, certainly proves the capability of the chosen tools, their interaction and the attainable effectiveness of the server model and toolchain. As such, the provided implementation can stand as an example for a possible solution to the proposed problem of collaborative SEGA Mega Drive development. Obviously not perfected yet by any means, many elements of the toolchain could and should be scrutinized, replaced and experimented on to gain even more understanding of the possible effectiveness different elements can have for such a solution. Which elements in particular should be investigated for improvements, what direction the improvements might be taken in and what the work conducted as part of this thesis brings to the larger table of game development and education, will all be discussed in detail in the following last chapter.

6. — PERSPECTIVE ON FUTURE ADVANCEMENTS

Turning the focus finally to a bit of reflection, the presented implementation of a toolchain for SEGA Mega Drive development is one of many possibly valid ones. While some presented elements turned out to work rather well when put in practice, other elements certainly could have benefited from a different approach, or more in-depth work. As mentioned, the wiki and general onboarding process for uninitiated team-members certainly could have needed more careful thought. Overall, the rather positively identified result shows the potential this approach can have for the proposed problem of creating a universal modern toolchain for the SEGA Mega Drive platform.

This desire for universality however presents the most thrilling aspect of anticipating future advancements to be made. Different development intentions, team constellations, game genres and tool familiarities can all play equal roles in changing the applicability of the chosen elements for this toolchain. Creating a universally applicable model for such a toolchain, capable to be integrated seamlessly with a wider variety of tools and deployment procedures would lower the barrier of entry to whole teams of interested developers even more. The provided implementation certainly illustrates the possibilities for team-based work in this restrictive context, but has more room to grow, hopefully serving as a stepping stone for other developers to base their SEGA Mega Drive toolchains and development setups on.

However the case may be, the main driving force behind this undertaking, making the entire process possible in the first place, is the SGDK framework by Stephane Dallongeville. The amount of exciting new projects for the console stemming from this framework alone is astounding, having possibly reached a critical mass as to forever changing the landscape of SEGA Mega Drive homebrew development. The ability to develop games for such a beloved platform in such a comfortable environment as modern C, allows a wide variety of creative individuals with a love for games to create something authentic and substantial feeling.

This profound feeling of having created, what could have been, a game form a time long gone, is remarkable. The motivational effect of this could and should absolutely be leveraged for educational contexts in basic computer science and game development studies. With appreciation and nostalgia for pop-culture and media, maybe not even personally

experienced, being so prevalent nowadays, that feeling of having created something authentic is to be cherished.

6.1 — Further improvements & adaption

Reflecting a bit more on the specific elements that could benefited the most from additional improvements could be a good entry point for interested developers to begin working on. Firstly, the server setup created as part of the presented toolchain is functional, but rather plain. Beyond the simplicity of many of the implemented server-side elements, the security is of higher concern. Having been setup with complete local development in mind, the security for server access and general protection against outside influences is lacking considerably. Especially in current times, in which in-person meeting often is not a possibility for health concerns, an adaption and improvement of this server model would be important. Either through self-hosting on a local machine with port-forwarding, or directly cloud-hosted using a service such as “Digital Ocean” or “Linode”, a true online server to centralize development efforts to people or groups working remotely would be an important step. The firewalls, encryption key setups and general hardening of the server environment would have to be evaluated, implemented and tested to gain more understanding.

The overall management of the server would be another point to be improved. The git repositories in particular are all manually created via a secure shell session. Scripts to be remotely executed could be written, automating the process of backing up repositories and preparing new repositories to be developed on. Generally consolidating long chains of commands to manage parts of the server into easily usable scripts could up the usability of the toolchain even more.

Efforts to improve the rather simplistic build automation present in the current solution would also be an important step. Possibly integrating this building and distribution process with an industry-standard solution for iterative or continuous integration, such as Circle-CI, would bring more professionalism to the table. The concurrent building of multiple branches, not just the testing branch, could be a welcome outcome of this improvement as well, possibly

even resulting in a better branching methodology to be found for SEGA Mega Drive development.

Lastly, the toolbox program could also see some improvements. To move to a true cross-platform approach, porting to an open-source user interface framework would be a great start. Additionally, creating installation programs or scripts with which to place all needed files correctly and setting up the environment variables automatically would be more professional and could save much time when setting up development machines.

Beyond these points, the tool selection is of course subject to wild experimentation and personal preference for all interested developers. The presented tools however are a great starting point, representing commonly used tools within the homebrew scene for SEGA Mega Drive, with tried-and-true examples in released games. If other tools work better for a given team or individual however, those should be the clear choice in that circumstance.

6.2 — Closing remarks & acknowledgements

To end this last chapter on an appreciative note, the process of diving deep into such a largely undocumented environment has been quite exhilarating and satisfying. Facing the challenge to adapt all of these amazing tools, frameworks, scripts and programs into a whole usable system with which to develop games collaboratively was rewarding and hopefully provided stimulating results. Especially considering that the platform for which all of these efforts have been made, the SEGA Mega Drive, is such a rightfully beloved system, makes the conducted work also beneficial to the continued creative interest in a piece of gaming history.

Professor Dr. Martin Leissler, having both presented the possibility of a bachelor thesis on this subject and acted as the main project supervisor, deserves much appreciation for his continued support and positive, constructive feedback throughout the development of this thesis and the related work. Acting as the second supervisor, Thomas Nickel also helped tremendously with his deep knowledge of the SEGA Mega Drive platform and both the historic and current market around its games and projects. The highest level of gratitude is extended especially to both of the thesis supervisors and the credited individuals from the

homebrew development scene, having offered much appreciated help and support throughout this journey.

LIST OF FIGURES

LIST OF REFERENCES

Alex Hope. (2013) "The Evolution of the Electronic Sports Entertainment Industry and its Popularity"

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.867.2936&rep=rep1&type=pdf#page=88

Brent Cowan; Bill Kapralos. (2014) "A Survey of Frameworks and Game Engines for Serious Game Development"

https://ieeexplore.ieee.org/abstract/document/6901570

Thomas Bryant David. (2017) "New Retro: An Exploration of Modern Video Games With A Retro Aesthetic"

http://rave.ohiolink.edu/etdc/view?acc_num=miami1493401505332341

Jack England. (2021) "Pixel Art and the Retro Game Revival / A Connection to the Past for a New Generation"

https://bitcade.co.uk/blogs/news/pixel-art-and-the-retro-game-revival

Andrew Higson. (2013) "Nostalgia is not what it used to be: heritage films, nostalgia websites and contemporary consumers"

https://www.tandfonline.com/doi/abs/10.1080/10253866.2013.776305

Melanie Swalwell. (2021) "Homebrew Gaming and the Beginnings of Vernacular Digitality"

https://books.google.de/books?hl=en&lr=&id=3HQJEAAAQBAJ&oi=fnd&pg=PR11&dq=homebrew+game+development&ots=bXMvb7d9hf&sig=kpoBXzxFoTSM7G5NAJEKWU0Ssj8&redir_esc=y#v=onepage&q=homebrew%20game%20development&f=false

Retro Reversing. (2020) "Sega Mega Drive (Genesis) Development Kit Hardware"

https://www.retroreversing.com/sega-mega-drive-genesis-development-kit/

Neurosys. (2020) "Why People Say Unity Engine Is Bad & What Is It Good For?"

https://neurosys.com/why-people-say-unity-engine-is-bad

Lexaloffle Games Ltd. (2015) "PICO-8 Website"

https://www.lexaloffle.com/pico-8.php

Ashley Haugen. (2021) "Homebrew Computer Club: Everything You Need to Know"

https://history-computer.com/homebrew-computer-club/

Copmuter History Museum. (2020) "Timeline of Computer History - 1980s"

https://www.computerhistory.org/timeline/1980/

Sega Retro. (2021) "List of Sega Mega Drive homebrew games"

https://segaretro.org/List_of_Sega_Mega_Drive_homebrew_games

Titan. (2013) "Overdrive"

https://archive.org/details/demo_titan_overdrive_rev1.1-106-final

Sarang. (2010) "Space Pixy"

https://www.dcemu.co.uk/vbulletin/threads/321816-Mega-Drive-Space-Pixy

Matt Phillips / Big Evil Corporation. (2019) “Tanglewood"

https://github.com/BigEvilCorporation/Beehive

https://github.com/BigEvilCorporation/TANGLEWOOD

https://uk.linkedin.com/in/mattphillips1

https://www.youtube.com/watch?v=gNHCxH35wYU

Stephane Dallongeville. (2022) "SGDK - A free and open development kit for the Sega Mega Drive "

https://github.com/Stephane-D

http://icy.bioimageanalysis.org/

Kubilus1. (2015) "GENDEV - Genesis development environment for Linux."

https://github.com/kubilus1/gendev

Andre DeRosier. (2017) "Marsdev. Cross platform Mega Drive / 32X toolchain and Make-file abuse."

https://github.com/andwn/marsdev

Zerasul. (2020) "Genesis Code extension for Visual Studio Code"

https://marketplace.visualstudio.com/items?itemName=zerasul.genesis-code

TigerNT. (2009) "68000 Instruction Set"

http://www.tigernt.com/onlineDoc/68000.pdf

Steven Weber. (2004) "The Success of Open Source"

https://books.google.de/books?hl=en&lr=&id=ELieXMxR1h4C&oi=fnd&pg=PP15&dq=open-source+code&ots=0rxI2FHAKG&sig=koxiNtTuoTy8OPbHJHtdh_x5PHw&redir_esc=y#v=onepage&q=open-source%20code&f=false

Raspberry Pi Foundation. (2022)

https://www.raspberrypi.org/

Mattermost Inc. (2021) "Focalboard: Project Tracking"

https://www.focalboard.com/

Atlassian Corporation. (2011) "Trello"

https://trello.com/en

Second Dimension. (2022) "Second Dimension Home Page"

https://www.sbasic.net/

Hugues Johnson. (2022) "Sega Genesis Programming"

https://huguesjohnson.com/

https://huguesjohnson.com/programming/genesis/palettes/

Doragasu. (2022) "Docker-SGDK", "Docker-Deb-M68K"

https://gitlab.com/doragasu

https://gitlab.com/doragasu/docker-sgdk

https://gitlab.com/doragasu/docker-deb-m68k

Codeblocks IDE (2020) "The free C/C++ and Fortran IDE"

https://www.codeblocks.org/

Qt. (2022) "Qt Creator - A Cross-platform IDE for Application Development"

https://www.qt.io/product/development-tools

Eclipse Foundation. (2022) "Eclipse IDE for C/C++ Developers"

https://www.eclipse.org/downloads/packages/release/kepler/sr2/eclipse-ide-cc-developers

Microsoft. (2022) "Code editing. Redefined."

https://code.visualstudio.com/

Chez. (2001) "GrafX2 is a bitmap paint program inspired by the Amiga programs Deluxe Paint and Brilliance."

http://grafx2.chez.com/

Piskelapp. (2022)

https://www.piskelapp.com/

Bryan Ware. (2022)

https://www.pixilart.com/about

Igara Studio. (2022) "Animated Sprite Editor & Pixel Art Tool"

https://www.aseprite.org/

https://www.igarastudio.com/about/

Tenacity Team. (2022)

https://tenacityaudio.org/

https://github.com/tenacityteam/tenacity

Tom Nardi. (2021) "Muse Group Continues Tone Deaf Handling Of Audacity"

https://hackaday.com/2021/07/13/muse-group-continues-tone-deaf-handling-of-audacity/

Video Game Music Preservation Foundation (2020) "GEMS"

http://www.vgmpf.com/Wiki/index.php?title=GEMS

Shiru. (2011) "VGM Music Maker v1.1"

https://shiru.untergrund.net/

Leonardo Demartino. (2011) "DefleMask. The best chiptune tracker"

https://www.deflemask.com/

https://demartino.ar/

Sébastien Bénard. (2018) "An open-source 2D level editor from the creator of Dead Cells"

https://ldtk.io/

https://deepnight.net/about/

Under-Prog. (2021)

https://under-prog.ru/en/

https://under-prog.ru/en/ldtk-guide-to-the-best-level-editor/

https://under-prog.ru/en/sgdk-creating-a-platformer-for-sega-genesis/

https://vk.com/underprogru

Steve Palmer. (2010)

https://segaretro.org/Steve_Snake

https://segaretro.org/Kega_Fusion

Kaneda. (2006) "An advanced mod for Gens emulator for hacker, ripper and...coder"

https://gendev.spritesmind.net/page-tools.html

Michael Pavone. (2019) "BlastEm - The fast and accurate Genesis emulator"

https://www.retrodev.com/blastem/

https://twitter.com/mikepavone

Various mentioned image writing software

https://rufus.ie/en/

https://www.balena.io/etcher/

https://sourceforge.net/projects/win32diskimager/

Git Version Control. (2022)

https://git-scm.com/

https://git-scm.com/book/en/v2/Git-on-the-Server-Setting-Up-the-Server

https://git-scm.com/book/en/v2/Git-on-the-Server-The-Protocols

Perforce. (2022) "Perforce Version Control"

https://www.perforce.com/products/helix-core

Apache Project. (2021) "The Number One HTTP Server On The Internet"

https://httpd.apache.org/

Nginx Inc. (2021)

https://www.nginx.com/

Embarcadero Technologies Inc. (2020) "C++ Builder CE 10.4"

https://www.embarcadero.com/de/

https://www.embarcadero.com/de/products/cbuilder/

https://www.embarcadero.com/de/products/rad-studio

Jordan Russel. (2021) "Inno Setup"

https://jrsoftware.org/isinfo.php

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbmd
FACHBEREICH MEDIA
(Faculty of Media)

Bachelor of Arts – Animation and Game

Performance of high-level language frameworks for 16-Bit console game development

Research Paper

Student: Hölzel, Luca | 756566

First supervisor: Prof. Dr. Martin Leissler

Second supervisor: Thomas Nickel

Date of Submission: 24.01.2022

TABLE OF CONTENTS

ABSTRACT

Homebrew video game development for the beloved 16-Bit home consoles of the early 1990s is a largely undocumented, unintuitive and to this day almost unheard-of form of form of video game development, especially in a professional environment. Originally programmed with restrictive assembly languages, operating incredibly close to the hardware level, the barrier of entry to develop for real 16-Bit consoles was incredibly high. With many up-incoming frameworks and tools aimed to simplify the development of games for such systems, the question emerges if such abstracted tools could create games performant enough on very limited hardware, compared to the restrictive yet performant original assembly languages.

This research paper concerns itself with the process of learning video game development for the SEGA Mega Drive home console, by studying the hardware and learning both the original 68000 assembly language and development using the popular SGDK (Sega Genesis Development Kit) by Stephane Dallongeville. A benchmark program for the platform is implemented, highlighting the approaches for both development methods, their strong suits, as well as their individual shortcomings. The resulting performance data is gathered from original hardware and emulation software alike and subsequently visualized and compared. The findings reached as part of this research paper indicate both positive and negative implications for the performance of both approaches and the feasibility of learning the assembly programming approach in the modern day, from a student's perspective.

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

Background

If one looks back over the pop-cultural journey the media landscape has been subject to since its inception, they will surely attest to the constantly changing yet often cyclic nature that its path takes. Many forms of art and technology alike which we are now inclined to view as an important part of our daily lives and even generational history, had to undergo various challenging phases to reach their current level of importance and ubiquity in larger society.

Amongst these art forms and inventions, it is the perception and place which video games hold that has perhaps experienced one of the most dramatic shifts. Jumping from the general societal perception of a children's toy, which some adults seemed to obsess over, to the fastest growing, highest earning and most universal form of entertainment currently available, is not just unheard-of, but completely unprecedented in any other form of media to date. Modern gaming has forged paths previously relegated to the realm of science fiction, with massive multiplayer worlds to be explored and readily available virtual reality devices, coming close to redefining what the definition of a video game truly is, for years to come.

With all of this rapid progress in user experience, graphical fidelity and networking bringing forth a whole new approach to gaming, curiously a new trend slowly came up at the beginning of the last decade. Instead of focusing on all the new possibilities afforded by current technology, many players began developing nostalgia for the games they grew up with. Partly driven by human nature of reminiscing about a simpler time in life, partly by the voluminous changes to the original formulas behind the simple, yet fun games of the past, old video games started peaking in popularity once more. As classic games started receiving more attention, programmers, artists and musicians involved with the gaming industry or the solitary pursuit of game creation began taking note. The result was the first indie-game boom, creating a wave of various retro-revival games, celebrating the limitations of the systems of yesteryear and wearing them brightly on their sleeve, in the form of pixel-art, chip-tune music and a focus on tighter, simpler game loops.

Just like with many forms of media and other areas such as fashion, the cyclic revival nature had seemingly hit the video game market overnight. These games were viewed by many to

be a breath of fresh air in an industry which was starting to become much more disconnected from its roots.

Some individuals however started taking the retro-revival phenomenon more serious than others. The accuracy to the limitations of the past was secondary for most, as the visual style was more so a means to an end. Wanting to deliver on peoples nostalgic cravings and allowing for older, less casual-friendly game-design decisions to find a way to a new audience.

Others however did not want to imitate the past for the sake of creative expression. Some wanted not to imitate at all, but rather create original new games for the video game platforms they grew up with. This process of developing new games, designed from the ground up to run on the original hardware, with all limitations accounted for, became known as "homebrew" *(GamingHistorian101, 2012)*. The difference in specific programming skills required, knowledge of old hardware as well as the shier desire to not imitate limitation, but work through it to create something new, split the homebrew community far apart from most other groups developing retro-revival games at that time.

As the years went on many assumed the pixel art style and general retro aesthetic popularized by games from this time period would lose steam after a while, entering the ongoing history of video games, as just a fad. Instead this style has become a main-stay in not just the video game industry but our collective understanding of gaming as a whole. Beyond that, the desire to overcome limitations and develop new games authentically for old systems has increased as well, driving many new interested developers into homebrew communities and researching about the technical workings of old game consoles.

Very quickly new developers who find themselves in a position to develop such games will be confronted by difficulties associated with these limitations. From setting up an environment to begin development, to understanding the underlying components and their interaction, there are many things to take into account before a game can be developed given these circumstances. Most notably, developers familiar with modern programming languages will be faced with the processor specific, low-level languages (LLL), originally required to develop on most systems from the 1980s to the mid 1990s.

Over the past decade in particular, the ever increasing desire of modern developers to join in on the development of homebrew games has lead to the creation of many specialized frameworks utilizing more familiar high-level languages (HLL) such as C or Python to aid in more approachable development for these systems. These frameworks have done an admirable job of allowing a wider field of interested creators to take part in this style of development, with only minimal knowledge of hardware and language specific information. While these new frameworks have good intentions, some members of homebrew communities have voiced concern about new developers relying on them too much, neglecting their own research into the inner workings of the systems they want to develop for. Many frustrated members of these communities have since cultivated and elitist attitude towards homebrew development, often unfairly dismissing new frameworks and projects or individuals using them.

To begin with, this research paper and the directly related thesis will be covering the development for one system in particular to document specific data. The system chosen here is the Mega Drive® (or Genesis®) by SEGA® Enterprise Ltd., released originally in the fall of 1988. The reason for this choice was on one hand the interest of my professors, on the other the previously lesser documented nature of this console, compared to most popular consoles from other competitors at that time.

Problem

Even with the modern-day advancements such as frameworks in high-level languages for development and various art and music tools to ease all forms of creation for such limited hardware, the performance and memory usage is still of utmost importance in such a situation. While the related thesis covers using these modern tools as part of a team-based workflow and toolchain, one can not deny the importance of understanding the architecture of the hardware worked on and the low-level language originally intended to be used for development tasks.

Like with virtually all abstractions in computer science, overhead and inefficiencies are bound to creep in. Frameworks which aim to ease development will have to make many assumptions about what it is the user wants to create or achieve. These tasks are then

compartmentalized into a logical structure of functionality. By assuming these tasks, many optimizations for more specific implementations will be left out, potentially slowing the high-level language code down to a notable degree. That being said coming from a background of modern programming, as the target audience for such frameworks will be, the prospect of learning a low-level language simply to implement one game or finish a singular project does seem daunting. Learning a very different, much more restrictive way of implementing a logical flow for a game or application is quite the challenge and can lead to more inefficient code, if one is not already familiar with the language. This leads to a strange situation in which interested developers might want to learn the lower-level way of development, but might not have the required time to truly get proficient at this skill, as part of their projects time constraints.

As such the questions to be asked should not be purely about the raw possible performance achievable with both approaches. These results would most likely duplicate existing information in the field of computer science and also would not be emblematic of the real achievable performance of a developer beginning their journey into the development of homebrew games. Comparing the performance of, relatively familiar, high-level frameworks to the quality of low-level code feasibly implementable by a person trained in modern development tasks, should paint a much more accurate picture of what an aspiring team or individual could hope to achieve.

Motivation

The reasons and motivations for such a paper are certainly important to discuss. After all, what can truly be said about development for hardware that has been on the market for close to 35 years? The main goals this paper and the associated thesis have are both of preservation and reimagination of development challenges in the gaming industry. Casting ones mind back to a time in which almost every computer system was entirely different, the market was fragmented and development paradigms were truly grass roots, is difficult. These limitations and hardware specifics were a fact of life and posed many challenges that helped developers grow in their craft, rather than relying on the portability and compatibility of modern languages and target platforms. At the end of the day, programming will always be an act of problem

solving, overcoming limitations. As such, taking on the challenges of yesteryear has the potential to teach us more about where our modern solutions and approaches come from.

Beyond the philosophical, this paper in particular will describe the differences in game development and the resulting performance on the SEGA Mega Drive platform, using both high- and low-level implementations. Throughout the conducted research and evaluation, this paper aims to clear up many important technical questions, which beginning homebrew game developers will have about the SEGA Mega Drive console and by extension other target platforms from the time period.

STATE OF THE ART

State of Mega Drive Development

68000 Low-Level Development

Before putting together a development environment, we first need to observe the current state of development tools and practices common amongst the homebrew scene of the particular system to be worked on. It tends to be good practice to refrain from investigating the newest innovations and projects immediately, instead one should begin by inspecting the older standards and how they apply to current development styles. This would include informing oneself quite closely about the hardware specifications of the system as well, which are outlined in much more detail as part of the main thesis and the related wiki component. Once questions about the systems hardware, structure and capabilities have been sufficiently answered, the low-level language and closely hardware related development should come into focus next.

The Motorola 68000 CPU used as the SEGA Mega Drive's main processor follows the specifications of the 68000 Instruction set *(TigerNT, 2009).* This set of CPU instructions is used to program all parts of the processor's jurisdiction. The process of development using abstractions for direct processor tasks and registers will be unfamiliar to most programmers used to modern languages. In essence, each line of written code will call one of the processors instructions (commands) and typically give two parameters (operands) to be used when carrying out the instruction. These parameters can come in the form of immediate values, data registers or address registers. The differences and use cases of these will be explained later in the implementation chapter of this paper.

Being a CISC (Complex Instruction Set Computer) CPU, the 68000 also has a much larger range of instructions, which can be useful for specific tasks, while more cumbersome to learn. In contrast, many 8-Bit systems preceding the SEGA Mega Drive console, used the venerable 6502 CPU, a RISC (Reduced Instruction Set Computer) processor, which had much fewer commands, at the expense of longer code necessary for many tasks.

Regardless of the CPU one will find themselves developing for, every processor has an instruction set which can be used to program the processor directly. These languages are

referred to as “assembly”, as the programs converting them to executables, “assemblers”, quite literally assemble the more human readable instructions, down into byte-code which the CPU can execute directly. In contrast to the compilers found in higher-level languages, these assemblers translate each line of assembly code directly to the specified instruction with minimal additional operations applied, whereas compilers try to find the most optimal set of instructions to express the higher-level logic of the source code provided. Different processor instruction sets will require different assemblers. Which one to use for the 68000 is mostly inconsequential nowadays, as most assemblers available will work in the same fashion. However it is interesting to note some of the various older assemblers and 68000 development environments that are still indeed available.

The British game development studio Psygnosis created a few commercial development kits *(SegaRetro, 2011)* for various game systems and computers throughout the 1990s. The 68000 assembler used as part of their Mega Drive development kit was leaked onto the internet many years ago. Running in MS-DOS and a later version in 32-Bit Windows as well, this assembler was used in many early homebrew projects for the Mega Drive and even other 68000 based systems. However it is hard to track down and obviously comes with legal limitations for usage in both commercial and educational contexts. As such it remains simply a footnote. One slightly less historic option for 68000 development is the unique “EASy68k” *(Charles Kelly, 2014)* , with the capital letters standing for “Editor, Assembler, Simulator”. This interesting Windows program is essentially both an integrated development environment and emulator / translation layer for 68000 assembly code. While configurable for Mega Drive and Amiga development, it is mainly an educational tool, providing many additional user interfaces for the visualization of processor states and register operation. It might not be an ideal solution for modern day development, but still receives updates once in a while.

Nowadays the landscape of 68000 assemblers has been largely reduced to two main choices. Most projects and individuals utilizing pure assembly code for their Mega Drive development seem to have settled on the VASM assembler *(Dr. Volker Barthelmann, 2021).* Supporting multiple optimizations, syntax flavors and output-modules for variable command-line notation, this actively developed open-source assembler is perhaps the most mature assembler of its kind. Beyond just the 68000 line of CPUs, it also supports a large quantity of other

architectures. The other most common choice is an obvious one for many people in software development. The GCC (GNU Cross Compiler) *(GNU Project, 2021)* is perhaps the most ubiquitous open-source compiler available and has become the de-facto standard compiler for a vast amount of use cases. Being part of the "GNU binutils", it is also a standard component of any modern UNIX-like operating system, such as Linux, BSD and by extension the modern macOS. As part of the compilation process, the pre-processed source files need to be assembled to the byte-code of the desired target platform, which is handled by the GAS (GNU Assembler) forming the back-end of GCC. This powerful assembler is simply so wide spread because of the prevalence of its contextual use, that many developers familiar with GCC will use it by default. Furthermore, when developing Mega Drive programs which do not use assembly exclusively, but also make use of high-level frameworks in C-language, the ability to compile both source types together using one tool is a far more concise approach.

Beyond the conversion of written code to executable programs, the act of writing the code in the first place is something to be touched on as well. Integrated Development Environments (IDEs) emerged for the simple reason that writing source code for many processors and systems in those days was not standardized and had no supportive tools to ease development. As such most people simply wrote their code in a plain-text editor and relied on the assemblers output to find where potential errors might be coming from. For writing assembly code for the 68000, this is for the most part still the case. The two most common editors used by homebrew developers are "NotePad++" (Windows) *(Don Ho, 2022)* and "Visual Studio Code" (Cross-Platform) *(Microsoft Corp, 2022).* Both have their advantages and disadvantages, but do include many handy features out of the box.

NotePad++ includes syntax highlighting and auto-completes defined labels. Command-line macros for quickly building the code to a ROM file can be added quite quickly from the preferences menu. For those familiar with this tool already, it would be a good choice to begin development with. Visual Studio Code on the other hand has the advantage of its modular extension system. Syntax highlighting is effortlessly added by installing the "Motorola 68000 Assembly" extension and to add even more functionality, the "68K Counter" extension can be installed. The latter allows for a column to be displayed to the left of the source

code, displaying information about the size of the lines instruction and CPU cycles consumed, along with a few handy macros.

High-Level Framework Development

As explained in the introduction, the wave of modern developers with a lot of passion for classic games yet no experience with low-level development in assembly languages prompted many to theorize higher-level tools and frameworks to facilitate an easier jump into development for this class of hardware. Which programming language to choose, capabilities to support and what assumptions to make about the development style to be facilitated are all hard and unintuitive questions that developers of such frameworks have to ask themselves. One project that sprung from this need took quite close notes from the higher-level abstractions found in early home computers.

Before computers became consumption devices for most people, the point of buying a computer for home use was mainly to learn programming in simple, interpreted languages such as BASIC. Various computers from the early to mid 1980s booted straight into the respective companies BASIC interpreter, ready to be programmed. While slow and rather cumbersome, simple algorithms and program flows could be implemented, before many moved on to learn the underlying assembly language of their respective system. However, programs were released even back then to speed up basic programs by compiling them straight to executables. This historic notion of simple, interpreted language being compiled down to executables was applied to SEGA Mega Drive development by Second Dimension LLC. In their "Second Basic Studio" application *(Second Dimension, 2022)*. Providing a full integrated development environment, centered around the usage of a custom BASIC language, this tool made large waves in certain circles, as it's nature as a BASIC derivative requires very little knowledge of the Mega Drive hardware or the advanced memory and buffer manipulation of many full programming languages. Workflows for importing graphics and sound are also present and most aspects of the Mega Drives intended features are indeed present and addressable. While a few tech-demos and even commercial games have been created using this tool, the reliance on BASIC will sooner or later be a hinderance for keen programmers, as

many have pointed out in forums and twitter discussions. For generalists and intrigued creatives, it still offers many handy features without much additional knowledge required.

The most notable project however, which by many accounts became the catalyst for a lot of homebrew Mega Drive games that emerged over the last few years, is the SGDK (Sega Genesis Development Kit), developed Stephane Dallongeville *(Steaphane D, 2022).* This project got its start from conversations about the creation of a high-level framework on the "spritesmind" forum dedicated to homebrew development. In these conversations from around 2010 to 2012, Stephane began working on his development kit, which has now become the most wide-spread set of tools for advanced Mega Drive development. Utilizing both the C language as standard and the 68000 assembly language for additional control over the system, it strikes a great balance of possible performance, low-level system access and ease of use. Beyond simply satisfying the programmers needs of writing efficient code with low-level system control when needed, part of the SGDK project is the resource compiler (RESCOMP). As part of the default SGDK toolchain, this resource compiler reads a simple definition file and imports the defined graphics, music and sound effects from common .PNG, .VGM and .WAV files. Depending on the type of resource declared, the resource compiler automatically outputs C header files, referencing resource definitions output in binary object format. From impressive commercial games to low-level tech-demos utilizing even undocumented features of the hardware, SGDK has proven to allow for more streamlined, modern development of SEGA Mega Drive games, without much compromise in any field.

While quite adapt at many use cases, SGDK was designed with a Windows operating system in mind, structuring its toolchain around the specifics of this environment. Users on Linux or macOS operating systems were first using virtual machines or translation layers such as WINE to make use of SGDK, but since then a few forks and additional projects have sprung up. "GENDEV" *(Kubilus1, 2015)* first emerged in 2015, identified by the creator as a companion project for SGDK, that allows for high-level Mega Drive development to be set up on a Linux operating system. While providing simple makefiles to instantly compile SGDK projects on Linux, a complete toolchain with custom GCC version can be built from its sources as well, if desired. Two years later in 2017, another project on GitHub started making a name for itself. "MARSDEV" *(Andre DeRosier, 2017)* attempted a very similar concept to

GENDEV, but aimed to be completely cross-platform, negating the need for different toolchains and build environments for different operating systems. The MARSDEV repository also contains makefiles for related tools, in particular the "MDTOOLS" repository commonly used for graphic handling in pure assembly contexts.

Relevant Asset Tools

The ease of including created artwork and music is one of the largest benefits of the higher-level SGDK framework and its resource compiler component. Speaking briefly on the workflow for including or converting art for usage in 68000 assembly projects, it is quite a lot more fragmented and specific. Many individual game projects developed in assembly will include the custom editors and graphics converters written specifically for that project in their repository. These tools often take form of color palette generators, tile editors and simple level editors. As systems from this time period were often vastly diverse in their approach to storing graphical information, it still is not the worst of ideas to write custom utilities for an assembly project, especially for the more project dependent data, such as level layouts or dialogue trees.

While these custom utilities are common, quite often they prove to be useful enough to warrant improvements and subsequent distribution for others to use in their projects. One repository commonly credited in assembly coded games is the aforementioned "MDTOOLS". This includes utilities for converting .PNG images into binary data to be loaded into the Mega Drives VRAM, sound utilities for converting MIDI and other sequenced audio formats to audio formats supported by one of the various open-source sound engines for the platform, as well as miscellaneous tools for generating ROM headers and fixing smaller issues with assembled ROMS. A more integrated approach for these tools was chosen for the development of the "Beehive" Mega Drive content tool, developed by Matt Phillips for his popular "Tanglewood" Mega Drive project *(Matt Phillips, 2019).* This tool includes a level editor, outputting configurable binary formatted data to be included in the assembly source, while also handling the conversion and compression of tile and sprite data for the created levels. The implementation of all of these concepts is rather specific to the development style chosen

for his game project, so simpler assembly based games will most likely not benefit from this tool all too much.

For the high-level development approach with SGDK, most content is handled via the resource compiler, as explained previously. For graphics and sound it is more than sufficient, even including various compression standards and sprite streaming modes to deal with limited memory. Levels can also be handled using the resource compiler, allowing for levels to be imported as images consisting of predefined tiles. Creating levels via a graphical editor however would be very cumbersome. There are a few general-purpose level editors which can be adapted for level creation in SGDK. While “Tiled” *(Thorbjørn Lindeijer, 2012)* is a popular and functional choice, the easy python integration of “LDTK” (Level Designer Toolkit) *(Sébastien Bénard, 2018)* was chosen and explored in detail as part of the related thesis. In short, one of the main reasons to use a specific level editor for such projects is ease of use, speed of iteration and the ability to design levels with not just tiles, but also entity placements in mind.

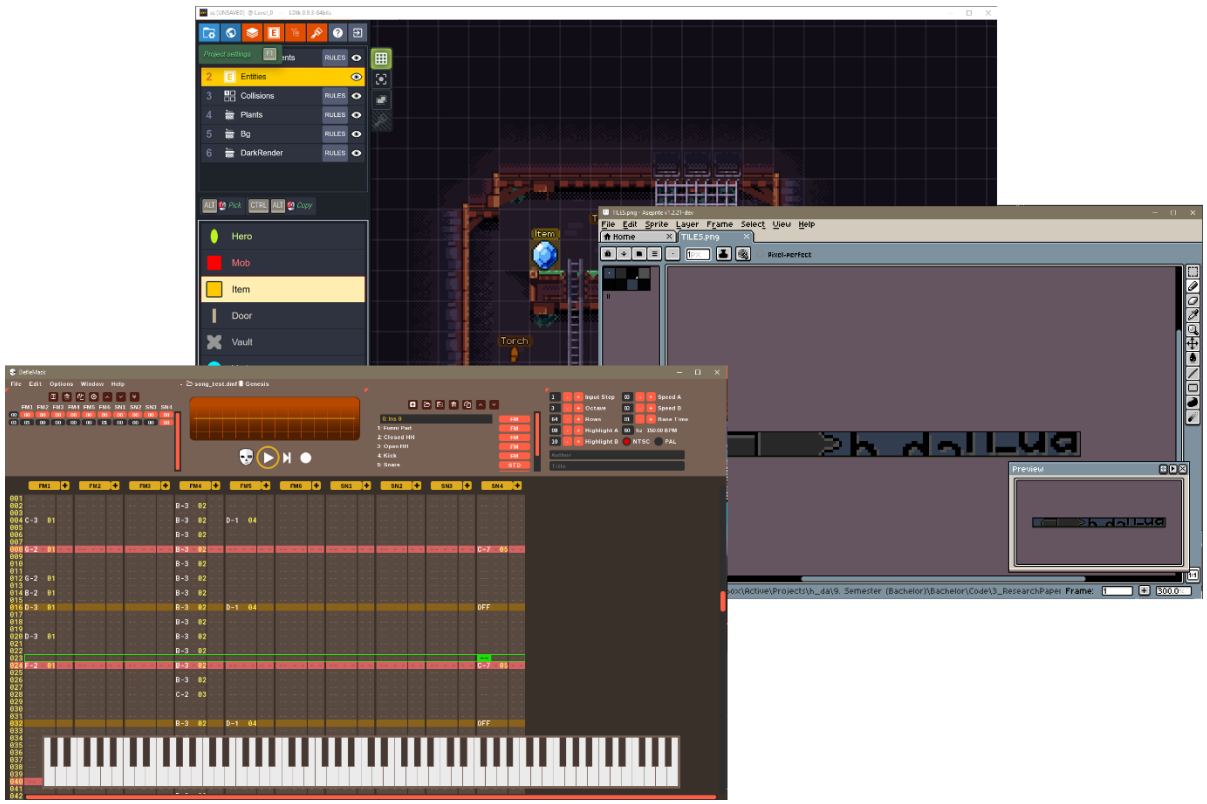


Fig. 01: Screenshot asset tools

As for the tools to create assets in the first place for both high-level and low-level approaches, there are a few commonly recommended choices. While Adobe Photoshop is the industry standard raster art tool, which can be quite comfortably configured for pixel-art, the strong importance of palleted color work gives pixel-art specific software packages, such as "Aceprite" *(Igara Studio, 2022)* or "GrafX2" a clear edge. For music, general MIDI compliant software can technically be used in conjunction with some of the aforementioned conversion tools, but the clear choice to be used for the Mega Drive is "Deflemask" *(Leonardo Demartino, 2011),* a music creation suite following the classic tracker design, specifically created for music creation on various old-school sound chips, including the Yamaha YM2612 of the Mega Drive. The exported .PNG and .VGM files of the mentioned programs can be loaded with SGDK's resource compiler, or the necessary conversion utilities for assembly into binary data. For more specific information of usage, both the implementation chapter of this paper and the related thesis offer more details.

State of Mega Drive debugging and performance profiling

Evaluating the code produced and handling emerging errors is of course a vital part of programming in any context. For the SEGA Mega Drive and other consoles of the time period, debugging possibly faulty code and gaining an understanding of the available memory and processing time consumed was equally vital, yet a lot less integrated compared to modern development. Most of this functionality was built into the physical development kits required to create games for these consoles. Commonly these development kits would consist of modified Mega Drive hardware, housing a reprogrammable cartridge and a development board taking control of the systems components. Said development board would then interface, usually over a serial connection, with a host PC, mostly running MS-DOS based software. Code written on the PC could be transferred and executed on the system, after which debugging software could be run, allowing for many familiar features such as breakpoints and line-by-line stepping through the code. Just like modern debugging functionality, the value of memory addresses and processor registers could be observed this way to catch possible faults in the written logic. The aforementioned "Tanglewood" project by Matt Phillips used an

original development kit from the time for authenticity, which proves the continued functionality of this approach. However most if not all functionality offered by these rare, expensive and scarcely documented devices can be replicated by modern means, if not exceeded.

With the advent of modern emulation software, the need for physical development kits has all but vanished. It is still an important point to make sure the written code runs on the real physical console hardware, however this can be achieved much simpler by use of a flash-cartridge and a standard console. The choice of emulator is an important one, as many different emulators exist for the SEGA Mega Drive system, all with different functionality and use cases. Perhaps the most popular emulator for the system is "Kega Fusion"*(Steve Palmer, 2010)*, which while decent for playing the occasional ROM, is neither very accurate to the original hardware, nor does it offer much in the way of debugging functionality. The "Gens" emulator is another popular choice, which despite also not being highly focused on accuracy, does come standard with more tools to poke around the system with. What makes Gens a lot more useful however is a modification of the original source code by a homebrew developer by the alias of Kaneda. This modification known as "GensKmod" *(Kaneda, 2006),* includes many additional features useful for debugging purposes. These features include a debug message window, which allows for arbitrary strings to be output from the running ROM. SGDK provides functions for this, making it trivial to quickly observe code paths traveled, amount of loop iterations or the value of variables at specific points. Additionally, memory watchers are supported, allowing to keep track of certain memory regions over time, which becomes quite useful when using multiple data buffers. Beyond that, this emulator also offers the ability to attach the GDB (GNU Debugger) to the running process, as well as windows for viewing the state of processor registers, next instructions to be executed, state of the VRAM, tile layers and sprite table entries. Rather fully featured for a modification and incredibly useful. Despite these features, the emulation accuracy of GensKmod is not ideal, allowing certain edge cases which would trigger an exception from the vector table on real hardware. To check a compiled / assembled ROM for accuracy, one can either run it on the real hardware (somewhat cumbersome, yet most accurate) or choose an emulator aiming for perfect accuracy. One of the most accurate emulators currently available is "BlastEM" *(Michael Pavone, 2019)*, which often gets recommended by members of the homebrew community.

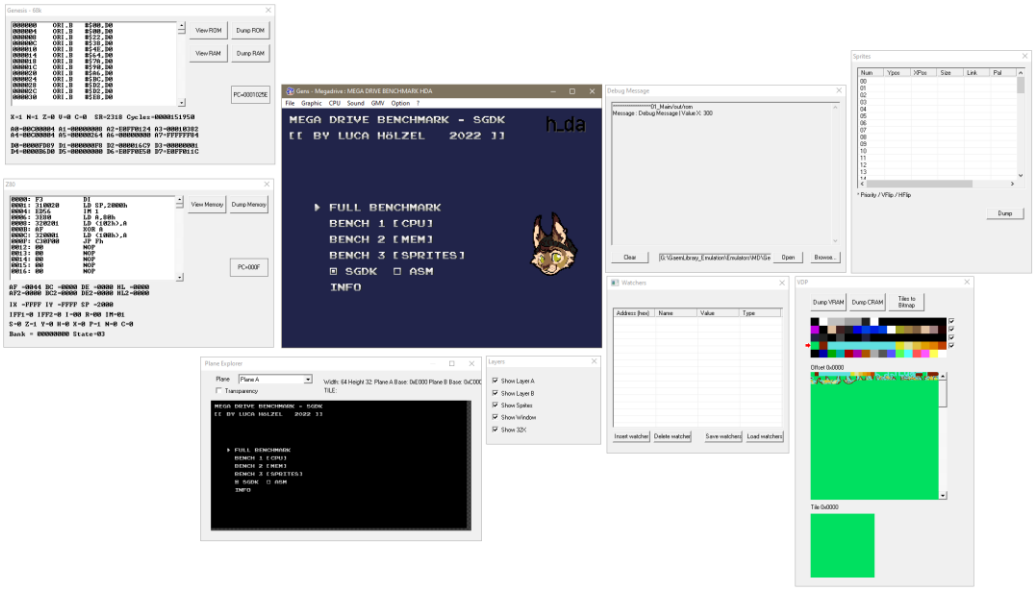


Fig. 02: Screenshot GensKmod debugging emulator

While the previously mentioned emulators duplicate most of the functionality present on the original development kits, they are still not close to the standard of profiling and debugging one might be accustomed to nowadays. A unique project that emerged from the need for a more complete profiling solution is “MD-Profiler” by GitHub user Tails8521. This tool is intended to be used with a fork of the aforementioned BlastEM emulator. Given an mdp trace file recorded using the emulator and the symbols (name of variable, functions, etc.) file generated by SGDK or manually created for assembly, the MD-Profiler application will generate a JSON trace file, which can be viewed with most profiling viewers, including the one built into the Google Chrome web-browser (chrome://tracing/). The graphical representation of execution times is intended to help finding bottlenecks, bugs and gaining a more immediate understanding of the achieved performance.

APPROACH

Researched Topics

During the early stages of semester preparations, many viable resources found across the internet were collected to begin approaching the development on this very limited system. To begin gaining an understanding of the hardware to be developed for, in-depth research into the system architecture of the SEGA Mega Drive and its components was conducted. This included research into the memory mapped nature of its system-bus, as well as the jurisdictions, interactions and limitations of the main 68000 processor, the Z80 co-processor and the VDP (Video Display Processor). In particular the handling of transferable bus-control via DMA (Direct Memory Access) posed a unique difference to any modern architecture one might be familiar with and required the cross-referencing of multiple sources to build and informed mental image of its intended usage. Beyond the abstract nature of memory access authority, the general information about the system was noted and compared to competing hardware of the time period. Available memory, processor clock speeds and capabilities of the video and sound hardware were categorized and understood by identifying their workings in emulators running games of the time period. From the information gathered, the sizes of various important structures and elements were identified, such as the 7-Bit color representation, 2-Pixel-Per-Byte tile data format and the size of possible offsets for tile-map locations in VRAM. All of which would later become vital for programming.

After an extensive dive into the hardware specifics of the system, one might think to move directly into the low-level development discussed in the previous chapters. Having only studied abstract information about these inner workings however, without any first-hand experiences in assembly development, the decision was made to work on the higher-level frameworks first. SGDK, the Sega Genesis Development Kit discussed in the previous chapter, uses the C language for all development. Being one of the first true higher-level languages of its kind, it was designed to bring programming to a new era, while still being fully capable of lower-level access to the processor and memory. Having previously worked with many modern, object-oriented C-derived languages, the environment would prove to be much more welcoming. Given the fact however that classic C and C++ are not managed languages, in contrast to C# and Java, meant that manual memory allocation and pointer arithmetic had to

be studied and understood to make informed usage of the capabilities of the SGDK framework. For this a section of time was allotted within which to study and experiment with the pointer and memory methodologies of the C language.

Once comfortable, the next step was to follow the collected information and guides for both setting up an SGDK build environment and beginning to implement a few simple programs. The former took more time than expected, as multiple development environments (CodeBlocks, Eclipse, VSCode) were supported and thoroughly tested, before settling on Visual Studio Code for its related extensions. The setup of build environments is documented in detail in the related thesis and the wiki component created as part of it. Beyond the documentation provided with SGDK itself, multiple useful guides and tutorials were referenced to begin forming a cohesive approach to implementing common game features and graphical elements. After studying these examples to completion, multiple open-source projects using SGDK were examined for their structure and implementation of various common tasks. A few proof-of-concept implementations were created after this point to put the learned concepts to use and account for possible misinterpretations of studied material, as well as learning the described debugging process for SGDK development.

After building a solid understanding for the functionality and use of SGDK, the remaining time was spent studying the low-level assembly development. First various online tutorials and snippets of information on the 68000 assembly language were referenced, which proved to provide very little overall understanding on the subject. One vital resource on the subject came in the form of a website, tutorial series and a full book on the development in various assembly languages by Keith ‘Akuyou’, aka “Chibiakumas”. As part of his personal website and YouTube channel he began creating a tutorial series on the assembly language for multiple classic home computers and game systems over the last decade. In late 2020 he released a full book covering the specifics of multiple CPU instruction sets including the Z80, 6502 and 68000 processors. The general information outlined during the earlier chapters of the book applied to all architectures and served as a great introduction into the vocabulary, restrictions and hardware knowledge required to begin understanding assembly development. Furthermore the book included example code and a pre-configured build environment using

NotePad++ and VASM, which like previously explained is a commonly used setup in current homebrew development.

While the book and related material by Chibiakumas did serve as a great introduction, the specifics of implementing game-relevant features were beyond their scope. Perhaps the most valuable resource to exemplify the development of a real game utilizing 68000 assembly language for the SEGA Mega Drive, turned out to be a series of blog posts. Hugues Johnson detailed his journey into Mega Drive assembly development on his blog, spanning from his personal early attempts at figuring out hardware features in 2015 to the last steps of development of his title “Retail Clerk ‘89” in 2020. The many explicit examples for input handling, VRAM tile and sprite loading, as well as the writing of subroutines for functional game loops proved to be a highly valuable resource to a beginning developer of the subject. While learning and internalizing more and more about this style of low-level development, the notes on hardware specifics and data formats that were gained from the above hardware research became vital to rapid and effective development. The memory locations of certain memory mapped devices and tables were particularly frequently used. From the information gathered, the final step in both preparation and research on this topic was the construction of a template project for pure assembly development. This template consists of multiple source files divided into folders based on contents, then included into the main assembly source file, creating a more modular base to work off of. The implementation of this template will be discussed in the implementation chapter.

Throughout the process of learning all required information and methods of implementation, many questions and frustrations did emerge, often leading to confusion about the learned material. Luckily the homebrew community is surprisingly eager and happy to assist newcomers and answering their questions. Most questions regarding SGDK were resolved by engaging in conversation with member of the official SGDK discord server and even the SGDK maintainer Stephane Dallongeville himself. After additional conversations in discord servers relating to assembly development and with individual homebrew developers on twitter, almost all questions that had hindered the understanding and continued development with both SGDK and 68000 assembly, were cleared up. The importance of such a supportive community cannot be understated for continuing the growth of homebrew development.

Relevant features for performance testing

To begin defining the features for examination, it is important to reiterate the aspired goal intended to offset the proposed problem. As stated, the differences in development and performance within the contexts of low-level assembly language and the high-level SGDK framework are to be examined and formalized to further the academic understanding of modern homebrew development for the SEGA Mega Drive system. The main focus of performance evaluation in this case will not consist of theoretically achievable performance in the most optimal cases, but rather to inspect the feasible performance achievable for an interested developer actively learning both styles of implementation.

To compare both development approaches, the same tasks need to be implemented in both low-level and high-level code. The resulting performance of both implementations can then be examined and its implications for the state of the art and future projects can be evaluated. The question of what to implement for such a comparison does not have one clear answer. While the ultimate goal is to further game development for the SEGA Mega Drive system, implementing a game-loop, or even full game, twice with completely different methods of implementation poses risks. Given the difference in development styles and methodology in assembly language and C language, certain differences in execution and performance will creep in, which would not be representative for the desired examination. As such, the implementations to be examined should each concern themselves with simpler, more theoretical tasks highlighting individual aspects of the hardware capability, rather than the interaction between all hardware aspects.

For instance, the speed of many typical game operations such as character movements and collision detection rely on both the speed of the processors calculations, as well as the bus-speed at which data from ROM or RAM can be loaded into the processors registers. Since the written assembly code will not be identical to the operations derived by the C compiler, dealing with two variable speeds within one performance benchmark could lessen the meaningfulness of the performed comparison. As such two benchmarks, each focusing on pure CPU performance and Memory reading/writing respectively, would act as good baseline readings for the unhindered performance of these hardware elements.

With these features being more theoretical, the graphical aspect of the system obviously needs to examined as well. Considering that a high amount moving sprites on screen is not only a mark of an impressive game for the time, but also typically the crux of performance, causing slowdowns or skipped drawing cycles, the importance of this feature cannot be understated. The drawing process of animated sprites of various sizes, as well as the interaction between them via collision detection should result in a benchmark closer to the graphical and logical needs of a real game. With these three proposed features, comparing the achievable performance for processor, memory and graphics on the SEGA Mega Drive console, via low-level and high-level implementations, should result in relevant information to further the academic knowledge on the performance of such a development cycle.

Benchmark program to be implemented for data aggregation

While the previously discussed template for a pure 68000 assembly project proved to be a great starting point for simple experiments, implementing more complex contexts such as a full benchmark of various system tasks would drastically slow down development. All the necessary code for gaging performance and implementing the surrounding navigation between tests would consume additional implementation time, without changing the performance of the actual tests to run. Worse, having two entirely separate implementations could lead to different configurations of memory allocation or tile-map placement, possibly impacting performance in unforeseen ways. As established in the current state of 68000 development, both the popular VASM and the ubiquitous GNU Assembler are equally valid choices for assemblers, picked more so because of familiarity with one over the other. As such, the ability to compile both C source code using the SGDK framework and assemble the hand-written assembly code into the same executable ROM would even the playing field, making sure both implementations have the exact same configuration of memory as their starting point. It also makes it possible to switch between C or assembly on the fly between tests, resulting in more rapid benchmarking of both implementations.

Firstly, the approach to the programs overall structure needs to be made clear. The previously noted point about keeping initial conditions for all tests in both implementations equal informs the decision for a modular approach to the program flow and test structure. The

benchmark program is to be designed with a state system in mind, implementing each test, menu and additional information display within an instance of a state structure. Each of these states would contain references to logic implementing preparation, execution and restoration of the desired state functionality, as well as references to the previously executed and next to be executed states, forming a linked list data type. This way the order of states can be arbitrarily changed in code or even during runtime, facilitating the execution of individual states or multiple states in succession. The first of these states, running on program startup, is to implement a simple menu system, allowing to pick which individual state or sequence of states (benchmarks) to execute, as well as offering toggling between using the C or assembly implementation of the states (benchmarks). Equally a reusable result state will be implemented, listing the measured performance of whichever benchmark was executed last.

The benchmark states implemented will follow the relevant features outlined in the previous section. In short one state will be implemented each for benchmarking the processor, memory and sprite graphics. For the CPU test, math operations of both 16-Bit and 32-Bit values (word / longword), as well as signed and unsigned values will be examined. These additions, divisions and multiplications (main mathematical operations of 68000 instruction set) will be repeatedly executed for a fixed number of iterations, after which the time taken and amount of data processed can be calculated and displayed. Similarly the memory benchmark will implement the writing to a memory buffer, as well as copying from one area of a memory buffer to another (memset, memcpy in C language terms), with each of these operations being executed three times with increasing buffer sizes. The sprite test will consist of three iterations of sprite operation, each repeated twice. For each of the three iterations animated sprites increasing in size will be displayed and moved across the screen. For the repetition of these operations, collision detection will be enabled, showcasing the performance loss of the additional computations.

Once implemented, the results produced by the benchmark will be aggregated for inspection. To give a better overview of possible performance differences, results will be sampled from the final ROM executable running under multiple emulators as well as on the real (PAL) SEGA Mega Drive console through the use of a flash cartridge. Additionally the noted fork of the BlastEM emulator will be used to record a performance trace file to be used with the

mentioned MD-Profiler application. Utilizing this tool a viewable profiling trace will be generated, adding an additional stream of visual information about the achieved performance to the results to be evaluated. All values collected as part of this procedure will be organized in tables from which graphs for visualization will be generated using Microsoft Excel.

IMPLEMENTATION

Preface of implementation

To begin explaining the implementation of the desired benchmarking program, sections are laid out with each going over an individual topic, to keep the information on each topic concise. While the benchmarking code itself has the highest relevance, understanding the logic behind the overall structure of the program is not only helpful in comprehending the provided code, but will aid in the approach to game development or similar benchmarking attempts on this hardware.

The basic premise of this benchmark implementation is to produce findings relating to both the performance and implementation quality of a student or other interested party, given a roughly equal time frame to study both high-level and low-level implementation approaches. To improve code readability and support this iterative nature, the program is constructed in a very modular way, allowing for certain functions to be easily replaced and the order of execution to be trivially changed. This allows the implementation to not just benchmark the current level of performance, but will also aid to evaluate possible better solutions one might find as their experience in the subject grows. As such the implementation represents a test-bed for not just abstract hardware operations, but for ones grasping of their usage, relation and performance.

Program Implementation

Program Structure & Framework

As explained previously, the implementation of both the high-level C/SGDK code and the low-level 68000 assembly code are integrated into one cohesive project folder. The SGDK project structure contained inside references assembly files which will be assembled along with the compiled higher-level C code into one executable ROM, leveling the playing field and removing differences in system state that might be unaccounted for. Familiarizing one-self with the structure of this surrounding SGDK project is necessary to continue further into the implementation itself.

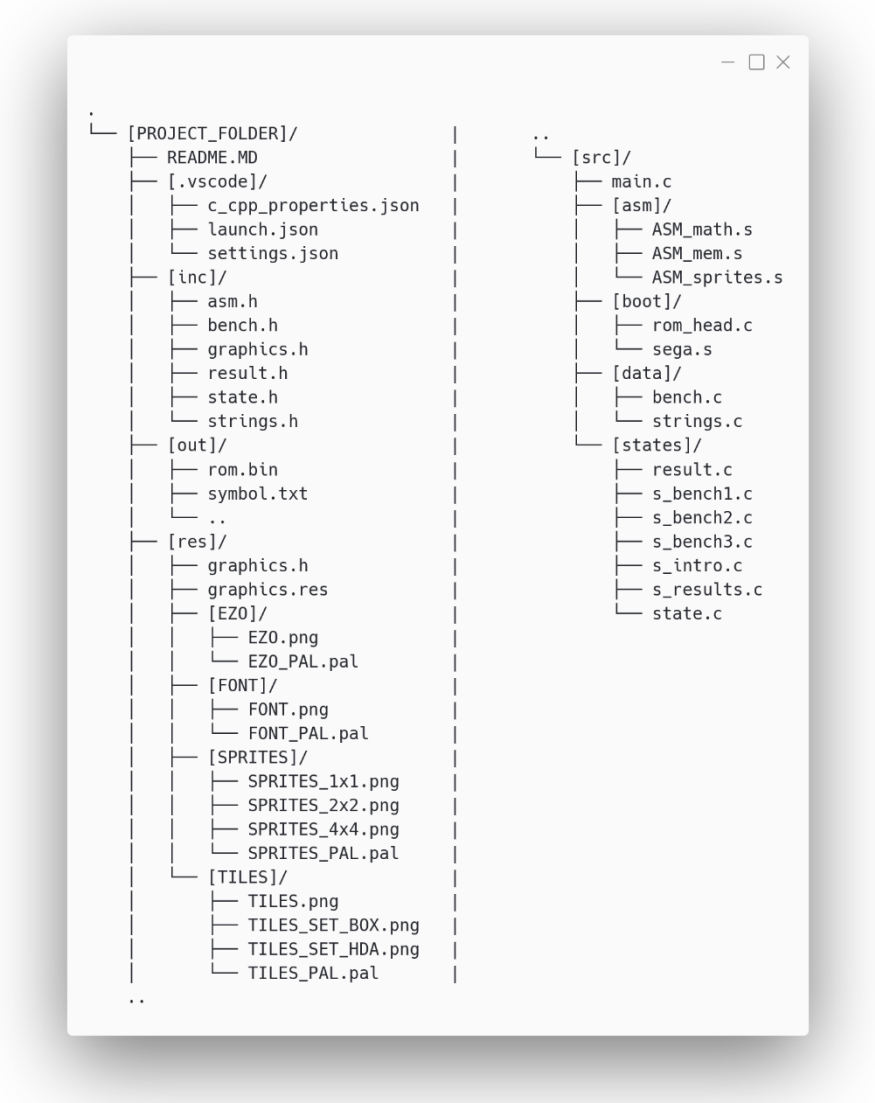


Fig. 03: Benchmark folder structure

Starting from the top, the innocuous seeming visual studio code configuration folder (.vscode) holds a lot of merit for the development setup, as it includes setting files in JSON format, configuring the include paths and debugging hooks for the development process. More information on this is found in the main thesis. Below that is the include folder, containing the C header files (.h) defining many commonly used functions, structures and variables. Of note here is the asm.h file, which defines the names of assembly functions later linked to the program during the assembly / compilation process. The function parameters defined here will also inform the creation of the assembly stack, when the relevant functions are called. The out folder contains all files generated during compilation. This includes the individual output files (.o) linked together in the later steps of compilation, forming the output rom file, ready to be run. Noteworthy as well is the symbol.txt file generated during this process by default. This file contains references and names to the functions and variables used, making it a commonly used input file for debugging purposes, specifically used in the profiling section of the following evaluation chapter. The res folder contains all resources and is the default folder operated on by the resource compiler as part of SGDK. The *.res file present in the folder contains definition syntax as instruction for the resource compiler, which then converts and prepares all requested data into binary object files for later linking. To be useful in code, a header file declaring these by name is also generated and copied to the include folder. Lastly the src folder naturally contains all the source code implementations which will be examined during the following sections of this chapter. The surrounding folders are named descriptively for their contents. The asm folder contains the 68000 assembly source files, implementing the routines defined in the asm.h include file. The data folder equally contains C source code files implementing the functions and values of their respectively named header files. Finally the states folder is home to all files implementing or defining the state system used in this project, forming a simple design pattern for modularity. The definition and logic behind this state system will be examined next. The present main.c file and boot folder containing startup code and rom header are default in any SGDK project, but to be modified for the desired implementation. It is important to note that the basic top-level structure of this program follows the basic structure of any SGDK program template. If specific implementations benefit from or require a vastly different file structure, changes to

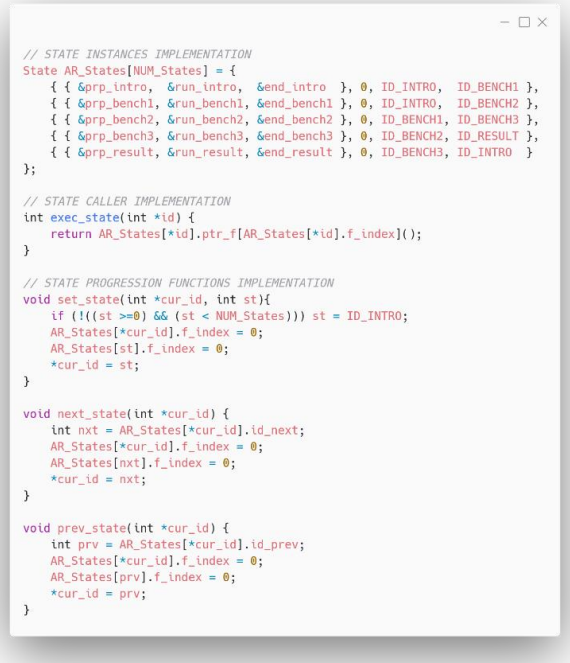
SGDKs makefiles have to be implemented. This serves as one benefit for the lesser-structured pure assembly method examined as the last section of this chapter.

The state system deployed for this implementation serves as the organizational backbone of the project. Many similar approaches to game development or benchmark implementation in the SGDK framework rely on simple functions arranged in one larger caller function, making the rearrangement or addition to the available benchmarks and functions quite cumbersome. A more modern and modular approach was implemented to aid with these organizational issues. The state.h and state.c files present in the file structure define and subsequently implement the state structure and its functionality. Firstly, the state structure data type is defined, containing an array of three function pointers, an index into that array as well as indexes referencing the previous and next state. The three functions referenced by the array always follow the order of preparing, running and ending the state respectively. While the running function is designed to simply execute the wanted benchmark or other code, the preparation and ending functions load and unload needed data or settings, making sure the next state receives the same initial conditions, ensuring consistency. The index of the function pointer array thereby controls the flow within one state, whereas the previous and next indexes define a linked list of states, forming the overall program flow.

For a global reference of available states and their order an enumeration type is defined, naming each state and their index for better readability. The individual state functions and further functionality for the access and manipulation of states are defined following the enumeration. The implementation of these structures and functions is laid out in the accompanying state.c file, starting with an array of state instances, with each array element implementing one of the desired states and being initialized accordingly. The following function implementations operate on the state array, allowing to execute states and following the linked list of states in either direction. Noteworthy here is the integer pointer parameter received by these functions. To keep the flow of the program dependent on the state structure, a single variable is used program-wide to determine which state to run. This variable is defined outside of the state structure, to remain modular and reusable in further projects. The address of this variable is passed to the functions, each working with the value pointed to by the address.

```c
// MAIN STATE STRUCTURE
#define NUM_State_Func 3
typedef struct State {
    int (*ptr_f[NUM_State_Func])();
    int f_index;
    int id_prev;
    int id_next;
} State;

// STATE INSTANCES
State AR_States[NUM_States];

// ENUM FOR STATE IDENTIFICATION
#define NUM_States 5
enum State_ID {
    ID_INTRO  = 0,
    ID_BENCH1 = 1,
    ID_BENCH2 = 2,
    ID_BENCH3 = 3,
    ID_RESULT = (NUM_States-1),
};

// INDIV. STATE FUNCTIONS
int prp_intro();
int run_intro();
int end_intro();
//--------------
int prp_bench1();
int run_bench1();
int end_bench1();
//..

// STATE PROGRESSION FUNCTIONS
void set_state(int *cur_id, int st);
void next_state(int *cur_id);
void prev_state(int *cur_id);
void set_stateEX(int *cur_id, int st, int fi_cur, int fi_st);
void next_stateEX(int *cur_id, int fi_cur, int fi_nxt);
void prev_stateEX(int *cur_id, int fi_cur, int fi_prv);
void set_result_flow(int last, int next);
```

Fig. 04: State header file



```c
// STATE INSTANCES IMPLEMENTATION
State AR_States[NUM_States] = {
    { { &prp_intro,  &run_intro,  &end_intro  }, 0, ID_INTRO,  ID_BENCH1 },
    { { &prp_bench1, &run_bench1, &end_bench1 }, 0, ID_INTRO,  ID_BENCH2 },
    { { &prp_bench2, &run_bench2, &end_bench2 }, 0, ID_BENCH1, ID_BENCH3 },
    { { &prp_bench3, &run_bench3, &end_bench3 }, 0, ID_BENCH2, ID_RESULT },
    { { &prp_result, &run_result, &end_result }, 0, ID_BENCH3, ID_INTRO  }
};

// STATE CALLER IMPLEMENTATION
int exec_state(int *id) {
    return AR_States[*id].ptr_f[AR_States[*id].f_index]();
}

// STATE PROGRESSION FUNCTIONS IMPLEMENTATION
void set_state(int *cur_id, int st){
    if (!((st >=0) && (st < NUM_States))) st = ID_INTRO;
    AR_States[*cur_id].f_index = 0;
    AR_States[st].f_index = 0;
    *cur_id = st;
}

void next_state(int *cur_id) {
    int nxt = AR_States[*cur_id].id_next;
    AR_States[*cur_id].f_index = 0;
    AR_States[nxt].f_index = 0;
    *cur_id = nxt;
}

void prev_state(int *cur_id) {
    int prv = AR_States[*cur_id].id_prev;
    AR_States[*cur_id].f_index = 0;
    AR_States[prv].f_index = 0;
    *cur_id = prv;
}
```

Fig. 05: State implementation

Because each of these states contain their own respective initialization (preparation) and cleanup (ending) code, the main.c file serving as the programs entry point only needs to setup settings for system initialization or those relevant to all states. In fact this file only contains one short initialization function, concerned with the screen resolution for NTSC and PAL systems, the global size reserved for tilemaps in VRAM, the globally unchanged palettes and the globally used font. These function calls are wrapped in SYS_disableInts() and SYS_enabneInts() calls to ensure no system interrupts happen during initialization. This is used throughout the program as it is documented to be best practice, potentially avoiding memory corruption. After calling this initialization function, a never ending while loop is entered. This game loop presents the modular nature of this state approach in full effect, as the only thing required to be call from this loop is the exec_state function, defined in state.h, given the address of the global state index. From this point, the implemented logic of moving between states of their own jurisdiction takes over the program flow and keeps iterating with every loop cycle.

```
void main_init() {
    SYS_disableInts();

    // Set Video Modes
    VDP_setScreenWidth320();
    if (IS_PALSYSTEM) {
        VDP_setScreenHeight240();
    }else{
        VDP_setScreenHeight224();
    }

    // Set Plane sizes
    VDP_setPlaneSize(64,32,TRUE);

    // Load Palettes
    PAL_setPalette(PAL0, IMG_FONT.palette->data, DMA);
    PAL_setPalette(PAL1, IMG_EZO.palette->data,  DMA);
    PAL_setPalette(PAL2, IMG_TIL.palette->data,  DMA);
    PAL_setPalette(PAL3, SPR_SML.palette->data,  DMA);

    // Setup Font
    VDP_loadFont(IMG_FONT.tileset, DMA);
    VDP_setTextPalette(PAL0);
    VDP_setTextPlane(BG_A);

    // Setup Results
    prp_results_memory();
    SYS_enableInts();
}

int main() {
    main_init();
    while(TRUE){ exec_state(&CUR_State); }

    return (0);
}
```

Fig. 06: Main source file

As an example for the implementation of such a state, the “Intro” state serves well at this point, as it does not implement any benchmark specific features, but rather acts as the main menu, allowing for benchmarks and execution type to be picked. The preparation function here, again internally disabling and later enabling the system interrupts, sets up the background color (32/64 = first color of third palette), then loads the required tilesets into VRAM. References to these tiles are then set into one of the VRAM tilemaps at the desired position. These functions especially need the interrupts to be disabled, as these high-level function wrappers access the VDP over its memory mapped ports, then initialize a DMA transfer. Afterwards the remaining text and box graphics are drawn and SGDKs input system is initialized. Each state typically implements their own logic for input handling depending on their required functionality (if any). These handlers are referenced to the input system by

assigning them as callbacks to be executed and receive new input data whenever the underlying system registers an input change.

Finally for the preparation, the function pointer of the “Intro” state is increased, now pointing to the running function implemented just below. As all menu logic is dependent on controller input, the running function here implements nothing but the system-call to do the vertical blank process. In most SGDK implementations, this call needs to be executed in the game loop to synchronize frames and allow the queued DMA requests to be executed.

The ending function switches to the state value buffered from the selection made in the input callback function. If a full benchmark run is selected the next chain in the list is chosen, else the position of the cursor is used to determine the next wanted state. As a last point the background color and tilemap planes are cleared, for the next state to have a clear screen to work with.

```
// INTRO STATE FUNCTIONS
int prp_intro() {
    SYS_disableInts();

    //Setting Background Color Index
    VDP_setBackgroundColor(32);
    [..]
    // (HDA)
    VDP_loadTileSet(IMG_HDA.tileset, IND_TILE_PROG, DMA);
    VDP_setTileMapEx(BG_B, IMG_HDA.tilemap, TILE_ATTR_FULL(PAL2, 0, 0, 0, IND_TILE_PROG), 33, 1, 0, 0, 5, 2, DMA);
    IND_TILE_HDA    = IND_TILE_PROG;
    IND_TILE_PROG += IMG_HDA.tileset->numTile;

    // (BOX)
    VDP_loadTileSet(IMG_BOX.tileset, IND_TILE_PROG, DMA);
    IND_TILE_BOX    = IND_TILE_PROG;
    show_box(FALSE);
    IND_TILE_PROG += IMG_BOX.tileset->numTile;

    // (EZO)
    [..]
    // Drawing Top Text
    VDP_drawText(STR_Title,  1, 1);
    VDP_drawText(STR_Credit, 1, 3);

    // Input setup
    JOY_init();
    JOY_setEventHandler(&input_intro);

    SYS_enableInts();
    AR_States[ID_INTRO].f_index++;
    return 0;
}

int run_intro() {
    SYS_doVBlankProcess();
    return 0;
}

int end_intro() {
    SYS_disableInts();
    // Determine next wanted state
    BUF_NEXT = (BENCH_IsFullLRun ? AR_States[ID_INTRO].id_next : CURSOR_DY);
    // Switch State
    set_state(&CUR_State, BUF_NEXT);
    // Clean up state
    VDP_setBackgroundColor(0);
    VDP_clearPlane(BG_A, TRUE);
    VDP_clearPlane(BG_B, TRUE);
    [..]
}
```

Fig. 07: Intro menu state function implementation

Lastly the input callback will move and keep track of the vertical position of the cursor. When a selection input is made, the cursor_exec function is called to act on the currently selected item. Depending on the selection, variables will be set accordingly for the use of C or assembly language during benchmarking or the nature of full or individual benchmark runs. After this, the function pointer index is increased again, executing the previously described ending function for this state.

```
void input_intro(u16 joy, u16 changed, u16 state) {
    if (CURSOR_ALLOW) {
        if (state & BUTTON_UP) {
            if (CURSOR_DY <= 0) { CURSOR_DY = 0; return; }
            cursor_del();
            CURSOR_DY--;
            cursor_put();
        }
        if (state & BUTTON_DOWN) {
            if (CURSOR_DY >= CURSOR_DY_MAX) { CURSOR_DY = CURSOR_DY_MAX; return; }
            cursor_del();
            CURSOR_DY++;
            cursor_put();
        }
    }
    if (state & (BUTTON_A | BUTTON_B | BUTTON_C | BUTTON_START)) {
        cursor_exec();
    }
}

void cursor_exec() {
    // Box Close
    if (!CURSOR_ALLOW) { show_box(FALSE); return; }

    // Menu selection
    switch(CURSOR_DY) {
        case (0):
            // FULL BENCHMARK
            BENCH_IsFullLRun = TRUE;
            AR_States[ID_INTRO].f_index = NUM_State_Func-1;
            break;
        case (1):
            // BENCH 1
            BENCH_IsFullLRun = FALSE;
            AR_States[ID_INTRO].f_index = NUM_State_Func-1;
            break;
        case (2):
            // BENCH 2
            BENCH_IsFullLRun = FALSE;
            AR_States[ID_INTRO].f_index = NUM_State_Func-1;
            break;
        case (3):
            // BENCH 3
            BENCH_IsFullLRun = FALSE;
            AR_States[ID_INTRO].f_index = NUM_State_Func-1;
            break;
        case (4):
            // C / ASM Switch
            USE_ASM = !USE_ASM;
            show_box(FALSE);
            break;
        case (5):
            // INFO BOX
            show_box(TRUE);
            break;
    }
}
```

Fig. 08: Intro menu state input functionality

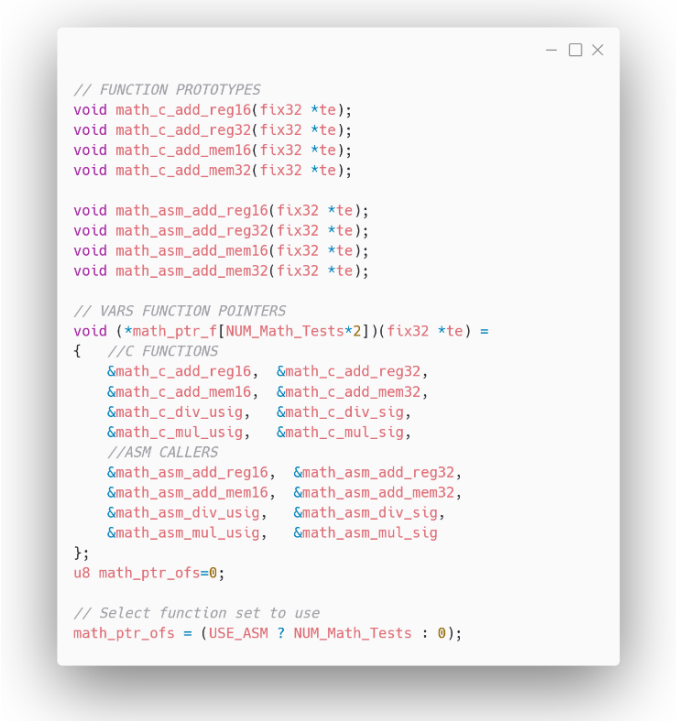
While rather simple, this approach to state implementation keeps overlapping jurisdiction over variables to a minimum and allows for frequent changes to program flow, without much refactoring. The concepts outlined during this section will be carried forward into the following sections on benchmark implementation, without additional repetition of the stated workflow.

CPU Benchmark implementation

As described in the third chapter of this paper, the first benchmark is to concern itself with the performance of purely CPU bound tasks, in order to get a performance reading which is not hindered by memory access speeds of any kind. The obvious tasks that fit this description are mathematical operations. Once all values are loaded into the processors registers, the mathematical operations conducted on them are purely bound in execution time by the raw power with which the CPU can execute the desired operations. Which mathematical operations to pursue as part of this benchmark was simply based on the mathematical operations which the 68000 instruction set was designed to handle, potentially revealing not just the speed difference between assembly and C implementation, but also the speed differences between these operations on a CPU level.

The operations chosen were the addition of 16-Bit (word) and 32-Bit (longword) numbers, unsigned and signed division (of two words, resulting in longword output) as well as unsigned and signed multiplication. To compare the speed difference when memory access is indeed involved, it was also decided to include 16-Bit and 32-Bit additions from memory locations as well. The lack of 8-Bit math operations stems from early experiments, revealing CPU bound additions of 8-Bit numbers to result in identical performance as 16-Bit numbers, resulting in their omittance as essentially duplicate values.

Starting with the declaration of used functions and variables, a common pattern emerges about the structure of function calling for both C and assembly purposes. After function prototypes for the C benchmark functions and assembly benchmark function callers are defined, they are referenced into an array of function pointers, each section occupying half of the array. An offset value is then initialized to either 0 (index of first C function pointer) or the number of math tests (index of first assembly caller). This value is later used as an offset into the array, needing only one implementation of the function call from the state loop, instead of branching depending on the state of the assembly variable.



```
// FUNCTION PROTOTYPES
void math_c_add_reg16(fix32 *te);
void math_c_add_reg32(fix32 *te);
void math_c_add_mem16(fix32 *te);
void math_c_add_mem32(fix32 *te);

void math_asm_add_reg16(fix32 *te);
void math_asm_add_reg32(fix32 *te);
void math_asm_add_mem16(fix32 *te);
void math_asm_add_mem32(fix32 *te);

// VARS FUNCTION POINTERS
void (*math_ptr_f[NUM_Math_Tests*2])(fix32 *te) =
{   //C FUNCTIONS
    &math_c_add_reg16,  &math_c_add_reg32,
    &math_c_add_mem16,  &math_c_add_mem32,
    &math_c_div_usig,   &math_c_div_sig,
    &math_c_mul_usig,   &math_c_mul_sig,
    //ASM CALLERS
    &math_asm_add_reg16,  &math_asm_add_reg32,
    &math_asm_add_mem16,  &math_asm_add_mem32,
    &math_asm_div_usig,   &math_asm_div_sig,
    &math_asm_mul_usig,   &math_asm_mul_sig
};
u8 math_ptr_ofs=0;

// Select function set to use
math_ptr_ofs = (USE_ASM ? NUM_Math_Tests : 0);
```

Fig. 09: Benchmark 1 function pointers

The three state functions implemented for the first benchmark state follow a similar process as described previously. The preparation function draws information about the conducted test to the screen, then increases the function pointer to begin the running function. It is inside this function that we get our first look at the performance measurement methodology deployed for the benchmark. After initializing a 32-Bit fixed point variable for time measurement (built-in type within SGDK), a while loop is entered, looping the for exact number of mathematical tests to be run. Inside the loop, the appropriate function is picked from the array of function pointers, then passed the address to the time measurement variable. The general

approach for time and performance measurement applied to all of the benchmarks in this program involves passing a reference to a time variable to each function, then calculating the elapsed time within the function and setting it accordingly, to be further processed after the run. After the task was run and the time value was set, the progress bar on screen is updated and a new entry is added to the results data structure, used for later displaying the assorted data.

Once all tasks are completed and all values have been added to the results, the state function index is increased once more, executing the ending function on the next call. The ending functions of the benchmark states free allocated memory and clear changes to VRAM, but also concern themselves with result formatting and branching to the result screen, displaying the information collected during the running function. This process of collecting, converting, adding, then displaying the collected performance data is standard across the three benchmarking states present in the program.

```
// BENCH1 STATE FUNCTIONS
int prp_bench1() {
    // Draw description text
    VDP_drawText("BENCHMARK 1: CPU", 1, 1);
    VDP_drawText("     REGISTER & MEMORY BASED MATH", 1, 3);
    VDP_drawText("     [80K INSTRUCTIONS  PER TEST]", 1, 5);
    VDP_drawText("     [>                         ]", 1, 8);

    // Select function set to use
    math_ptr_ofs = (USE_ASM ? NUM_Math_Tests : 0);

    AR_States[ID_BENCH1].f_index++;
    return 0;
}

int run_bench1() {
    // Setup Tests
    fix32 t_elapsed;
    int i = NUM_Math_Tests;
    reset_results();

    // Run Tests
    while (i--) {
        math_ptr_f[(NUM_Math_Tests-i-1)+math_ptr_ofs](&t_elapsed);

        math_draw_progress_bar(NUM_Math_Tests-i-1);
        math_add_results(t_elapsed, (char*)STR_BENCH1_LBLS[NUM_Math_Tests-i-1]);
    }
    math_draw_progress_bar(-1);
    waitMs(300);
    AR_States[ID_BENCH1].f_index++;
    return 0;
}

int end_bench1() {
    // Cleanup State
    VDP_clearPlane(BG_A, TRUE);

    // Benchmark Progress
    result_title = (char*)(USE_ASM ? STR_BENCH1_TITLE_ASM : STR_BENCH1_TITLE_C);
    set_result_flow(ID_BENCH1, ID_BENCH2);
    set_state(&CUR_State, ID_RESULT);

    return 0;
}
```

Fig. 10: Benchmark 1 state function implementation

Now that the state structure surrounding the CPU benchmark is clear, the implementation of the benchmark procedure itself should come into focus. For a clearer understanding the implementation in C language will be discussed first, after which the assembly language implementation and the respective caller functions are discussed. To keep overlapping information to a minimum the following will discuss the CPU bound addition of 32-Bit (longword) values within the CPU registers. As discussed the function is passed a pointer to the variable to be filled with the time consumed by the operation. First, a few variables are initialized. Fixed-point numbers for starting and ending time, the loop counter variable and two 32-Bit unsigned numbers used to perform the additions. The loop counter variable is derived from the amount of math instructions to be executed (NUM_Math_Is), divided by the amount of individual instructions executed within one loop (NUM_Math_loop_iteration).

Before the loop of instructions begins iterating, the start time is determined by calling the SGDK function getTimeAsFix32. After this the loop begins running and will execute the exact amount of the specified instruction. Once complete, another measurement of time will be taken immediately after. The value of the elapsed time variable is then set accordingly, by subtracting the start time from the end time. Time within the SGDK context is always determined with processor ticks since initial code execution, thereby subtracting both times are aligned to system startup.

Of note here are the use of the “#pragma” compiler directive and the declaration of the two mathematical operands as “volatile”. The three compiler directives employed in the sample code serve a similar purpose as the volatile declaration. Modern C compilers are very efficient. Once a contiguous set of repeating instructions is found, they will be optimized by compiling them into fewer instructions that move more data. In the case of this extreme example, the entire loop was optimized into a few basic instructions, leading to the effective time difference being 0.00. The directives used around the function completely turn off CPU optimization for any functions implemented between them. Volatile does effectively the

same thing, but on a variable basis, being more portable. In the challenges section following the implementations, the issues regarding these optimizations will be discussed.

```
#pragma GCC push_options
#pragma GCC optimize ("O0")

void math_c_add_reg32(fix32 *te) {
    fix32 s,e; u32 i;
    volatile u32 l=getZeroU32(); volatile u32 r=getZeroU32();
    i = NUM_Math_Is / NUM_Math_loop_iteration;

    s = getTimeAsFix32(TRUE);
    while (i--) {
        l+=r; l+=r; l+=r; l+=r; l+=r; l+=r; l+=r; l+=r; //00-07
        l+=r; l+=r; l+=r; l+=r; l+=r; l+=r; l+=r; l+=r; //08-15
        l+=r; l+=r; l+=r; l+=r; l+=r; l+=r; l+=r; l+=r; //16-23
        l+=r; l+=r; l+=r; l+=r; l+=r; l+=r; l+=r; l+=r; //24-31

        l+=r; l+=r; l+=r; l+=r; l+=r; l+=r; l+=r; l+=r; //32-39
        l+=r; l+=r; l+=r; l+=r; l+=r; l+=r; l+=r; l+=r; //40-47
        l+=r; l+=r; l+=r; l+=r; l+=r; l+=r; l+=r; l+=r; //48-55
        l+=r; l+=r; l+=r; l+=r; l+=r; l+=r; l+=r; l+=r; //56-63
    }
    e = getTimeAsFix32(TRUE);
    *te = e-s;
}

#pragma GCC pop_options
```

Fig. 11: Benchmark 1 C-Implementation

The C implementation is quite readable and relatively self-explanatory. The time resulting from its execution is later divided by the amount of data moved, resulting in a value of data (kilobytes) per second, which is added and subsequently displayed on the result screen. Having now illustrated the procedure as done in the C language, the remaining step for this benchmark is to look at its implementation in the 68000 assembly language. The two functions listed below do not in fact stem from the same file. As described previously, assembly functions are implemented in their own file type, then referenced in a header file for usage within a C context. The above caller function is part of the first benchmarks logic, accessing the separate assembly function bellow via the included asm.h header file.

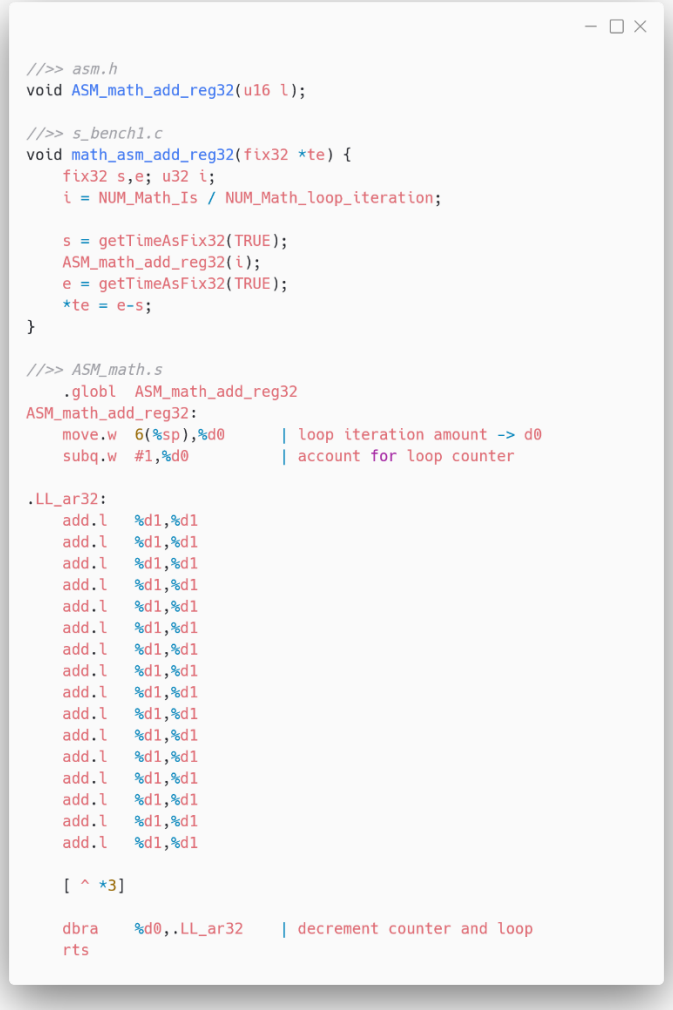
The caller function for the assembly implementation of the 32-Bit CPU register addition is responsible for measuring the time, as the provided SGDK time functionality would be superfluously re-implemented in 68000 assembly. Instead, the same structure of the time measurement setup is used as in the previous C implementation. Only instead of the while loop

executing the additions in C, the assembly routine is called, which loops the operations within itself. Following is the observation and explanation of its structure.

To begin with, the lower word of the first variable passed to the assembly routine is moved into data register 0 (d0). The declaration of the routine from asm.h declared it to receive a 16-Bit number (word) as its first parameter. The stack pointer (sp) points at the program stack, starting at the memory reserved for the function call. The top word of the first (32-Bit) parameter starts 4 bytes ahead of the current stack pointer position. As such, 2 additional bytes ahead of this position lies the lower word of the first argument, which is why the address of the stack pointer is read with an offset of 6. This value is the iteration amount, which just like previously was precalculated to fit a loop of containing a set amount of instructions (64 in this case). Once read, a literal value of 1 is subtracted from data register 0. This accounts for the loop offset, as one more loop would be executed compared to a while loop in C. Once the register is set up, a unique loop label is defined to branch back to later.

After this, longword (32-Bit) add operations are performed on data register 1 (d1), 64 times (one loop amount). While d1 is empty, the operation will not be faster, as in assembly any operation take a fixed time and will not be optimized for its contents. As such, filling d1 with a value is superfluous. Once the 64 operations are complete, a decrement-branch (dbra) instruction is encountered, instructing the processor decrease the value of d0 (loop iteration counter) and loop back to the defined label, until d0 has reached the value of 0, after which it will call rts (return from subroutine)

The time spent within this assembly implementation is then calculated as before in the C implementation, once the assembly routine returns the execution. Results are then prepared and added to the results display just as before.



```
//>> asm.h
void ASM_math_add_reg32(u16 l);

//>> s_bench1.c
void math_asm_add_reg32(fix32 *te) {
    fix32 s,e; u32 i;
    i = NUM_Math_Is / NUM_Math_loop_iteration;

    s = getTimeAsFix32(TRUE);
    ASM_math_add_reg32(i);
    e = getTimeAsFix32(TRUE);
    *te = e-s;
}

//>> ASM_math.s
    .globl  ASM_math_add_reg32
ASM_math_add_reg32:
    move.w  6(%sp),%d0      | loop iteration amount -> d0
    subq.w  #1,%d0          | account for loop counter

.LL_ar32:
    add.l   %d1,%d1
    add.l   %d1,%d1
    add.l   %d1,%d1
    add.l   %d1,%d1
    add.l   %d1,%d1
    add.l   %d1,%d1
    add.l   %d1,%d1
    add.l   %d1,%d1
    add.l   %d1,%d1
    add.l   %d1,%d1
    add.l   %d1,%d1
    add.l   %d1,%d1
    add.l   %d1,%d1
    add.l   %d1,%d1
    add.l   %d1,%d1
    add.l   %d1,%d1

    [ ^ *3]

    dbra    %d0,.LL_ar32    | decrement counter and loop
    rts
```

Fig. 12: Benchmark 1 Assembly-Implementation

The time spent within this assembly implementation is then calculated as before in the C implementation, once the assembly routine returns the execution. Results are then prepared and added to the results display just as before.

Memory Benchmark implementation

Before moving on to the memory benchmark it should be noted that there are small differences in the state function implementation for each benchmark state, as all of them check for different things and prepare / clean up different data for the tasks to run. However, the structure and overall logic behind their implementation is the same and easily graspable in the full source code, if the previous methodology was of state setup was understood. As such the following explanations will only concern themselves with the implementations of the benchmarks themselves.

While the previous benchmark only concerned itself with the CPU operations, how quickly large portions of addressable memory can be interfaced with is often a much more common problem in the implementation of a full game on such a limited platform. Especially once a game requires multiple data buffers to be allocated, filled and accessed, the throughput on memory-centric operations is crucial, as such the second benchmark deals with memory access on the SEGA Mega Drive.

Luckily, direct manipulation of memory buffers is something the C language was designed to aid with, as such there are many basic C functions intended for just this purpose. The memset function allows for setting a certain amount of bytes of a given buffer to a given value. With these three parameters, a whole buffer (of part of it depending on amount) can be initialized, cleared or generally filled with a desired value. The memcpy function equally does what it sounds like. For this function, instead of giving it a value to set the buffer to, a second buffer (or same buffer with offset) is passed, facilitating a memory-to-memory copy, upping the needed memory access.

After handling the allocation of a sizable buffer in memory (using the SGDK specific MEM_alloc implementation of the standard C malloc), three tests each of the two described memory functions are run. Each time the respective memory function is executed 10000 times in succession, with the difference being the amount of bytes to set or copy, going from 256 to 1024 and for the last test to 4096, moving up in powers of 2.

The functionality of measuring the time taken, conversion to data over time representation and display via the results screen remains virtually the same across these benchmarks, as such the structure of the memset and memcpy benchmark loops should be familiar by now.

```c
u8 *mem_buf = MEM_alloc(VAL_Mem_Buf);

// BENCH2 TEST FUNCTIONS - C
void mem_c_loop_memset(fix32 *te, u8 *buf, int amnt, int size) {
    fix32 s,e; u32 i = amnt;

    s = getTimeAsFix32(TRUE);
    while (i--) { memset(buf, 0, size); }
    e = getTimeAsFix32(TRUE);
    *te = e-s;
}

void mem_c_loop_memcpy(fix32 *te, u8 *buf, int amnt, int size, int
of) {
    fix32 s,e; u32 i = amnt;

    s = getTimeAsFix32(TRUE);
    while (i--) memcpy(buf, buf+of, size);
    e = getTimeAsFix32(TRUE);
    *te = e-s;
}
```

Fig. 13: Benchmark 2 C-Implementation

Like previously, the implementation of the 68000 assembly equivalent to these functions are called via C wrapper functions. These call the assembly routines and measure their time, reporting that time back to the result display via state logic. In fact, the assembly callers for this benchmark only differ in a the single line calling them.

Jumping into the assembly code for the memset benchmark, the first more complex implementation in 68000 assembly is revealed. For those unfamiliar with assembly code, blocks of instructions like these can be intimidating at first, but their functionality is easy enough to grasp. Like before in the CPU benchmark, the assembly code begins with moving passed variables from the stack to the CPU registers to be worked on. This time the address to the buffer which we want to set is loaded into address register 0 (a0), then also duplicated to address register 1 (a1), to keep an unaltered buffer position available. Lastly the amount of repetitions and length (in bytes) of buffer to fill are moved into data register 3 (d3) and data register 1 (d1) respectively. Like before each is subtracted by 1 to account for the loop offset, however next a few more complicated instructions are executed.

To explain the coming operations we first need to establish a few basic rules of optimization. While the C language memset and memcpy functions take the amount of bytes to be set as

arguments, the internal logic does not simply write one byte at a time. Each memory movement instruction takes a baseline of time, regardless of how much memory is actually moved, meaning that looping over the given buffer and moving each byte of the desired length in one instruction would be terribly inefficient. The largest single move operation possible in 68000 assembly is a longword (32-Bit), equivalent to 4 bytes (8-Bit). A simple optimization for the process of writing this memory would be to move the desired byte value 4 times at once, with the value occupying each byte of the longword. This is bound to increase performance, however it also leads to the problem of rounding. If the wanted byte count to set in the buffer is not divisible by 4, there will be either too many or too few bytes set, an unacceptable implementation.

Luckily the 68000 instruction set (and many CPU instruction sets) are designed around common problems like this. Like previously said, if 4 bytes can be moved at once in one longword, the amount of bytes to move (length to set in buffer) would have to be divided by 4. The 68000 (unsigned) division instruction, also used during the math benchmark, will have to be used here which offers extra functionality, that someone coming from higher level languages might not expect. When the division is processed, the quotient is saved in the lower word (2 bytes) as requested, but the remainder is stored in the upper word (2 bytes). By executing the 68000 swap instruction, the two words of any register can be swapped to easily read how many excess bytes are left to be set, after all pairs of 4 bytes are set. This way two loops can be constructed, the first iterating over the amount of longwords (4 bytes) to set, the next finishing the job by looping over the remaining single bytes and setting them accordingly.

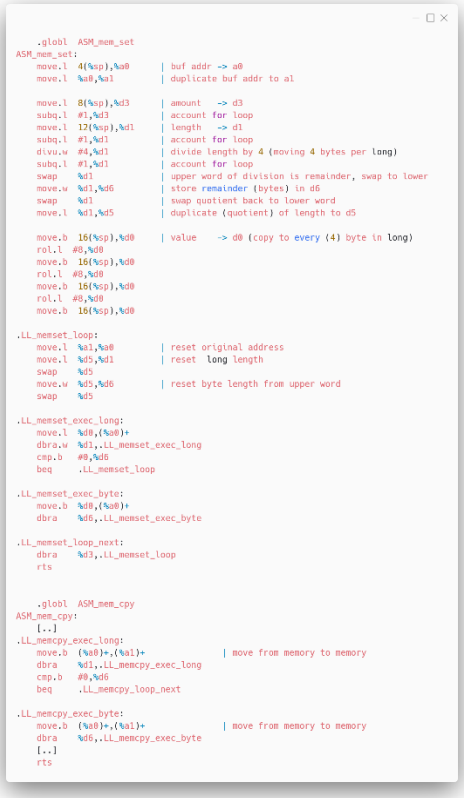
One more look at the assembly code reveals that exactly this is done here. The length in d1 is divided by a literal value of 4 and then swapped to move the remaining bytes into data register 6 (d6). Now that counters for both loops are present, the longword of 4 byte values needs to be constructed. This is simply done by moving the value from the stack into data register 0 (d0) and then rotating (shifting) the bits in d0 left by 8 (1 byte worth). Now the value from the stack is loaded into the lower byte again, repeating 4 times, until all bytes of the register are full.

Now that everything is prepared, loop labels are defined. Finally, iteration over the amount of 4-byte transfers starts, followed by the transfer loop of the single bytes if the division had a remainder. After this, the loop restarts as often as repetitions were given to the function (10000), after which we return out of the assembly subroutine, back into the C code, like in the previous benchmark

The assembly code for the memcpy implementation is virtually identical, with the difference of taking an offset into the specified buffer and copying the single bytes and 4-byte pairs from one memory location to the other, rather than a value to a memory location. This process of learning to optimize assembly code by using extra functionality built into the instruction set is a large part of improving at homebrew development and gaining a greater understanding of the possibilities and potential bottlenecks.

```c
//>> asm.h
void ASM_mem_set(u8 *buf, u16 amnt, u16 l, u16 v);
void ASM_mem_cpy(u8 *buf, u16 amnt, u16 l, u16 of);

//>> s_bench2.c
u8 *mem_buf = MEM_alloc(VAL_Mem_Buf);

// BENCH2 TEST FUNCTIONS - ASM
void mem_asm_loop_memset(fix32 *te, u8 *buf, int amnt, int size) {
    fix32 s,e; u32 i = amnt;

    s = getTimeAsFix32(TRUE);
    ASM_mem_set(buf, i, size, 0);
    e = getTimeAsFix32(TRUE);
    *te = e-s;
}

void mem_asm_loop_memcpy(fix32 *te, u8 *buf, int amnt, int size, int of) {
    fix32 s,e; u32 i = amnt;

    s = getTimeAsFix32(TRUE);
    ASM_mem_cpy(buf, i, size, of);
    e = getTimeAsFix32(TRUE);
    *te = e-s;
}
```

Fig. 14: Benchmark 2 Assembly-Caller



```
    .globl  ASM_mem_set
ASM_mem_set:
    move.l  4(%sp),%a0      | buf addr -> a0
    move.l  %a0,%a1         | duplicate buf addr to a1

    move.l  8(%sp),%d3      | amount   -> d3
    subq.l  #1,%d3          | account for loop
    move.l  12(%sp),%d1     | length   -> d1
    subq.l  #1,%d1          | account for loop
    divu.w  #4,%d1          | divide length by 4 (moving 4 bytes per long)
    subq.l  #1,%d1          | account for loop
    swap    %d1             | upper word of division is remainder, swap to lower
    move.w  %d1,%d6         | store remainder (bytes) in d6
    swap    %d1             | swap quotient back to lower word
    move.l  %d1,%d5         | duplicate (quotient) of length to d5

    move.b  16(%sp),%d0     | value    -> d0 (copy to every (4) byte in long)
    rol.l   #8,%d0
    move.b  16(%sp),%d0
    rol.l   #8,%d0
    move.b  16(%sp),%d0
    rol.l   #8,%d0
    move.b  16(%sp),%d0

.LL_memset_loop:
    move.l  %a1,%a0         | reset original address
    move.l  %d5,%d1         | reset  long length
    swap    %d5
    move.w  %d5,%d6         | reset byte length from upper word
    swap    %d5

.LL_memset_exec_long:
    move.l  %d0,(%a0)+
    dbra.w  %d1,.LL_memset_exec_long
    cmp.b   #0,%d6
    beq     .LL_memset_loop

.LL_memset_exec_byte:
    move.b  %d0,(%a0)+
    dbra    %d6,.LL_memset_exec_byte

.LL_memset_loop_next:
    dbra    %d3,.LL_memset_loop
    rts


    .globl  ASM_mem_cpy
ASM_mem_cpy:
    [..]
.LL_memcpy_exec_long:
    move.b  (%a0)+,(%a1)+           | move from memory to memory
    dbra    %d1,.LL_memcpy_exec_long
    cmp.b   #0,%d6
    beq     .LL_memcpy_loop_next

.LL_memcpy_exec_byte:
    move.b  (%a0)+,(%a1)+           | move from memory to memory
    dbra    %d6,.LL_memcpy_exec_byte
    [..]
    rts
```

Fig. 15: Benchmark 2 Assembly-Implementation

Sprite Benchmark implementation

As outlined previously, the third benchmarks is concerned with sprites. Sprites are somewhat of a foreign concept nowadays, but simply describe small tilemaps referencing the same tile-set used by all background graphics, which can be freely moved on a pixel basis, used to represent anything that moves or is to be animated. Everything from the player to the enemies, items, effects and more were created with sprites on 8-Bit and 16-Bit systems. With a limited amount of sprites on screen and a premium on performance, how many sprites to use and how much memory to reserve for their graphics was a constant concern in the 16-Bit era.

To test out the graphical capabilities of SGDK and attempt to implement the same functionality by hand in 68000 assembly, a small test concept was introduced, oriented towards the specifics of the SEGA Mega Drive's video chip (VDP) capabilities. The sprite benchmark consists of 3 tests, each repeated once. Each of the three tests loads a set amount of predefined sprites into the VDP's sprite table, then moves them around the screen with simple integer velocities both horizontally and vertically. Each of the three tests uses a different size of sprite, going from 8*8 pixels (1*1 tiles) to 16*16 pixels (2*2 tiles) to 32*32 pixels (4*4 tiles, maximum sprite size on Mega Drive). Sprite graphics for all theses sizes are converted with SGDKs resource compiler, then loaded into VRAM in the preparation function of this benchmark state. In the first run of the three tests, the sprites are only to collide with walls, simply bouncing off the screen bounds, while in the second run the sprites will collide with both the screen border and each other, increasing computational load by a lot. In general, the computations and hardware-access required for this benchmark were a lot more complex than the previous benchmarks, raising questions as to feasibility both on hardware and relevant skill basis.

Of note as well is that one of the SGDK's largest achievements is the compartmentalization of accessing the video chip. The high-level functions available within SGDK make working with graphical elements such as tiles, plane scrolling and sprites much easier to work with and implement than the low-level assembly method of access. Additionally in the case of sprites, SGDK handles many intricate methods of memory and VRAM conservation automatically. For instance, SGDK will by default stream only the required sprite graphics for the current frame of animation into a specific section of VRAM, to save on VRAM space

taken up by sprite graphics. While a smart thing to enable by default for newcomers, this inherently slows performance when many calculations are performed on the sprites. For this the automatic sprite handling can be turned off, allowing the user to still use the comfortable high-level functions of SGDK, but simply setting the tile index in VRAM to the preloaded sprite graphics to be displayed, freeing up processing time. For a benchmark without many assets, this is a clear choice and would also prove to be more easily implemented in 68000 assembly, as the streaming of assets adds much complexity.

One important note about working with sprites within SGDK is that sprites themselves should never be used to store values about themselves. Reading from and writing to the sprite table in VRAM is very slow, as such a separate structure with all needed values for a sprite (position, velocity, size, …) should be constructed and used to do movement and collision operations, after which the newly determined position or animation frames should be written to the VRAM. In the related implementation, this structure is refereed to as “sprite entities”, instanced by spr_ents[ ]

Beginning with the C implementation of this proposed benchmark requires a few things to be set up first. The SGDK sprite system is initialized with SPR_init() and the sprite reference definitions to use, along with sprite sizes and tile indexes are determined by the passed values. To setup the sprite table list inside VRAM, a for loop begins iterating for the amount of desired sprites, calling the SPR_addSprite(..) function and passing the previously deducted values. Additional functions are called right below to turn off the aforementioned sprite engine features, to be set manually later. Still within the same loop, the sprite entities representing the sprites physical data are initialized as well, receiving random positions, random integer velocities, size in pixels and current animation frames. Interestingly the SGDK sprite implementation contains an unsigned 32-Bit (longword) integer value named “data”, which serves no specific purpose. A pointer to the associated sprite entity is stored here during the above sprite initialization process. This can help tremendously for debugging, as any misbehaving sprite’s internal sprite entity can be easily identified by dereferencing its data value. After all sprites and their representative entities are setup, the benchmark itself can begin.

At the beginning of the benchmarking process the predetermined runtime of each test is calculated by adding the desired time to the current time, then looping until the current time has

reached that value. Within the main loop, another loop is constructed iterating over the amount of present sprites to be processed. Pointers to the current sprite and sprite entity are defined to begin with, then all needed operations on both can commence. Depending on the specific sprite run and the values passed to the function, the sprite entities are checked against either just the screen bounds or both the screen bounds and against each other for collision. The collision logic takes velocity into account and checks the vertical and horizontal axis separately, achieving very accurate modern collision, at the expense of performance. Pushing the capabilities is more important for this test than perfectly optimized game-ready implementations. After collisions are processed and velocities have been set accordingly, all sprite entity positions are moved by their velocity, after which the new positions calculated during this process are set in the sprites themselves, using SPR_setPosition(..).

Lastly, the entities animation frame variable is increased, then logically AND-ed by 1, essentially checking if the number is odd or even. This might sound strange but is a common, performant practice to create an expression that holds true every other frame. This way, every other frame the tile offset for the next animation frame is calculate and then set using SPR_setVRAMTileIndex(..). This effectively animates all sprites every 2 frames, resulting in 60 FPS movement and 30 FPS animation, a trick which was revealed by a credited member of the SGDK discord server. Once out of the inner loop iterating all sprites and entities, the SGDK sprite system update function is called with SPR_update(), followed by the SGDK system call to wait for the VBlank and perform DMA transfers with SYS_doVBlankProcess().

Once returned from the vertical blank interrupt, the last thing inside the outer while loop is the performance calculation. For this the SGDK system function SYS_getCPULoad() is used, returning the estimated processing load of the main CPU, determined by the amount of time spent waiting for the vertical blank interrupt during the last SYS_doVBlankProcess() call. This value represents the needed processing load, not the free time that was left, which is much more telling. To get this value, the received value is subtracted from the next higher multiple of 100, then bitshifted to the right (or divided by a power of two) accordingly. This last step is done, because values over 100 represent that more CPU load was needed than available, cutting the framerate in half. This is the famous “slow down” present in many 16-

Bit games when too many sprites are on screen. Accounting for these to not count as full free frames is obviously vital for accurate results, hence the subtraction and division. The values arrived at from this operation are then all added to a containment variable, which gets divided by the time spent after the outer while loop finishes. This way the free CPU load amount per second is calculated.

Fig. 16: Benchmark 3 C-Implementation

Keeping the benchmark implementations to one function each was one of the initial goals to upkeep, which at this point became apparent to not be such a great idea, as the amount of logic required for an involved sprite test such as this one was more than initially expected. More concerns started mounting as the difficulty of an assembly implementation for such a task became clear. It is unfortunate to announce that the full implementation of this

benchmark was out of feasible reach for the level of assembly language skill acquirable in the time frame.

However, the low-level hardware research conducted and the level of assembly skill that indeed was acquired certainly sufficed to show and explain how most individual components of such a sprite implementation would work. In the end, creating the sprite entities in assembly, loading sprites into the VRAM sprite table using assembly and clearing said table equally in assembly were all implemented, if not quite put to full use with movement, collision and performance metrics.

Firstly, a few hexadecimal and binary constant values are defined. The first two values represent the hexadecimal addresses of the memory mapped control and data port of the Mega Drive video chip (VDP) on the system bus. The third entry is the default VRAM address of the sprite table used within SGDK projects. The following block of values are base VDP commands for reading and writing to the three different types of VDP controlled memory: VRAM (Main video ram. Tiles, tilemaps, sprites), CRAM (Palette data), VSRAM (vertical scroll values for tilemaps). The last value is simply a bitmask to force the third palette to be used.

```
|;;================= CONSTANTS  ================;;

#define VDP_CONTROL 0x00C00004
#define VDP_DATA 0x00C00000

#define VDP_SPR_TBL_ADR 0x0000F400

#define VDP_CRAM_READ 0x20000000
#define VDP_CRAM_WRITE 0xC0000000
#define VDP_VRAM_READ 0x00000000
#define VDP_VRAM_WRITE 0x40000000
#define VDP_VSRAM_READ 0x10000000
#define VDP_VSRAM_WRITE 0x14000000

#define SPR_PAL_WORD 0b0110000000000000
```

Fig. 17: Benchmark 3 Assembly constants

These addresses are used to initiate any interaction with the video memory, via the VDP control port. However, addressing any wanted VRAM address, say the table address listed as the third item, is not as easy as one might suspect. The VDP control port accepts these addresses in a very specific order of bits, requiring the desired address to be processed in a unique way. Specifically, the wanted VRAM address needs to occupy bits 1-0 in the command (highest bits of address) and bits 29 – 16 (remaining lower bits of address) in the command. Additionally bits 31-30 and 7-4 in the command also act as flags for which type of video memory (see above) to be accessed. These are the bits preset for the block of *_READ and *_WRITE commands found in the constants declaration. To not go through the labor of performing these operations at every given point where video memory is accessed, writing a subroutine or macro for this task is essential to maintain a usable set of routines. The following is an explanation for the macro written as part of the attempted implementation:

Firstly, the values of data register 2 (d2) and data register 7 (d7) are pushed onto the memory stack, ensuring their value will be able to be fetched again once operations are complete. D2 is vital in this case, as it always acts as SGDKs time counter, possibly corrupting any time calculations that might be done after returning from the assembly code. After calculating the table offset, the wanted address (16-Bit word) given to the macro is bitshifted to the right by 14, leaving the highest two bits in the lowest two bits. This value is then also pushed onto the stack for later reference. The original address is loaded into the same register again, but this time it is bitmasked (logically AND-ed) with 0x3FFF, equivalent to a 16-Bit word, missing the first two bits. By performing this operation, then swapping the lower and higher word with the swap command, we have now brought the wanted address into the format understandable to the VDP, with the highest two address bits at the bottom and the remaining address bits offset by two from the top of the longword register. The saved address offsets are then taken back from the stack and logically OR-ed with the base write address for VRAM. Resulting is a command that will initiate a write to that specific address in VRAM, that is sent to the VDP control port. After the previous values of d2 and d7 are loaded back in, the macro ends and the VDP data port can then be given then data to written to the address desired.

This is indeed a very cryptic method of memory access, bringing a whole new level of respect, but also understanding of the logistics surrounding assembly development. The process of figuring this out would not have been possible without the great documentation on the "Sega Retro" website regarding the VDP, as well as the previously mentioned blog of Hugues Johnson, covering sprite manipulation on the hardware. Nonetheless, the macro implemented as part of this research paper does work for quickly writing to a simply noted address in VRAM.

```
.macro VDP_DO_VRAM_WRITE adr regof
    move.l  %d7, -(%sp)             | push used register values onto the stack
    move.l  %d2, -(%sp)

    move.l  %\regof, %d2            | multiply offset by 8 (size of sprite definition)
    mulu.w  #8, %d2

    move.l  #\adr, %d7              | shift wanted address to the right (bits 0-1)
    add.l   %d2, %d7
    asr.l   #8, %d7
    asr.l   #6, %d7
    move.l  %d7, -(%sp)             | push result onto stack

    move.l  #\adr, %d7              | shift wanted address to the left and bitmask (bits 2-13)
    add.l   %d2, %d7
    and.l   #0x3FFF, %d7
    swap    %d7
    move.l  %d7, -(%sp)             | push result onto stack

    move.l  #VDP_VRAM_WRITE, %d7    | load base VRAM write address into d7

    or.l    (%sp)+, %d7             | bitwise or with address offsets
    or.l    (%sp)+, %d7

    move.l  %d7, (VDP_CONTROL)      | send constructed command to VDP control port
    move.l  (%sp)+, %d2             | pop used register values back from the stack
    move.l  (%sp)+, %d7
.endm
```

Fig. 18: Benchmark 3 Assembly macro

With the research conducted about the hardware and its workings, as well as the findings about the Mega Drive's video chip address formats, a few simple functions to at least approach the desired sprite benchmark were implemented, learning much about assembly stack pointer operation, VDP sprite table entry addressing and overall program structure within 68000 assembly in the progress. The ASM_sprites_setup routine creates a desired amount of sprite entity equivalent values inside a given data buffer, which the ASM_sprites_setup_table routine then acts on. This large routine constructs the VDP sprite table entries by iterating

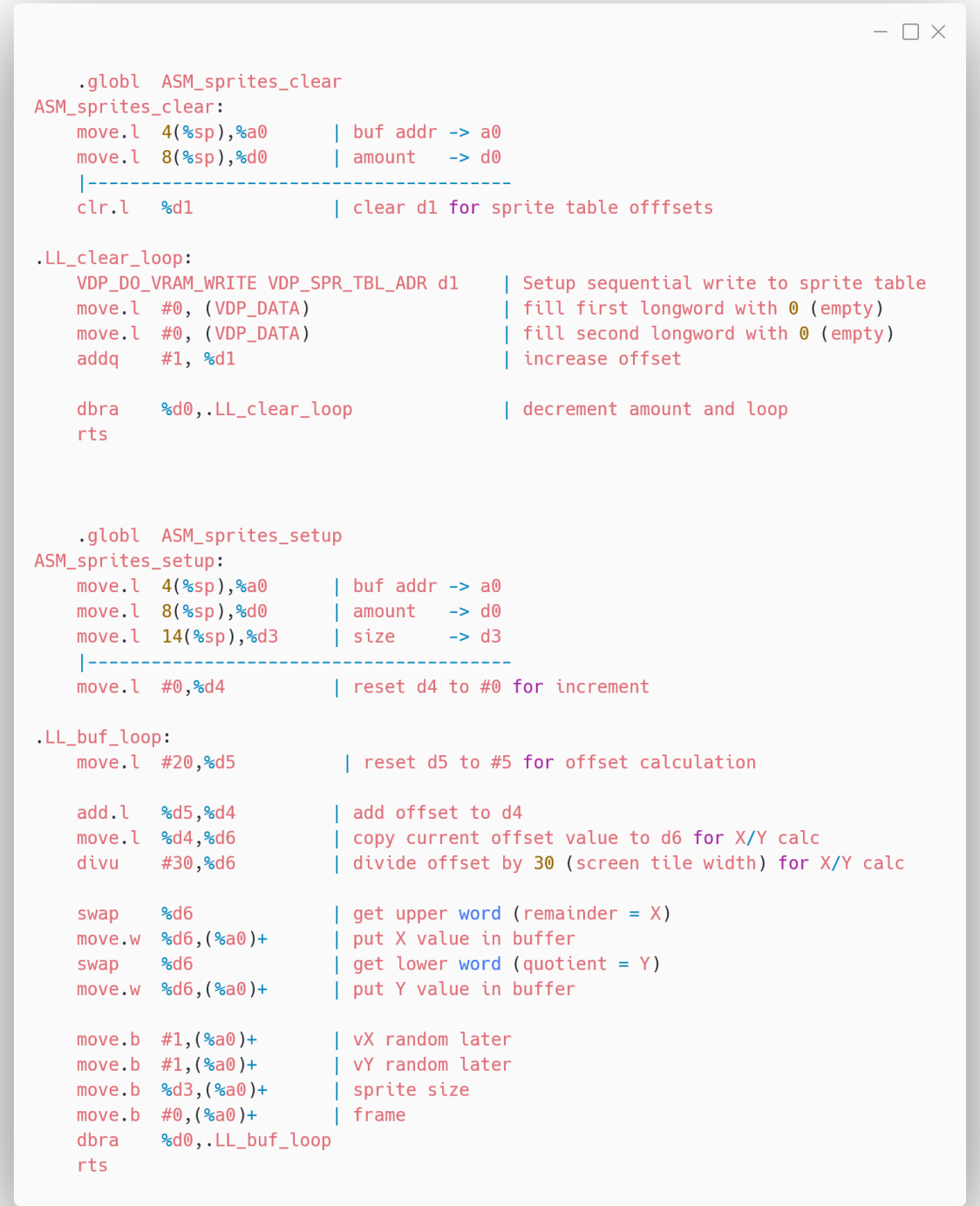
over the filled entity buffer and manipulating the values according to the required bit patterns. Of note here is the sprite link field. The third byte of a sprite table entry represents which sprite to draw next (below it), implementing essentially a singly linked list. If a sprite index is skipped, the sprite at that index will not be drawn, making this step crucial in displaying all wanted sprites.

Once the two longwords (8 Bytes) for such an entry are completed, the previously discussed macro is given the sprite table address and an offset, setting up the VRAM write for the current sprite entry, after which both longword values are passed into the VDP data port, facilitating the sprite table write. This is iterated for the given amount of sprite entity data in the buffer. While not near the quality of implementation hoped for, the desired sprites are rendered correctly with their desired tile patterns, palette and render queue position intact. Finally the ASM_sprites_clear routine uses the same macro to write two longwords worth of zeros into every slot of the sprite table, effectively clearing all sprites, as to not interfere with the SGDK sprite system when returning to C code.

While not the intended quality of implementation, the knowledge gained and the level of functionality achieved with the skills acquired still exemplifies an understanding of the hardware, which given more time and experience could easily be expanded to achieve much more complex tasks. After all, the premise of this very paper was pushing the limits of both hardware performance and feasibility of time restricted low-level 68000 assembly implementation. For interested students without much prior experience, this should set a decent baseline for what to expect from such an undertaking.

```
    .globl  ASM_sprites_setup_table
ASM_sprites_setup_table:
    move.l  4(%sp),%a0      | buf addr  -> a0
    move.l  8(%sp),%d0      | amount    -> d0
    move.l  12(%sp),%d3     | size      -> d3
    move.l  16(%sp),%d4     | base tile -> d4
    |---------------------------------------
    // Prepare size
    divu    #8, %d3           | divide size by 8 to get representation 1 - 3
    swap    %d3               | clear remainder from long
    clr.w   %d3
    swap    %d3
    subq    #1, %d3           | substract 1 => 0-2 / 00, 01, 11
    move.w  %d3, %d6          | copy to d6 as buffer
    rol.w   #2, %d3           | shift 2 to the left for horizontal size
    or.w    %d6, %d3          | or with d6 to merge horizontal and vertical
    rol.l   #8, %d3           | shift to second byte of lower word

    move.l #0, %d1

.LL_table_main:
    // Prepare first long
    move.l  2(%a0), %d5       | load Y value
    mulu    #8, %d5           | multiply from tile to pixels
    add.w   #0x80, %d5
    swap    %d5               | sawp to higher word of long

    move.w  %d1, %d6          | setup link value pointing to next index
    addq    #1,  %d6
    cmpi    #0, %d0
    bne     .LJ_not_last
    clr.l   %d6               | set link value back to 0 for last sprite (complete circle)
.LJ_not_last:
    or.l    %d3, %d5
    or.l    %d6, %d5          | or results together into d5 (d6 now free to use again)

    // Prepare second long
    move.l  #SPR_PAL_WORD, %d6 | load base palette word without tile index
    or.l    %d4, %d6          | or wit tile index to get needed value word
    swap    %d6               | swap to higher word for offset

    move.l  (%a0), %d7        | load X value
    mulu    #8, %d7           | multiply from tile to pixels
    add.w   #0x80, %d7
    or.l    %d7, %d6          | or results together into d6 => d5 upper sprite def, d6 lower sprite def

    VDP_DO_VRAM_WRITE VDP_SPR_TBL_ADR d1    | Setup VRAM sprite table write
    move.l %d5, (VDP_DATA)                  | write first longword of calculated sprite definition
    move.l %d6, (VDP_DATA)                  | write second longword of calculated sprite definition

    addq    #1, %d1             | increase sprite definition offset
    adda    #8, %a0             | increase entity buffer offset

    dbra    %d0, .LL_table_main  | decrement amount and loop
    rts
```

Fig. 19: Benchmark 3 Assembly-Setup-Table



```
    .globl  ASM_sprites_clear
ASM_sprites_clear:
    move.l  4(%sp),%a0      | buf addr -> a0
    move.l  8(%sp),%d0      | amount   -> d0
    |---------------------------------------
    clr.l   %d1             | clear d1 for sprite table offfsets

.LL_clear_loop:
    VDP_DO_VRAM_WRITE VDP_SPR_TBL_ADR d1    | Setup sequential write to sprite table
    move.l  #0, (VDP_DATA)                  | fill first longword with 0 (empty)
    move.l  #0, (VDP_DATA)                  | fill second longword with 0 (empty)
    addq    #1, %d1                         | increase offset

    dbra    %d0,.LL_clear_loop              | decrement amount and loop
    rts



    .globl  ASM_sprites_setup
ASM_sprites_setup:
    move.l  4(%sp),%a0      | buf addr -> a0
    move.l  8(%sp),%d0      | amount   -> d0
    move.l  14(%sp),%d3     | size     -> d3
    |---------------------------------------
    move.l  #0,%d4          | reset d4 to #0 for increment

.LL_buf_loop:
    move.l  #20,%d5          | reset d5 to #5 for offset calculation

    add.l   %d5,%d4         | add offset to d4
    move.l  %d4,%d6         | copy current offset value to d6 for X/Y calc
    divu    #30,%d6         | divide offset by 30 (screen tile width) for X/Y calc

    swap    %d6             | get upper word (remainder = X)
    move.w  %d6,(%a0)+      | put X value in buffer
    swap    %d6             | get lower word (quotient = Y)
    move.w  %d6,(%a0)+      | put Y value in buffer

    move.b  #1,(%a0)+       | vX random later
    move.b  #1,(%a0)+       | vY random later
    move.b  %d3,(%a0)+      | sprite size
    move.b  #0,(%a0)+       | frame
    dbra    %d0,.LL_buf_loop
    rts
```

Fig. 20: Benchmark 3 Assembly-Sprites-Setup / Assembly-Clear

Challenges Faced

During development of the related benchmark program for the SEGA Mega Drive, multiple hurdles and challenges were met, overcome or eventually compromised with. Namely the unexpected difficulty of the assembly implementation of the sprite based benchmark left the overall expected progress fall behind a desired point to reach, however the extra time spent trying to overcome the issue and learning from the failures along the way have certainly resulted in a much wider perspective of the development process for such a limited system at large.

Besides the turbulent development cycle in 68000 assembly, SGDK also came with a few challenges itself. Relatively close to the final state of the benchmark, the ordered SEGA Mega Drive flash cart finally arrived, sadly revealing that the benchmark up to this point did not run on the original hardware without crashing. Following this discovery a deeper dive into the available emulators and their accuracy was conducted, resulting in the thorough gathering of benchmark data presented in the next chapter. Eventually, the problem emerged to be a memory freeing operation, performed on an uninitialized buffer. This was not a problem on a debugging friendly emulator, but caused a crash on real hardware and more accurate emulators. Once the problem was resolved and the assembly implementation was pushed as far as the capabilities were able to, the rest of the implementation went without much trouble.

EVALUATION

Collecting and structuring data

Once the implementation had concluded, the first step to take towards evaluating the resulting findings was to collect the relevant data produced by the benchmark. This was a rather straight-forward process, as the benchmarks internal structure was designed around showing an easily accessible result screen for every individual benchmark run. As part of the research for the sate of the art representation for the second chapter of this paper, multiple emulators were inspected for their features, quality of emulation and representative usage withing the retro gaming and homebrew development communities. Having another critical look at the available emulators, four of them were chosen to represent a wide range of SEGA Mega Drive emulation standards, from simple, to highly technical and highly accurate.

These emulators in order of later listing are “Kega Fusion”, “Gens”, “BlastEM” and “Exodus”. The fifth entry in the later listing is occupied by a real SEGA Mega Drive (Model 1), running the same executable ROM file via an “Everdrive” flash cartridge. As the German distribution model of the console was used during the test, the benchmark ran in PAL 50Hz mode, prompting all emulators to be exclusively operated in PAL mode as well for the data collection conducted. After every benchmark in both C / SGDK and 68000 assembly implementation were run within all emulators and the physical hardware, all results were captured and organized into adjacent image files. These images were taken for every benchmark run and the remaining images not displayed below will be present in the repository containing all relevant work conducted for this research paper.

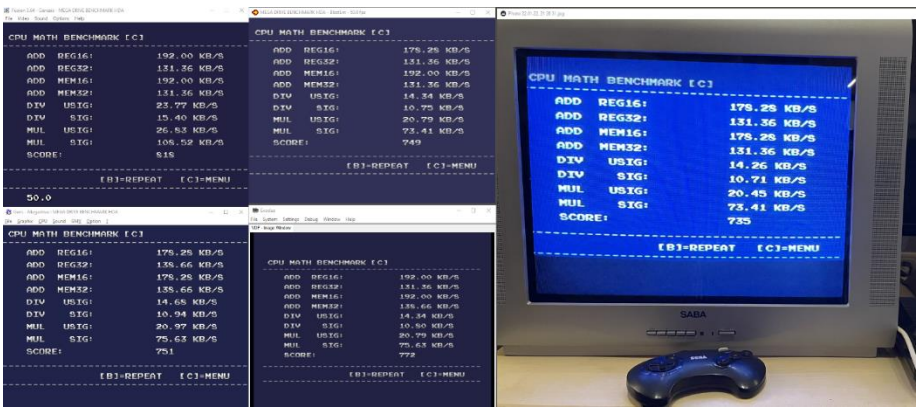


Fig. 21: Value aggregation benchmark runs

After all images had been curated, the displayed values for each run on each execution platform were copied into a Microsoft Excel sheet and categorized by run and platform, resulting in one table each for every benchmark executed, with the values from all execution platforms neatly comparable. Using the graphing functionality of Microsoft Excel, diagrams and graphs representing the collected data sets were generated and color-coded. With a diagram each for both the C / SGDK and assembly implementation of each benchmark, as well as a line graph comparing the two approaches directly, the resulting observations will be discussed in the upcoming section.

To gain better insights into the time spent in certain parts of execution and to visualize the overall flow of the program, the aforementioned MD-Profiler by GitHub user Tails8521 was used to create a full profiling trace, using a modified version of the BlastEm emulator to record a basic trace file, followed by the usage of the MD-Profiler tool itself to convert the file into a viewable JSON file for use with a wide range of profiling viewers, including the profiler built into the Google Chrome or Chromium browsers. Comparing written code to execution times of certain functions in the visual tracing representation would proof to be a viable source of development information as well.

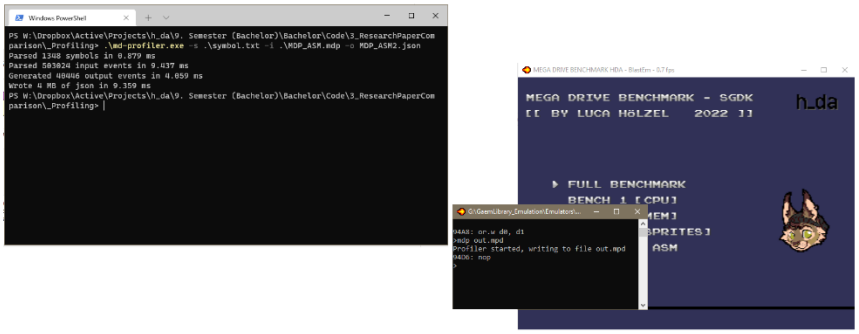


Fig. 22: Generating JSON trace with MD-Profiler and BlastEM

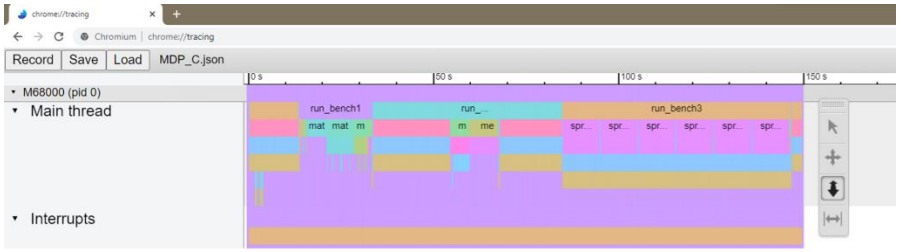


Fig. 23: Loading JSON trace with browser profiling viewer

Comparing formatted data

After creating all tables and generating the resulting diagrams and graphs, comparing the assorted data should hopefully provide insight into achieved performance, differences between C / SGDK and assembly approaches and also highlight differences between different emulators and the original SEGA Mega Drive hardware. Firstly, the CPU math benchmark results will be examined:

C / SGDK - CPU Math Benchmark (PAL)

Environment		Add Reg 16	Add Reg 32	Add Mem 16	Add Mem 32	Div Unsigned	Div Signed	Mul Unsigned	Mul Signed	Total
(Emu) Kega Fusion	(KB/s)	192.00	131.36	192.00	131.36	23.77	15.40	26.83	108.52	821.24
(Emu) Gens	(KB/s)	178.28	138.66	178.28	138.66	14.68	10.94	20.97	75.63	756.10
(Emu) BlastEM	(KB/s)	178.28	131.36	192.00	131.36	14.34	10.75	20.79	73.41	752.29
(Emu) Exodus	(KB/s)	192.00	131.36	192.00	138.66	14.34	10.80	20.79	75.63	775.58
(HW) Mega Drive 1	(KB/s)	178.28	131.36	178.28	131.36	14.26	10.71	20.45	73.41	738.11

ASM - CPU Math Benchmark (PAL)

Environment		Add Reg 16	Add Reg 32	Add Mem 16	Add Mem 32	Div Unsigned	Div Signed	Mul Unsigned	Mul Signed	Total
(Emu) Kega Fusion	(KB/s)	2496.00	832.00	832.00	499.19	832.00	832.00	1248.00	1248.00	8819.19
(Emu) Gens	(KB/s)	2496.00	1248.00	832.00	499.19	226.90	208.00	499.19	624.00	6633.28
(Emu) BlastEM	(KB/s)	2496.00	1248.00	832.00	499.19	249.59	208.00	499.19	624.00	6655.97
(Emu) Exodus	(KB/s)	2496.00	832.00	832.00	499.19	208.00	178.28	499.19	624.00	6168.66
(HW) Mega Drive 1	(KB/s)	2496.00	1248.00	832.00	499.19	249.59	226.90	499.19	499.19	6550.06

Tbl. 01: Benchmark 1 (CPU) result tables (C / Assembly)

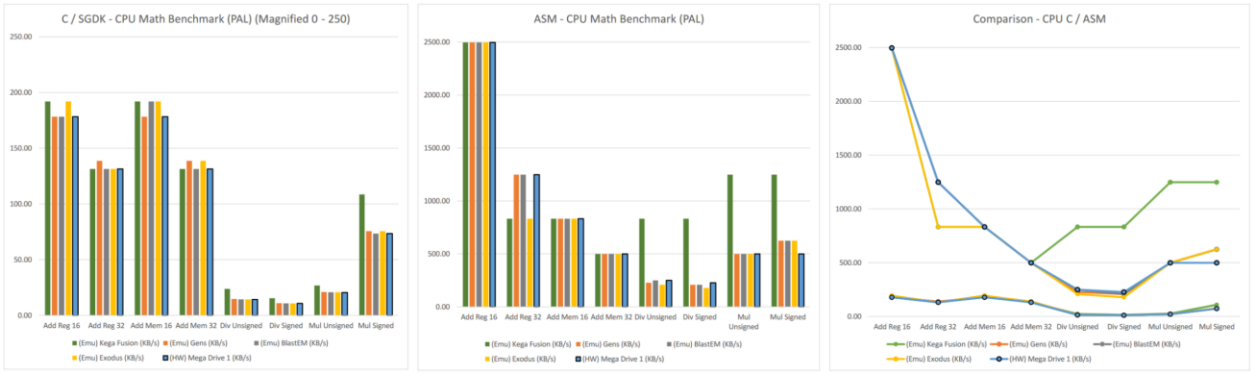


Fig. 24: Benchmark 1 (CPU) result diagrams (C / Assembly)

With one glance at the resulting data, the concerns expressed about the nature of volatile variable declaration seem to have manifested in the results. The values gathered from the mathematical operations in unoptimized C are surely not representative of the performance of calculations within the implementation of a normal game project in SGDK. The values produced by the custom assembly implementation seem to match the expectation as to what the hardware should be able to achieve, gathered from conversations with homebrew developers online. A different set of CPU heavy benchmarks could have been chosen to possibly reveal more telling data.

Even if the data itself is not quite as representative of the performance as hoped, the differences between the emulators and the real hardware and the relational scale of similar operations, still reveal a few interesting findings. Firstly, the difference in speed between 16-Bit register addition and 32-Bit register in the assembly implementation are are 2:1 or larger depending on execution platform. As quite literally half the data has to be worked on, this result is to be expected. With the C implementation however, both basic and memory allocated additions of 16-Bit and 32-Bit numbers result in close to the exact same result, leading to the conclusion that regardless of implementation, the volatile variables used for the operations get allocated into memory all the same, rather than only copying the values back to memory upon completion of the calculations. It is also notable that in the assembly implementation, the 16-Bit and 32-Bit memory allocated additions are less than half the speed and do not quite reach the 2:1 scale of their register based counterparts, as the performance floor for memory access erodes the inherent advantage of less processed data away.

Comparing the division operations between both the C / SGDK results and the 68000 assembly results, they are clearly different in the performance values reached, because of the described optimization issues, but the scale between the signed and unsigned division is almost identical between them, proving once more that division is the slowest operation to execute, even to the point that the signed or unsigned nature has little effect on the resulting performance.

Jumping to the results from the signed and unsigned multiplication benchmarks, a few interesting results have come up as well. The C / SGDK implementation of the signed multiply benchmark exceeds the unsigned multiplication by roughly 3.5:1, most likely being an artifact of the troubled optimization process. Interestingly, in the assembly implementation of the signed multiplication benchmarks most emulators achieve higher performance compared with the unsigned multiplication operation, whereas both operations achieve the exact same performance on the real SEGA Mega Drive console.

Lastly it is important to compare the results achieved by the different execution platforms. The real SEGA Mega Drive console should obviously be picked as the base line, as it is the original hardware all emulation software wants to come as close to as possible. The first noteworthy thing is that BlastEM seems to be the closest to the original console across the

board. It is not 100% accurate, but is definitely within the margin of error of the implementation tested. With Kega Fusion being perhaps the most popular emulator for the console it was quite shocking to see such wild inaccuracies, almost always performing much better than it should compared to the real hardware it is trying to emulate.

C / SGDK - Memory Benchmark (PAL)

Environment		MSET 10K * 256	MSET 10K * 1024	MSET 10K * 4096	MCPY 10K * 256	MCPY 10K * 1024	MCPY 10K * 4096	Total
(Emu) Kega Fusion	*(KB/s)*	1600.00	2000.00	2188.03	1142.85	1333.33	1376.34	9640.55
(Emu) Gens	*(KB/s)*	1600.00	2000.00	2169.49	1142.85	1306.12	1376.34	9594.80
(Emu) BlastEM	*(KB/s)*	1600.00	2000.00	2151.25	1142.85	1306.12	1361.70	9561.92
(Emu) Exodus	*(KB/s)*	1600.00	2000.00	2188.03	1142.85	1333.33	1383.78	9647.99
(HW) Mega Drive 1	*(KB/s)*	1454.54	2000.00	2115.70	1066.66	1306.12	1347.36	9290.38

ASM - Memory Benchmark (PAL)

Environment		MSET 10K * 256	MSET 10K * 1024	MSET 10K * 4096	MCPY 10K * 256	MCPY 10K * 1024	MCPY 10K * 4096	Total
(Emu) Kega Fusion	*(KB/s)*	1333.33	1333.33	1347.36	1333.33	1333.33	1347.36	8028.04
(Emu) Gens	*(KB/s)*	1333.33	1333.33	1340.31	1333.33	1333.33	1347.36	8020.99
(Emu) BlastEM	*(KB/s)*	1333.33	1333.33	1333.33	1333.33	1306.12	1333.33	7972.77
(Emu) Exodus	*(KB/s)*	1333.33	1333.33	1347.36	1333.33	1333.33	1354.49	8035.17
(HW) Mega Drive 1	*(KB/s)*	1333.33	1280.00	1299.49	1333.33	1306.12	1299.49	7851.76

Tbl. 02: Benchmark 2 (Memory) result tables (C / Assembly)

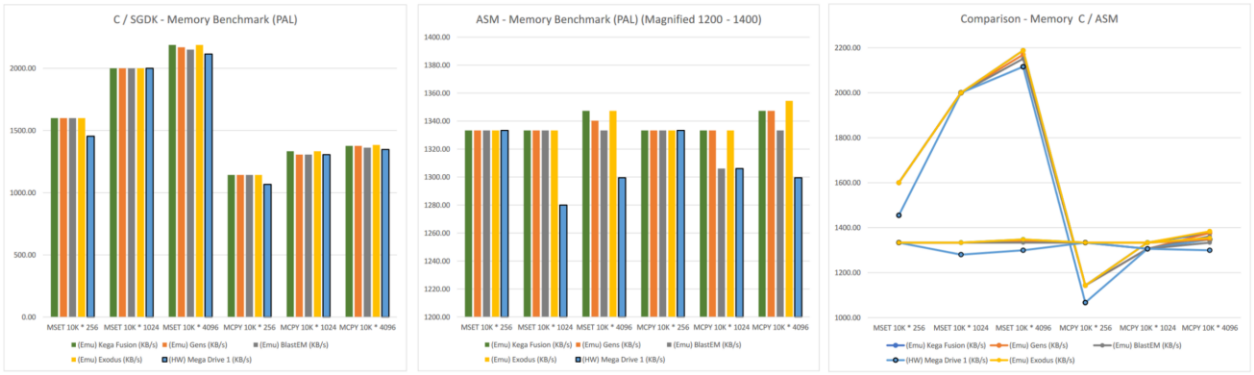


Fig. 25: Benchmark 2 (Memory) result diagrams (C / Assembly)

The data resulting from the memory based benchmarks turned out to be a lot more telling and thankfully a lot closer to the expected performance as the previous benchmark. Overall, the performance of both approaches is not too far off, with the memset and memcpy functions of the C / SGDK implementation winning out overall. The fact that C was faster in this case is to be expected, considering how important the memset and memcpy functions are to the overall C approach to programming, relying on pointers and buffers for memory to be managed explicitly. Supporting the assumption of the built in C functions being heavily optimized is the fact that both memset and memcpy benchmarks achieve higher data transfer speeds as

the amount of bytes to set in each buffer increases, pointing to optimizations for larger buffer spans.

Interestingly the real SEGA Mega Drive hardware falls short of all results from emulators in almost every single benchmark performed. This holds true for both C / SGDK and assembly. Why exactly this might be is not fully clear, yet it is fair to assume it is simply a lack of true hardware accuracy in certain edge cases like this highly memory dependent scenario. Noteworthy however is the wild differences between all execution platforms that emerges in the assembly implementation, as the amount of bytes to set per loop iteration increases. Looking at the generated comparison graph, it is clear that C / SGDK is beats out the custom assembly implementation for the memset benchmarks, but surprisingly falls short of the crown in the memcpy benchmarks. While the C / SGDK benchmark does take over the memcpy test with the highest amount of bytes to set, which helps its speed as established, the first (smallest) memcpy tests falls surprisingly short of the assembly implementation. One could make the point, that if many, smaller buffers are used in a project, which requires very consistent timings in their access, a version of the assembly implementation presented could be the better choice.

C / SGDK - Sprite Benchmark (PAL)

Environment		8*8 NO COL	16*16 NO COL	32*32 NO COL	8*8 COL	16*16 COL	32*32 COL	Total
(Emu) Kega Fusion	(FCPU/s)	4.13	4.14	4.13	0.51	0.47	0.38	13.76
(Emu) Gens	(FCPU/s)	3.79	3.79	3.79	0.45	0.39	0.32	12.53
(Emu) BlastEM	(FCPU/s)	3.78	3.79	3.78	0.42	0.37	0.27	12.41
(Emu) Exodus	(FCPU/s)	4.12	4.13	4.13	0.54	0.49	0.41	13.82
(HW) Mega Drive 1	(FCPU/s)	3.79	3.79	3.78	0.40	0.35	0.28	12.39

Tbl. 03: Benchmark 3 (Sprites) result table (C)

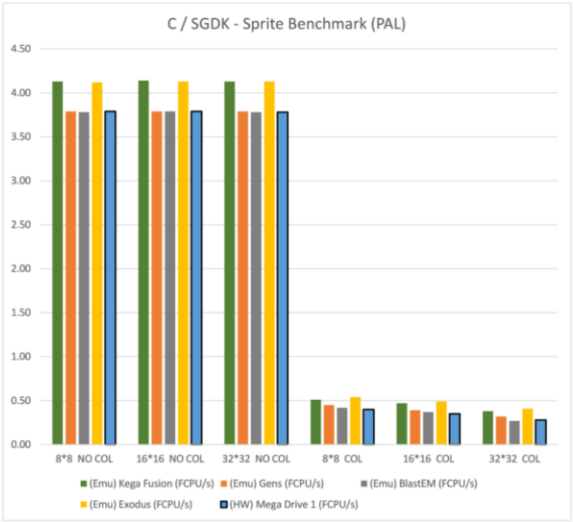


Fig. 26: Benchmark 3 (Sprites) result diagram (C)

As established, there is no meaningful assembly implementation to compare to the generated C / SGDK data with. While this might diminish the possible implications made clear from the benchmark, the data is still a unique look into the performance observable with the SGDK sprite engine. As clear to see the size of sprites makes virtually no difference in the three benchmark runs without collision enabled. To be fair, colliding the sprites with the screen boundaries, moving them and updating their animation frames is still very much being executed on the CPU, but as observable with the values being within a 0.01 margin of error, it is not enough to create a noticeable difference in the free CPU time spent inside the loop waiting for the vertical blank interrupt. The drastic decline in performance with collision enabled is due to the quality of collision code used, as the implementation is a very accurate modern one, doubling the required comparisons, compared to more typical simple collision checks for many games from the time. Notable are also the emulator differences in this case. While both the BlastEM and Gens emulators seem perfectly accurate with the real hardware without collision enabled, Gens starts drifint further away from the real hardware, while BlastEm

sticks very close to the real console, again reinforcing its place as the most accurate emulator tested with.

Finally, the results generated from the JSON trace profiling of the aforementioned MD-Profiler application can be observed. It is important to note that this form of visualizing the function execution can help finding certain bottlenecks or bugs, but it is less of a concrete performance measurement system and more of a simple visualization tool, especially for the kind of profiling that can be feasibly done on such limited hardware. Nonetheless, the fact that a modern performance profiling solution is indeed available for a system as ancient as the SEGA Mega Drive is truly baffling and underlines the true dedication this community has to improve the development experience for their favorite console.

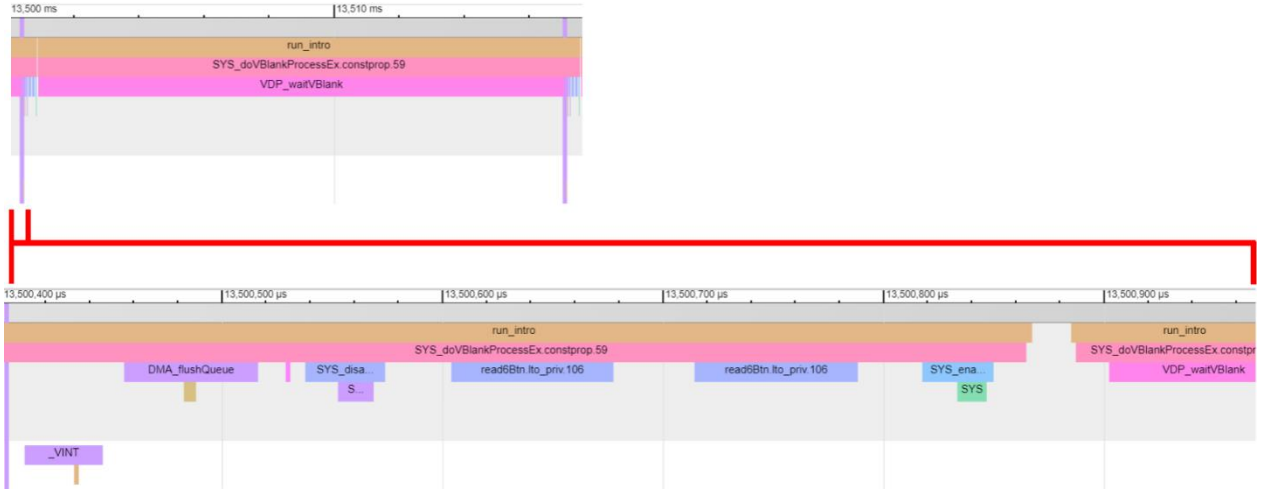


Fig. 27: Intro state profiling results

Observing the beginning of the intro state running function we can see the first vertical blank occurring. The purple line markers indicate the occurrence of the vertical interrupts. For every vertical interrupt executed by the system, the DMA (Direct Memory Access) queue is flushed, making room for new DMA requests to be collected over the coming frame time. Shortly after the DMA queue is flushed, the system interrupts are disabled for the reading of the joypads, through the call to read6Btn.Ito_priv(), before reenabling the system interrupts. These are the underlying function calls of the SYS_doVBlankProcess() function, showing that system interrupts are always disabled before accessing IO devices on the system bus, also by the built-in functions of SGDK itself.

Fig. 28: Benchmark 1 profiling results

The beginning of the run function for the first (CPU) benchmark state also reveals the function calls implemented as they appear in the source code. At the very beginning of the running function for this state the reset_reults() function is called, internally freeing and reallocating the memory the 8 result character buffers to be displayed during the results. The 8 green memory function calls are exactly those meomory allocation calls. The distance between them indicates the rest of the reset_results() loop does take a bit more time than expected. On the other hand, looking at the end of the first math add register benchmark, we can clearly see the function calls for drawing the (text) progress bar on screen (VDP_drawTest()), converting the previously calculated benchmark result to a string (fix32ToStr()) and the subsequent spitnf() call to format the string for getting passed to the results display.

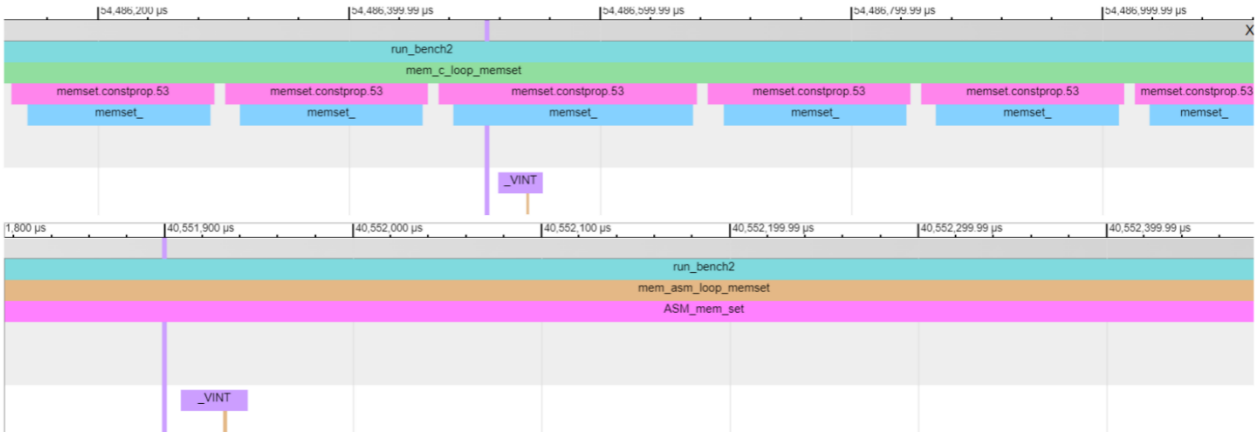


Fig. 29: Benchmark 2 profiling results (C / Assembly)

Comparing the visualization of the running function of the second benchmark in the C/SGDK (top) implementation and the assembly (bottom) implementation, we can clearly see that the individual function call to memset in C are recognized, but the assembly implementation shoes up as one contiguous strip, because the routine loops within itself. Implementing the assembly routine to use outside looping from a while loop in C would have illustrated possible differences here better. But it is useful to know that assembly routines are not as transparently profilable as the standard C implementation, as no other internal functions are called from assembly code (at least in this case).

Insights gained

All elements of the experience working on this subject have been incredibly interesting and fascinating to try and grasp, define and implement. From the beginning of concerning myself with a base understanding of the underlying SEGA Mega Drive hardware, to the first experiments using SGDK and intently teaching myself the basic C language with all oft its lower-level features, all the way to the deep dives into the depths of 68000 assembly development, the journey was worth it. The desired project goal may not have been implemented to the original specification and the results achieved may not have been the most representative of the real-world performance to be expected, but the valuable knowledge absorbed, tested, used and communicated through this research paper and the related thesis are more than enough

for more interested developers to learn from and start their very own journey into homebrew development.

Among the most important insights gained during this procedure has been the usage, limitation and comparison of the C approach and the low-level assembly code achievable by a representative game developer, not by an individual specifically trained in this form of development. The fact that it is very much possible to learn such a restrictive way to program and still implement representative features up to a certain degree speaks to this skill being a lot more learnable than many younger people interested in the subject might think. The possibility of an assembly programming course as part of a more computer-science oriented curriculum does not seem as unrealistic as once thought.

Most relevant for the active game developers and retro gaming enthusiasts who may read this paper is the overall robust and easily learned nature of the SGDK framework. With a few guides and minimal effort, less programming focused game designers and pixel artists can absolutely get to grips with this high-level framework and begin contributing to the SEGA Mega Drive homebrew community by doing what they do best, possibly inspiring programmers to form new teams with them and creating a new stream of 16-Bit games for years to come.

Possibly the most valuable insight gained for myself is to be more persistent at daunting tasks, shining more and more light onto difficult topics instead of shying away or going an easier route. Even if the ultimate goal is not fully reached, deciphering more of the difficult and daunting problems however much is possible will already clear up more questions than avoiding them, not to mention having the ability to propel other likeminded people into a position to uncover what I could not, by giving them a head-start through my research.

OUTLOOK AND FUTURE WORK

Relevance for future projects

The research and work conducted during this semester spent on the pursuit of improving the limited academic accessibility to development models, experiences and technical knowledge relating to the SEGA Mega Drive platform has personally been incredibly rewarding. The implementations, workflows and chosen software used during both the conducted research and the related main thesis were combined and used to shine new light on important questions to be asked about the subject in an academic context. For this the current state-of-the-art was largely covered and utilized to my best present ability, hoping to potentially contribute something to the advancement of this subject through the presented work.

While the results of the performance benchmark implementation of this research paper might not have been as empirically indicative of the performance to be expected of larger projects, as hoped, the research conducted, knowledge acquired and technical foundation built as part of the process are solid foundations for future students and interested developers to build off of. All interested parties should be able to make helpful use of these findings for their possible future endeavors into homebrew development. After all, demystifying this more obscure part of the game development landscape can only bring about more creative use of classic hardware, something that the homebrew scene and gaming at large can only benefit from.

As for myself, the knowledge acquired during this process has been massively interesting and many ideas for games and graphical demos possible on the SEGA Mega Drive have run through my mind. I am sure that, given enough available time in the coming year, I will see to produce at least one personal project using either the SGDK framework or my personal assembly language template for the platform. On that note, I wish to thank my professor Dr. Leissler for this unique opportunity to work on this subject. The SEGA Mega Drive was the first console I ever owned and has a very special place in my heart. Without these memories I would have never chosen to pursue game development and design professionally, so being able to quite literally come full circle with this incredibly rewarding project is more than humbling.

Lessons learned

At multiple points during the conducted work, things did not go according to my initial plan. Learning from these missteps was vital to quickly improve and keep moving towards the intended goal. Foremost the choice of benchmarks implemented as part of this research paper should have had more ground in the reality of contextual development on the SEGA Mega Drive system. The abstract nature of the CPU and Memory benchmarks in particular were not very indicative of what new developers could expect and the problems with compiler optimizations skewed the resulting data even further. The assembly and SGDK implementations still provided a worthy learning experience for myself and possible readers, yet this problem could have been avoided with a better choice of benchmarks.

Learning a type of assembly language for the first time equally confronted me with many questions and initially a lot of frustration. Once somewhat sufficiently studied however it also became a unique experience as the mentality of organizing and structuring source code proved to be quite different in this more limited environment. Needless to say, the restrictive nature proved difficult, but the positives of speed and true low-level understanding provided a rewarding learning experience.

Of general note was the truly inspiring willingness of the small homebrew communities on various platforms to engage with my questions and help me with my understanding of a variety of topics, down to even offering full support sessions for me to learn as much as possible about their passion. Being typically more cautious when asking for help from essentially online strangers, this really gave me a boost of confidence to keep working on the project and has changed my typical hesitance when jumping into such foreign topics as the one covered in this paper.

REFERENCES

GamingHistorian101. (2012) "Homebrew"
https://gaminghistory101.com/2012/01/17/homebrew/

TigerNT. (2009) "68000 Instruction Set"
http://www.tigernt.com/onlineDoc/68000.pdf

SegaRetro. (2011) "PSY-Q Development System (Mega Drive)"
https://segaretro.org/PSY-Q_Development_System_(Mega_Drive)#

Charles Kelly. (2014) "EASy68K Home Page"
http://www.easy68k.com/moreinfo.htm

Dr. Volker Barthelmann. (2021) "vasm. A portable and retargetable assembler."
http://sun.hasenbraten.de/vasm/

GNU Project. (2021) "GCC, the GNU Compiler Collection"
https://gcc.gnu.org/

Don Ho. (2022) "https://notepad-plus-plus.org/author/"
https://notepad-plus-plus.org/downloads/

Microsoft Corp. (2022) "Code editing. Redefined."
https://code.visualstudio.com/

Second Dimension. (2022) "Second Dimension Home Page"
https://www.sbasic.net/

Stephane Dallongeville. (2022) "SGDK - A free and open development kit for the Sega Mega Drive "

https://github.com/Stephane-D

http://icy.bioimageanalysis.org/

Kubilus1. (2015) "GENDEV - Genesis development environment for Linux."

https://github.com/kubilus1/gendev

Andre DeRosier. (2017) "Marsdev. Cross platform Mega Drive / 32X toolchain and Makefile abuse."

https://github.com/andwn/marsdev

Sikthehedgehog. (2020) "MDTOOLS"

https://github.com/sikthehedgehog/mdtools

Matt Phillips. (2019) “Tanglewood"

https://github.com/BigEvilCorporation/Beehive

https://github.com/BigEvilCorporation/TANGLEWOOD

https://uk.linkedin.com/in/mattphillips1

Sébastien Bénard. (2018) "An open-source 2D level editor from the creator of Dead Cells, with a strong focus on user-friendliness"

https://ldtk.io/

https://deepnight.net/about/

Thorbjørn Lindeijer. (2012) "Full-featured Level Editor"

https://www.mapeditor.org/

https://github.com/mapeditor/tiled

Igara Studio. (2022) "Animated Sprite Editor & Pixel Art Tool"

https://www.aseprite.org/

https://www.igarastudio.com/about/

GrafX2. (2001) "GrafX2 is a bitmap paint program inspired by the Amiga programs Deluxe Paint and Brilliance."

http://grafx2.chez.com/

Leonardo Demartino. (2011) "DefleMask. The best chiptune tracker"

https://www.deflemask.com/

https://demartino.ar/

Kaneda. (2006) "An advanced mod for Gens emulator for hacker, ripper and...coder"

https://gendev.spritesmind.net/page-tools.html

Steve Palmer. (2010)

https://segaretro.org/Steve_Snake

https://segaretro.org/Kega_Fusion

Michael Pavone. (2019) "BlastEm - The fast and accurate Genesis emulator"

https://www.retrodev.com/blastem/

https://twitter.com/mikepavone

Tails8521. (2021) "md-profiler, a tracing profiler for the Sega MegaDrive/Genesis"

https://github.com/Tails8521/md-profiler

Keith 'Akoyou'. (2020) "Learn Multiplatform Assembly Programming with ChibiAkumas! "

https://www.chibiakumas.com/

https://www.amazon.com/Learn-Multiplatform-Assembly-Programming-ChibiA-kumas/dp/B08W7DWZB3

Makrey Jester. (2018) "MarkeyJester’s Motorola 68000 Beginner’s Tutorial"
https://mrjester.hapisan.com/04_MC68/

Hugues Johnson. (2022) "Sega Genesis Programming"
https://huguesjohnson.com/
https://huguesjohnson.com/programming/genesis/palettes/

BA ANIMATION & GAME | Declaration of Authorship

I hereby declare that the thesis I am submitting is entirely my own work except where otherwise indicated.

I have clearly indicated the presence of all material from other sources I have quoted, paraphrased or used in any way. This includes texts, presentations, graphics, diagrams, still or moving images programme code and sounds.

I have clearly indicated the presence of all material that I have created for other examinations or published elsewhere. I understand the rules and regulations regarding academic good practice and plagiarism.

Dornburg, 24.01.22 Hölzel, Luca

Place | Date | Last Name, First Name | Signature

BA ANIMATION & GAME | Archiving Declaration

Mark with a cross where applicable:

- [x] I hereby give my consent to the storage of a printed copy of my thesis in the Library of Darmstadt University of Applied Sciences

- [ ] I do not give my consent to the storage of a printed copy of my thesis in the Library of Darmstadt University of Applied Sciences.

  Reason:

  - [ ] My thesis is confidential, as it was written in cooperation with a company and is subject to the restrictions of a non disclosure agreement.

  - [ ] Personal reasons

Dornburg, 24.01.22 Hölzel, Luca

Place | Date | Last Name, First Name | Signature

BA ANIMATION & GAME | Eigenhändigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig erstellt und keine anderen als die angegebenen Hilfsmittel und Quellen verwendet habe.

Soweit ich auf fremde Materialien (Texte, Grafiken, Diagramme, Bildmaterial, Codes, Sound) oder Gedankengänge zurückgegriffen habe, enthalten meine Ausführungen vollständige und eindeutige Verweise auf die Urheber und Quellen. Dies gilt auch für Quellen, die ich selbst für andere Veröffentlichungen oder Prüfungen erstellt habe.

Alle weiteren Inhalte der vorgelegten Arbeit stammen von mir im urheberrechtlichen Sinn soweit keine Verweise und Zitate erfolgen.

Mir ist bekannt, dass ein Täuschungsversuch vorliegt, wenn die vorstehende Erklärung sich als unrichtig erweist.

Dornburg, 24.01.22 Hölzel, Luca

Ort | Datum | Name, Vorname | Unterschrift

BA ANIMATION & GAME | Erklärung zur Archivierung

Bitte Zutreffendes ankreuzen:

- [x] Mit der Archivierung der gedruckten Abschlussarbeit in der Bibliothek der Hochschule Darmstadt bin ich einverstanden.
- [ ] Mit der Archivierung der gedruckten Abschlussarbeit in der Bibliothek der Hochschule Darmstadt bin ich nicht einverstanden.
  Begründung:
  - [ ] Die Arbeit ist unterliegt der Geheimhaltung, da sie in einem Unternehmen durchgeführt wurde und ihr Inhalt ausdrücklich durch dieses gesperrt ist.
  - [ ] Persönliche Gründe

Dornburg, 24.01.22 Hölzel, Luca

Ort | Datum | Name, Vorname | Unterschrift

BA ANIMATION & GAME | Declaration of Authorship

I hereby declare that the thesis I am submitting is entirely my own work except where otherwise indicated.

I have clearly indicated the presence of all material from other sources I have quoted, paraphrased or used in any way. This includes texts, presentations, graphics, diagrams, still or moving images programme code and sounds.

I have clearly indicated the presence of all material that I have created for other examinations or published elsewhere. I understand the rules and regulations regarding academic good practice and plagiarism.

Dornburg, 14.02.22 Hölzel, Lucu

Place | Date | Last Name, First Name | Signature

BA ANIMATION & GAME | Archiving Declaration

Mark with a cross where applicable:

- [x] I hereby give my consent to the storage of a printed copy of my thesis in the Library of Darmstadt University of Applied Sciences

- [ ] I do not give my consent to the storage of a printed copy of my thesis in the Library of Darmstadt University of Applied Sciences.

  Reason:

  - [ ] My thesis is confidential, as it was written in cooperation with a company and is subject to the restrictions of a non disclosure agreement.

  - [ ] Personal reasons

Dornburg, 14.02.22 Hölzel, Lucu

Place | Date | Last Name, First Name | Signature

BA ANIMATION & GAME | Eigenhändigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig erstellt und keine anderen als die angegebenen Hilfsmittel und Quellen verwendet habe.

Soweit ich auf fremde Materialien (Texte, Grafiken, Diagramme, Bildmaterial, Codes, Sound) oder Gedankengänge zurückgegriffen habe, enthalten meine Ausführungen vollständige und eindeutige Verweise auf die Urheber und Quellen. Dies gilt auch für Quellen, die ich selbst für andere Veröffentlichungen oder Prüfungen erstellt habe.

Alle weiteren Inhalte der vorgelegten Arbeit stammen von mir im urheberrechtlichen Sinn soweit keine Verweise und Zitate erfolgen.

Mir ist bekannt, dass ein Täuschungsversuch vorliegt, wenn die vorstehende Erklärung sich als unrichtig erweist.

Dornburg, 14.02.22 Hölzel, Lucas

Ort | Datum | Name, Vorname | Unterschrift

BA ANIMATION & GAME | Erklärung zur Archivierung

Bitte Zutreffendes ankreuzen:

- [x] Mit der Archivierung der gedruckten Abschlussarbeit in der Bibliothek der Hochschule Darmstadt bin ich einverstanden.
- [ ] Mit der Archivierung der gedruckten Abschlussarbeit in der Bibliothek der Hochschule Darmstadt bin ich nicht einverstanden.
  Begründung:
  - [ ] Die Arbeit ist unterliegt der Geheimhaltung, da sie in einem Unternehmen durchgeführt wurde und ihr Inhalt ausdrücklich durch dieses gesperrt ist.
  - [ ] Persönliche Gründe

Dornburg, 14.02.22 Hölzel, Lucas

Ort | Datum | Name, Vorname | Unterschrift

Prof. Katharina Kafka
Prüfungsausschuss-Vorsitzende

Fachbereich Media
Studiengang Animation and Game

Besucheranschrift:
Max-Planck-Str. 2 D-64807 Dieburg
Fax +49.6151.16-9413
Katharina.Kafka@h-da.de
www.h-da.de

Hochschule Darmstadt FB Media
Haardtring 100 D-64295 Darmstadt

Herr
Luca Hölzel
Leimenkaut 16

65599 Dornburg

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Az. KK/pip-md
Darmstadt, 30. September 2021

Antrag auf Zulassung zur Bachelorarbeit

Sehr geehrter Herr Hölzel,

der Prüfungsausschuss des Studiengangs **Animation and Game** hat Ihren Antrag geprüft und lässt Sie zur Bachelorarbeit zu.

Thema der Bachelorarbeit:

Development of an Modern Toochain for 16-Bit Homebrew Games on Sega Genesis

Referent/in der Bachelorarbeit: **Leissler (Fb. Media)**

Korreferent/in der Bachelorarbeit **Nickel (Fb. Media)**

Abgabe: **14.02.2022 bis 12 Uhr im Prüfungssekretariat**

Der **praktische Teil** der Bachelorarbeit ist **dreifach** in elektronischer Form auf Datenträger, der **schriftliche Teil (Dokumentation)** ist **dreifach** in **gebundener** und gedruckter Form **sowie einfach in elektronischer Form** einzureichen.

In die gebundene Dokumentation sind **umseitige Erklärungen fest einzubinden**! Die Vorlage finden Sie im Intranet.

Die Dokumentation (also der schriftliche Teil der Bachelorarbeit) muss in deutscher oder englischer Sprache angefertigt werden. Bei Abgabe auf Deutsch muss eine Zusammenfassung in englischer Sprache beigefügt werden, bei Abgabe auf Englisch eine Zusammenfassung in deutscher Sprache.

Professor Katharina Kafka
Prüfungsausschuss-Vorsitzende

→

Diese beiden Erklärungen müssen inhaltlich in die Abschlussarbeit (fest eingebunden) übernommen werden:

Erklärung:

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig erstellt und keine anderen als die angegebenen Hilfsmittel und Quellen verwendet habe.
Soweit ich auf fremde Materialien, Texte oder Gedankengänge zurückgegriffen habe, enthalten meine Ausführungen vollständige und eindeutige Verweise auf die Urheber und Quellen.
Alle weiteren Inhalte der vorgelegten Arbeit stammen von mir im urheberrechtlichen Sinn, soweit keine Verweise und Zitate erfolgen.
Mir ist bekannt, dass ein Täuschungsversuch vorliegt, wenn die vorstehende Erklärung sich als unrichtig erweist.

Domburg, 14.02.22
Ort, Datum


Unterschrift

Erklärung zur Archivierung:

Bitte zutreffendes ankreuzen:

- [x] Mit der Archivierung der gedruckten Abschlussarbeit in der Bibliothek **bin ich einverstanden.**

- [ ] Mit der Archivierung der gedruckten Abschlussarbeit in der Bibliothek **bin ich nicht einverstanden**.

  Begründung:

  - Die Arbeit ist gesperrt, da sie in einem Betrieb durchgeführt wurde und ihr Inhalt ausdrücklich durch diesen gesperrt ist. (Vgl. ABPO § 18 (9))

  - Persönliche Gründe

Domburg, 14.02.22
Ort, Datum


Unterschrift