UNIVERSITY OF UDINE

Bachelor's Degree in Multimedia Science and Technology

Retrocomputing on PlayStation 1: design and development of two Tech Demos for the console's 30th anniversary

Supervisor: Demis Ballis

Student: Gabriele Passuello

Student ID: 157724

Academic Year 2023 / 2024

Contents

Abstract in Italian 1

Abstract 2

I Origins and Hardware of PlayStation 4

1 Thesis Objective 5

2 The Birth of PlayStation 7

2.1 Ken Kutaragi . . . 7
2.2 The Early Collaborations: Sony and Nintendo . . . 8
2.3 PS-X . . . 8
2.4 The Birth of Sony Computer Entertainment . . . 10
2.5 Market Assertion . . . 11

3 The Hardware of PlayStation 13

3.1 The Models . . . 13
3.1.1 The SCPH-1000 Model . . . 14
3.1.2 The SCPH-3000 Model . . . 16
3.1.3 The SCPH-5000 Model . . . 17
3.1.4 The SCPH-7000 Model . . . 18
3.1.5 The SCPH-9000 Model . . . 19
3.1.6 The SCPH-100 Model . . . 19
3.1.7 The DTL-H Models . . . 20
3.2 Hardware Overview . . . 21
3.2.1 System Architecture . . . 22
3.2.2 CPU Introduction . . . 23
3.2.3 Graphics System Introduction . . . 23
3.2.4 Audio System Introduction . . . 24
3.2.5 Additional Supports . . . 25
3.2.6 Motherboard Analysis . . . 26
3.3 The PSX CPU . . . 28

3.3.1 Processor Characteristics . . . 28
3.3.2 Processor Origins . . . 29
3.3.3 Coprocessors Analysis . . . 31
3.3.4 Memory with Addressable Access . . . 32
3.4 The PSX GPU . . . 35
3.4.1 VRAM . . . 36
3.4.2 Video Outputs . . . 36
3.5 The PSX SPU . . . 37
3.6 Management of PSX I/O Interfaces . . . 38
3.6.1 CD Module . . . 38
3.6.2 Front Ports . . . 39
3.6.3 Rear Ports . . . 39

II Development and Programming Fundamentals on PS1 40

4 Programming in MIPS Assembly 41
4.1 The 32 General-Purpose Registers . . . 41
4.2 Elementary MIPS Instructions . . . 43
4.2.1 Data Transfer Instructions . . . 43
4.2.2 Load Instructions . . . 44
4.2.3 Store Instructions . . . 45
4.2.4 Differences between Load and Store . . . 45
4.2.5 Jump Instructions . . . 45
4.2.6 Branch Instructions . . . 46
4.2.7 The NOP Concept . . . 46
4.2.8 Arithmetical Instructions . . . 47
4.3 Console Emulation . . . 48
4.3.1 Programming Example . . . 48
4.3.2 The PS-EXE Format . . . 49
4.3.3 The Pseudo-Instruction Concept . . . 51
4.3.4 The Sub-Routine Concept . . . 51
4.4 Handling Binary Data . . . 52
4.4.1 Managing Negative Numbers . . . 52
4.4.2 Sign Extension . . . 53
4.4.3 Logical Operations . . . 54
4.4.4 The Bit-Shifting Concept . . . 55
4.5 In-depth Study of the MIPS Pipeline . . . 57
4.5.1 MIPS Pipeline Structure . . . 57

4.5.2 Limits of the MIPS Pipeline . . . 58
4.5.3 Managing Delay Slots . . . 58
4.5.4 Optimizing Delay Slots . . . 58
4.6 In-depth Study of the RISC Processor . . . 58
4.6.1 Historical Evolution and Context . . . 58
4.6.2 Differences between RISC and CISC . . . 59
4.6.3 Advantages of RISC Architecture . . . 60
4.7 Graphics System . . . 60
4.7.1 The Frame Buffer . . . 61
4.7.2 Display Configuration Parameters . . . 62
4.7.3 Color and Depth . . . 63
4.7.4 PSX Primitives . . . 66
4.7.5 Packet Management and Communication between CPU and GPU . . . 67
4.7.6 First Example of Basic Rendering in MIPS Assembly . . . 68
4.8 Memory Management . . . 73
4.8.1 The MIPS Application Binary Interface . . . 73
4.8.2 The Concepts of Heap and Stack . . . 74
4.8.3 Second Example of Basic Rendering in MIPS Assembly . . . 75
4.8.4 The Concept of Variable and Vector Alignment . . . 77
4.8.5 Third Example of Basic Rendering in MIPS Assembly . . . 78

5 Programming in C **84**
5.1 History of the PSY-Q SDK . . . 85
5.2 Key Programming Concepts . . . 86
5.2.1 Double Buffering . . . 86
5.2.2 Z-Sorting . . . 86
5.2.3 Ordering Tables . . . 86
5.3 Main System Libraries . . . 89
5.4 Geometry Transformation Engine . . . 90
5.4.1 3D Transformations . . . 90
5.4.2 Examples of GTE Instructions . . . 91
5.4.3 GTE Register Set . . . 94
5.5 Clipping . . . 96
5.5.1 Backface Culling . . . 96
5.5.2 Cohen-Sutherland Algorithm . . . 98
5.5.3 Liang-Barsky Algorithm . . . 99
5.5.4 Bounding Boxes . . . 99
5.6 Fixed Point Math . . . 101

5.7 In-depth Analysis of the PSX BIOS . . . 104
5.7.1 Reading Joypad Inputs via BIOS . . . 106
5.8 Reading the CD-ROM . . . 107
5.8.1 CD-ROM . . . 107
5.8.2 The Unique Design of PSX CDs . . . 110
5.8.3 Types of PlayStation Files . . . 110
5.8.4 Function to Read Binary Data from the Disc . . . 112
5.8.5 Anti-Piracy Mechanisms . . . 113
5.9 Textures . . . 115
5.9.1 Foundations of Texture Mapping . . . 115
5.9.2 Concept of T-PAGE . . . 118
5.9.3 Concept of CLUT . . . 119
5.9.4 Insight on the TIM format . . . 119
5.9.5 PSX Graphic Artifacts . . . 120
5.10 Audio . . . 124
5.10.1 Types of ADPCM formats . . . 124
5.10.2 Details on VAG and XA formats . . . 125
5.10.3 Management of Audio Tracks on CD-ROMs . . . 127
5.10.4 Example of audio implementation . . . 129

III Creating Demos on PlayStation: A Technical and Creative Showcase 130

6 Demo Disc One (Showcase of Various Demos) 131
6.1 Cube Transformations . . . 131
6.2 Bouncing Cubes . . . 132
6.3 Multiplayer . . . 133
6.4 Texture Mapping . . . 134
6.5 Fog . . . 134
6.6 Phong . . . 135
6.7 Movie / M-DEC . . . 137
6.8 3D Animation . . . 139

7 Demo Disc Two (WipEout) 141
7.1 Reverse Engineering WipEout . . . 141
7.2 Structure of a PRM File (Object) . . . 143
7.3 Ship Rendering Part 1 (Primitives and Wireframe) . . . 145
7.4 Structure of a CMP File (Texture) . . . 146

7.5 Ship Rendering Part 2 (Textures) 147
7.6 Structure of TRV, TRF, and TRS Files (Track Vertices, Faces, Sections) 149
7.7 Track Rendering 151
7.8 Implementation of Physics, Gameplay, and Sound Effects 154

IV Resources and Final Considerations 160

8 Conclusions 161

Glossario 162

Visual Documentation 164

General Bibliography 169

Abstract in Italian

Nel 1994, l'azienda giapponese Sony rivoluzionò il settore videoludico con il lancio di PlayStation, una console che ridefinì gli standard tecnologici e creativi dell'epoca grazie ad un' architettura innovativa, nettamente superiore rispetto alle console della generazione precedente. A trent'anni dal suo debutto, questa tesi si propone di celebrare l'importanza tecnologica di PS1, analizzandone l'impatto storico e ricreando, in un contesto moderno, le condizioni operative degli sviluppatori dell'epoca attraverso un approccio di retrocomputing.

Con retrocomputing si intende lo studio, la ricostruzione o l'utilizzo di sistemi informatici del passato, con l'obiettivo di ricreare esperienze originali in un contesto moderno o di sperimentare con tecnologie obsolete.

L'obiettivo principale di questa analisi vuole essere la valorizzazione delle potenzialità tecniche della console attraverso lo sviluppo di due Tech Demo ispirate ai software promozionali Sony, utilizzati all'epoca per mettere in evidenza le capacità grafiche, computazionali e sonore del sistema. Questi dischi dimostrativi hanno l'obiettivo di esplorare funzionalità come il rendering di oggetti in tre dimensioni, l'ottimizzazione delle risorse hardware e le principali novità tecniche della console.

La tesi affronta inoltre lo sviluppo di software per PS1, approfondendo la programmazione in due linguaggi di programmazione, "MIPS Assembly" (linguaggio di basso livello) e "C" (linguaggio di alto livello). Attraverso questi due linguaggi vengono analizzati concetti chiave di computer grafica come il texture mapping, il rendering 3D e gli algoritmi di clipping.

A fare da supporto alla parte pratica viene dedicata un'analisi dettagliata alla storia e all'hardware della console, per comprendere al meglio il funzionamento interno della console e le sue innovazioni tecnologiche. Lo studio dei componenti hardware come CPU, GPU, Memoria e moduli I/O permette di comprendere appieno le nuove soluzioni adottate.

Questo lavoro non solo vuole ricostruire fedelmente il processo di sviluppo degli anni '90 ma anche offrire una retrospettiva unica sull'importanza di PS1 nel panorama videoludico, evidenziandone le sfide, i limiti tecnologici e le innovazioni che hanno definito un'epoca di industria del gaming.

Abstract

In 1994, the Japanese company Sony revolutionized the video game industry with the launch of PlayStation, a console that redefined the technological and creative standards of the era thanks to an innovative architecture, significantly superior to the previous generation of consoles. Thirty years after its debut, this dissertation aims to celebrate the technological significance of the PS1 by analyzing its historical impact and recreating, in a modern context, the operational conditions of the developers of that time through a retrocomputing approach.

Retrocomputing refers to the study, reconstruction, or use of past computer systems with the aim of recreating original experiences in a modern context or experimenting with obsolete technologies.

The main goal of this analysis is to highlight the technical potential of the console by developing two Tech Demos inspired by Sony's promotional software, which were used at the time to showcase the system's graphical, computational, and audio capabilities. These demonstration discs aim to explore features such as 3D object rendering, hardware resource optimization, and the console's key technical innovations.

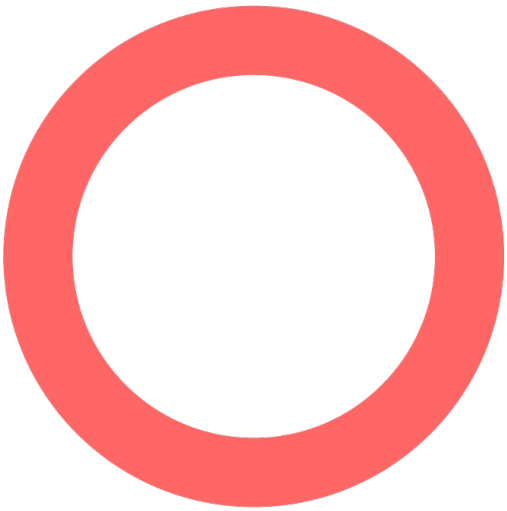
The dissertation also delves into the development of PS1 software, focusing on programming in two languages: "MIPS Assembly" (a low-level language) and "C" (a high-level language). Through these languages, key computer graphics concepts such as texture mapping, 3D rendering, and clipping algorithms are analyzed.

To support the practical work, a detailed analysis of the history and hardware of the console is provided, offering insights into the internal workings of the system and its technological innovations. This analysis of hardware components such as the CPU, GPU, memory, and I/O modules allows for a comprehensive understanding of its technical advancements.

This work not only aims to faithfully reconstruct the development process of the 1990s but also offers a unique retrospective on the importance of the PS1 in the gaming landscape, highlighting the challenges, technological limitations, and innovations that defined an era in the video game industry.

Part I

Origins and Hardware of PlayStation



Chapter 1

Thesis Objective

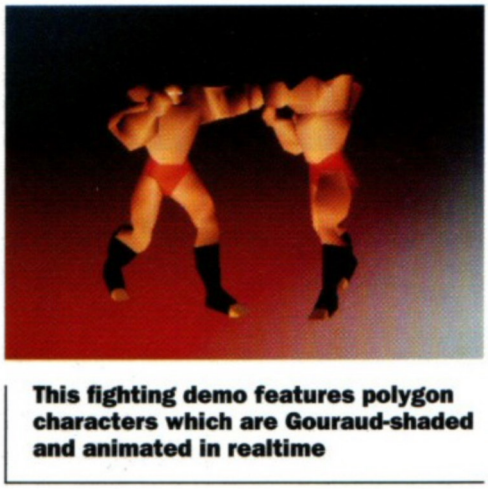
In 1994, SONY, a leading Japanese technology company, entered the video game market with the launch of its first console, the PlayStation—an event that marked a turning point for the entire industry. This product succeeded in redefining the technological and creative standards of gaming, thanks to a hardware and software architecture that was highly innovative for its time. Also known as the PS1 or PSX, the console represented a significant generational leap compared to previous platforms such as the Super Nintendo Entertainment System and the Sega Genesis.

Thirty years after its debut, the PS1 anniversary provides a unique opportunity to reflect on the technological and cultural impact of this console, analyzing the advances it introduced and the technical challenges that accompanied its development. This celebration goes beyond mere nostalgia, aiming instead to highlight the console's role as a milestone in the evolution of the video game medium.

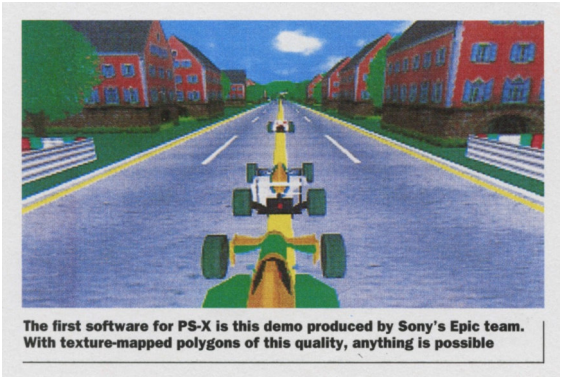
This thesis aims to analyze and showcase the technical potential of the console by recreating, in a modern context, the working conditions faced by developers of the time.

To this end, a series of "Demo Discs" will be developed, inspired by the demonstration software used by SONY to showcase the console's capabilities during its promotional phase (targeted at the press, journalists, and the public) prior to its official release. These programs were intended to highlight the system's graphical, audio, and computational capabilities, demonstrating what the console was able to achieve using the technology available in the 1990s.

The work was carried out by simulating the use of original or equivalent period tools, such as a 32-bit development platform and SONY's official development software. This approach will faithfully reproduce the historical software development process for the PlayStation, offering a close-up view of the techniques and constraints programmers faced at the time. In particular, the focus will be on creating two "Tech Demos" capable of leveraging the console's strengths—3D graphics rendering, high-polygon models, advanced audio handling, and optimized use of



This fighting demo features polygon characters which are Gouraud-shaded and animated in realtime



The first software for PS-X is this demo produced by Sony's Epic team. With texture-mapped polygons of this quality, anything is possible

The PS-X's polygon-handling capabilities were aptly demonstrated by this T-Rex head. Using a standard joypad, a Sony representative was able to open and close the dinosaur's jaws – all smoothly and in real-time

Labyrinth (aka *Legend* in Edge 9), features a dungeon modelled entirely in realtime (it runs in one frame!) and with unlimited viewpoints controlled by the joypad

Figure 1.1: Excerpts from the British magazine *Edge* (issues 05, 06, 09, 11) [1]

hardware resources.

The two discs, which represent the tangible outcome of this process and a concrete link to the past, will be burned onto CD-ROMs, allowing them to be tested on an original console and thus providing an additional level of practical and historical experience.

Through this research, the aim is to recreate an experience that explores the PlayStation's technological capabilities while simultaneously restoring the development conditions present at the time of its launch. At the same time, the goal is to achieve as faithful a reproduction as possible of the challenges and innovations that characterized the console's early years. In doing so, it will be possible to gain a more concrete understanding of the PSX's importance as a turning point in the evolution of gaming, connecting contemporary development work with that undertaken by programmers thirty years ago.

Chapter 2

The Birth of PlayStation

2.1 Ken Kutaragi

The birth of PlayStation is a story of vision, determination, and innovation. Behind this success lies a key figure: Ken Kutaragi, the visionary engineer who has been nicknamed the ”father of PlayStation.” His passion for technology and his commitment to tackling seemingly impossible challenges changed the course of video game history.

Figure 2.1: Ken Kutaragi
[2]

In 1984, during a presentation at the Sony Information Processing Research Center, Kutaragi was struck by the incredible real-time 3D graphical processing power offered by Sony’s ”System G,” a workstation designed for television broadcasting. This event sparked an idea in him that would later become revolutionary: bringing real-time 3D graphical power to video games, an objective that would lead to the PlayStation project. Although Sony was not initially a company tied to this sector, Kutaragi began to dream big, envisioning a console capable of competing with and even dominating the market.

2.2 The Early Collaborations: Sony and Nintendo

In the early 1990s, Kutaragi managed to secure the opportunity to work closely with Nintendo. After noticing the poor sound quality of the Famicom (NES), Kutaragi convinced Sony to develop a powerful audio chip for the Super Nintendo (SNES) console. This marked the beginning of a collaboration that, although brief, allowed him to gain experience in the video game industry and advance his vision.

At the same time, the video game market was undergoing a major transformation. The industry was adapting to the growing importance of CD-ROM technology, which promised more advanced interactive experiences and a cheaper solution compared to expensive cartridges. While other major companies in the sector, such as Sega, Philips, and Commodore, were already entering the field, Sony also began to consider the idea of developing its own CD-ROM-based gaming system.

Figure 2.2: A CD-ROM in the tray of a partially open CD-ROM drive.
[3]

In 1991, in collaboration with Nintendo, Sony designed a CD-ROM unit for the SNES, which would have combined a CD drive with the aforementioned console. The initial plan called for a collaboration in which Sony would create and produce the CD media, while Nintendo would maintain control over the production of cartridges. However, the relationship between the two companies quickly deteriorated. Nintendo, concerned about losing control over the potential CD-ROM market, decided to terminate negotiations with Sony and secretly struck a deal with Philips to develop a separate CD-I Add-On.

2.3 PS-X

During the 1991 Consumer Electronics Show, Sony proudly announced the new partnership with Nintendo, showcasing a prototype of the ”SNES CD-ROM PlayStation” and a series of games already in development. Shortly after, Nintendo publicly denied Sony’s announcement,

revealing a new collaboration with Philips (deemed more advantageous for Nintendo). As part of the agreement with the latter company, Nintendo granted the use of its intellectual properties to develop games for the ”Philips CD-i,” a console not produced by Nintendo.

Figure 2.3: Prototype of the Sony-Nintendo Play Station

The big N not only faced legal repercussions from Sony but also risked a serious negative backlash from the Japanese business community. The company had violated a sort of ”unwritten law,” according to which a Japanese company should never side against another Japanese company in favor of a foreign one.

This betrayal marked a crucial turning point. Ken Kutaragi, disappointed by Nintendo’s move, went to Sony’s president, Norio Ohga, and proposed to continue developing their own gaming console. Despite Sony’s initial skepticism about video games, Ohga, impressed by Kutaragi’s determination, responded with the famous decisive order: ”Just do it!”. Thus, Kutaragi began working in secret on the development of a gaming console that would use CD-ROM technology and a powerful 32-bit processor, with 3D graphics capabilities that would rival the rest of the expensive professional workstations.

The new console was named ”PlayStation” as a symbol of a new era for video games, but it was initially coded with the name ”PS-X” as a challenge to Nintendo. Kutaragi had a clear goal: to create a console capable of delivering an unprecedented real-time gaming experience, with audio and visual quality that would render the traditional cartridge-based system obsolete.

Figure 2.4: Sony PlayStation

2.4 The Birth of Sony Computer Entertainment

In 1993, Sony decided to establish a new division dedicated to video games, called Sony Computer Entertainment, which would manage the development and distribution of the new console. Despite the skepticism of many within Sony, who considered the video game market a marginal sector, Kutaragi continued to push for the project's realization. The initial doubts about Sony's ability to enter a market dominated by Sega and Nintendo were soon put aside as the powerful hardware of PlayStation and its innovative graphic technology began to make headway. The success of PlayStation was largely determined by its support for high-quality

Figure 2.5: Sony Computer Entertainment Logo (1993-2016)
[4]

Figure 2.6: Psygnosis Logo
[5]

titles, such as "Virtua Fighter," which demonstrated that 3D games were not only possible but could also be extremely well-received by players. Sony also invested huge resources to attract developers and programmers, managing to gain the support of over 250 development houses in Japan, including major partners like Namco, Capcom, SquareSoft, and Konami.

One of the key decisions for the success of PlayStation was the acquisition of Psygnosis, a British development studio known for its games on Atari ST and Commodore Amiga. The Psygnosis team, along with SN Systems, contributed to the creation of the official development kit for PlayStation, the PsyQ Development Kit, making development for the platform easier. Psygnosis not only offered technical support but also created some of the most iconic titles for PS1, such as Wipeout, Novastorm, 3D Lemmings, Assault Rigs, and Destruction Derby. Later, Sony renamed the company Sony Interactive Entertainment, making it an integral part of its strategy for the console's success.

Historic was also the announcement of the console's price at the 1995 E3: 299 dollars. This price was revealed in a press conference lasting about a minute, following Sega's announcement that the cost of its Sega Saturn would be 399 dollars.
Sources: [61].

Figure 2.7: From left to right: Covers of the games WipEout, Novastorm, and Destruction Derby.

[6][7][8]

2.5 Market Assertion

In December 1994, PlayStation was finally launched in Japan, and its success was immediate. In less than a year, the console sold over a million units. On September 9, 1995, Sony officially launched PlayStation in the United States and Europe, with an unprecedented marketing campaign aimed at conquering a mature and sophisticated audience. The motto ”If you’re not ready for PlayStation, you’re not ready for the future” became a symbol of a radical change in the perception of video games as entertainment.

PlayStation not only managed to break into the video game market but also exceeded Sony’s expectations, becoming a cultural icon thanks to its powerful hardware, a vast library of exclusive games, and the ability to attract more and more external developers. Among the best-selling titles were ”Ridge Racer,” ”Tekken,” and ”Crash Bandicoot,” which quickly became classics in the history of video games. Sources: [62][63].

Figure 2.8: Examples of some successful PS1 games.
[9][10][11][12][13][14]

Figure 2.9: Some covers of the British magazine EDGE related to PlayStation. [1]

Chapter 3

The Hardware of PlayStation

3.1 The Models

Figure 3.1: Various PlayStation models. From bottom to top: SCPH-1000, SCPH-3000, SCPH-5501, and SCPH-9001.

[15]

Each PlayStation model has a specific code used to identify its version. The first part of this alphanumeric code consists of an acronym. For models intended for the consumer market, the code SCPH is used, which stands for ”Sony Computer PlayStation Hardware,” while for consoles distributed as development kits to developers, the code DTL-H is employed, an acronym for ”Development Tool Hardware.”

This acronym is followed by a series of four numbers. The first three numbers indicate the model, while the final digit corresponds to the target region for commerce. For the PAL market, there is an additional subdivision, marked by the letters A, B, and C, which correspond respectively to the Australian, British, and Continental European markets.
Sources: [64].

Below is a list of regions:

Code	Country	Boot ROM	Region	Video	Power Supply
0	Japan	Japanese	NTSC	NTSC	100V
1	USA/Canada	English	NTSC/C	NTSC	110V
2	Europe/Australia/PAL	English	PAL	PAL	220V
3	Asia	Japanese	NTSC	NTSC	220V

Table 3.1: Subdivision of regions and boot characteristics

3.1.1 The SCPH-1000 Model

The SCPH-1000 model identifies the first version of PlayStation, released in Japan on December 3, 1994. Subsequently, it was introduced in the United States (as SCPH-1001) and in Europe (as SCPH-1002) in 1995. This console not only allowed playing titles on CD-ROM but also enabled music playback from audio CDs. A choice that significantly contributed to its success. The ability to play music easily transformed the PlayStation into a stereo system, while the low cost of CDs benefited both Sony and developers, allowing games to be sold at competitive prices while still maintaining a good profit margin.

Figure 3.2: SCPH-1000 model packaging
[16]

The architecture of the SCPH-1000 actually had some structural issues. The laser lens was positioned in the upper left corner of the disc compartment, very close to the internal power supply, causing overheating and interfering with the performance of the optical block. Moreover, the support of the optical block was made of thin ABS plastic with a high concentration of weight

on the left side of the laser slide. Over time, this design led to misalignment of the lens in 85% of cases, compromising the reading of CD-ROMs. Some users attempted to solve the problem by turning the console upside down or adding a counterweight to the slide, but these operations voided the warranty. Later, Sony introduced a new aluminum optical block to correct these defects.

Figure 3.3: SCPH-1000 model disc compartment
[17]

Despite these defects, the SCPH-1000 is considered the model with the best video and audio quality of the entire PlayStation line. The presence of numerous capacitors and an RGB Encoder ensured a clear video signal, while the logic board integrated a 16-bit A/D audio converter with Delta-Sigma modulation, developed by Asahi Kasei Microsystems (model AK4309VM). This combination allowed realistic and linear audio, making the SCPH-1000 the reference model for audio quality.

Today, the SCPH-1000 is a rare model, both because it was the very first product and because few units were made before the transition to the next model. Moreover, it is the only one to include, in addition to the ”AV Multi-Out” connector, also individual RCA connectors and a 5-pin S-VIDEO port, which were removed in later versions.

Figure 3.4: Back of the SCPH-1000 model
[18]

It is important to emphasize that the SCPH-1000 code does not correspond to the same characteristics in models intended for international markets. The SCPH-1001 (USA) and SCPH-1002

(Europe) versions, released in 1995, were based on the architecture of the Japanese SCPH-3000 model, launched around the same time to solve the initial structural problems. Therefore, the ”true” SCPH-1000 model is exclusive to the Japanese market, while the Western models follow the specifications of the SCPH-3000 series. Sources: [19].

3.1.2 The SCPH-3000 Model

Figure 3.5: SCPH-3500 model packaging
[20]

The SCPH-3000 and SCPH-3500 series were released exclusively in Japan. The SCPH-3500 series was sold within a bundle called ”Fighting Box,” aimed at fans of fighting games, which included 2 controllers and a Memory Card. Starting from these versions, the S-video output was removed, a feature present in the previous SCPH-1000 models.

Figure 3.6: Back of the SCPH-3000 model
[21]

Although these models were never marketed in the West, the SCPH-1001 (USA) and SCPH-1002 (Europe) versions, released in September 1995, are based on the same architecture as the Japanese SCPH-3000. Therefore, despite belonging to the SCPH-1000 series, the Western versions could be considered part of the SCPH-3000 series due to their characteristics. Sources: [65].

3.1.3 The SCPH-5000 Model

With the introduction of the 5000 series, Sony made some changes to the hardware structure and design of PlayStation, while maintaining most of the original features. One of the main updates concerned the position of the laser lens, which was moved to the right and placed in a central area, away from the electrical compartment, to improve stability and durability. On the back of the console, the classic AV Multi-Out replaced the three individual RCA connectors (red, white, and yellow), while still keeping the AKM DAC chip for audio management, ensuring compatibility with AV systems without altering audio quality.

In Europe, the SCPH-5502 series introduced aesthetic changes to align with the symbolic conventions of the buttons: the ”Power” and ”Open” buttons were replaced with more recognizable universal symbols, standardizing the design for different markets.

Figure 3.7: The position of the optical reader has been changed compared to previous models [22]

From a hardware point of view, the 5000 series does not present significant changes compared to the previous series, except for updates to the BIOS and new packaging, which in some markets include bundles with specific games or the new Dual Shock controller. This series constitutes the second model released in the West, following the SCPH-1000 series. In Japan, however, the 5000 series represents the third model, having been preceded by the SCPH-3000 series, equivalent to the SCPH-1000 series released in Europe and the United States.

Notably, the SCPH-5903 model is the only one in the 5000 series capable of reading Video CDs, designed exclusively for the Asian market, where the Video CD format was more popular. This model differed in its ability to reproduce multimedia content and represented an attempt to expand the console’s offering towards home entertainment functionalities. Sources: [66].

Figure 3.8: PlayStation Video CD. SCPH-5903 model
[23]

3.1.4 The SCPH-7000 Model

The SCPH-7000 series represented an important evolution for PlayStation, with innovations aimed at both the operating system and the aesthetics of the console. While maintaining an identical external design to the previous model, Sony introduced a new graphical interface and a system of visual effects for the music player, previously available only on the ”DEMO ONE” disc.

From a hardware perspective, this version eliminates the standard composite video and stereo audio outputs, leaving the AV Multi-Out as the only option for audio-video connection. The SCPH-7000 series is also the last to mount the AKM DAC chip (AK4309AVM), replaced in a rare variant of the model with the BB DAC (PCM1729E). The internal components were also reinforced, definitively solving the overheating issues of the console.

Figure 3.9: Back of the SCPH-7000 model
[24]

This series also marked the adoption of the Dual Shock as the standard controller. With the SCPH-7003 model, Sony expanded the distribution of PlayStation to other regions of Asia. Finally, to celebrate the 10 million units sold in the three main markets (Japan, USA, Europe), Sony released a limited edition SCPH-7000W, known as the ”10 Million Model,” characterized by an exclusive Midnight Blue color. Sources: [67].

Figure 3.10: Posters related to the launch of the DualShock controller and the Midnight Black edition

[25][26]

3.1.5 The SCPH-9000 Model

With the SCPH-9000 series, Sony removed the parallel I/O port to block the use of devices like Action Replay and Gameshark, thus preventing the use of illegal cheats.

Figure 3.11: Back of the SCPH-9000 model

[27]

This model also improved the durability of the laser, but the audio quality suffered a significant decline compared to previous versions. The playable demo CD was also removed from the packaging.

Launched globally in June 1999, the SCPH-9000 series hit the market at a price of $129.99 in the United States and 15,000 yen in Japan. Sources: [68].

3.1.6 The SCPH-100 Model

Despite the debut of PlayStation 2 in March 2000, Sony decided not to withdraw the original PlayStation, given its continued popularity and high sales. Thus, on July 7 of the same year,

PSone was launched, a redesigned and more compact version of the original PlayStation.

PSone featured a more articulated and easily portable design, with external power supply to reduce weight and overheating. It did not include either the parallel port or the AV serial cable for multiplayer gaming. Additionally, the adapters for the Memory Card and DualShock controllers were soldered into the console. The light gray color of the PSone, called ”Light Gray,” gave it a modern and distinctive appearance.

The console also allowed the use of an additional display, making it completely portable. Furthermore, Sony collaborated with NTT DoCoMo to enable connection with I-Mode mobile phones via a dedicated cable, available only in Japan.

Figure 3.12: PSone model with its peripherals
[69]

PSone continued to be produced until March 2006, when Sony announced the end of its production. In total, 102.49 million PlayStation units were sold, of which 28.15 million were PSone. Sources: [28].

3.1.7 The DTL-H Models

The PlayStation debug consoles were similar to the retail versions but designed to facilitate game testing. They were generally in blue or green colors, with some special units in gray for demonstration purposes. Unlike the development kits, which had 8 MB of RAM, the debug consoles had only 2 MB and used standard boot ROMs. The main difference was in the CD controller, modified to identify as ”licensed” any disc with a data track, allowing developers to test games on written CDs. This also allowed the launch of games from other regions, although it was not officially supported. Sony produced specific debug consoles for each region, and its ”technical requirement checklist” required game testing on these units. The differences in the

case colors derived from a hardware change: the blue models (DTL-H100x, DTL-H110x) used the Rev. A or Rev. B chip, while the green ones (DTL-H120x) mounted the Rev. C chip, faster in some operations. Programmers had to test games on both types of machines, as the hardware differences affected the behavior of the games. Sources: [70].

In 1997, Sony released a special version of PlayStation called Net Yaroze in the West (it was released a year earlier in the Japanese market). More expensive than the original console (750 dollars compared to the 299 dollars of the standard model), it was also harder to find, as it was available only by mail order. Yaroze can be translated as "Let's work together!". The console had a matte black finish, instead of gray, and came with tools and instructions that allowed users to program games and applications for PlayStation, without the need for a professional development kit. The Net Yaroze did not offer all the functionalities of the complete development kit, such

Figure 3.13: On the left, the DTL-H1202 Debugging Station model. On the right, the Net Yarooze DTL-H300x kit

[29][30]

as on-demand support and code libraries that official developers had access to. Moreover, programmers were limited to 2 MB of total RAM for games. One of the unique features of the Net Yaroze was the absence of regional locks, allowing the execution of games from any region. However, three specific models were produced for each region: Japan (DTL-H3000), North America (DTL-H3001), and Europe/Australia (DTL-H3002), with differences in video support: the European/Australian model used PAL mode, while the other two used NTSC. In any case, this console was still not able to reproduce the CD-R format, so it was not possible to create standalone Yaroze games without a modified PlayStation. Sources: [71].

3.2 Hardware Overview

After this overview of the various PSX models, the analysis will focus on the hardware in more detail. In this thesis, given its importance, the attention will be directed to the first model, the SCPH-1000, as it itself marked the beginning of the PlayStation story, from 1994 to today. Sources: [72][73][74].

3.2.1 System Architecture

PlayStation consists of groups of processors and devices that manage functions of all kinds, such as video and audio, around a 32-bit RISC CPU, illustrated in the block diagram below taken from the official Sony PlayStation hardware manual.

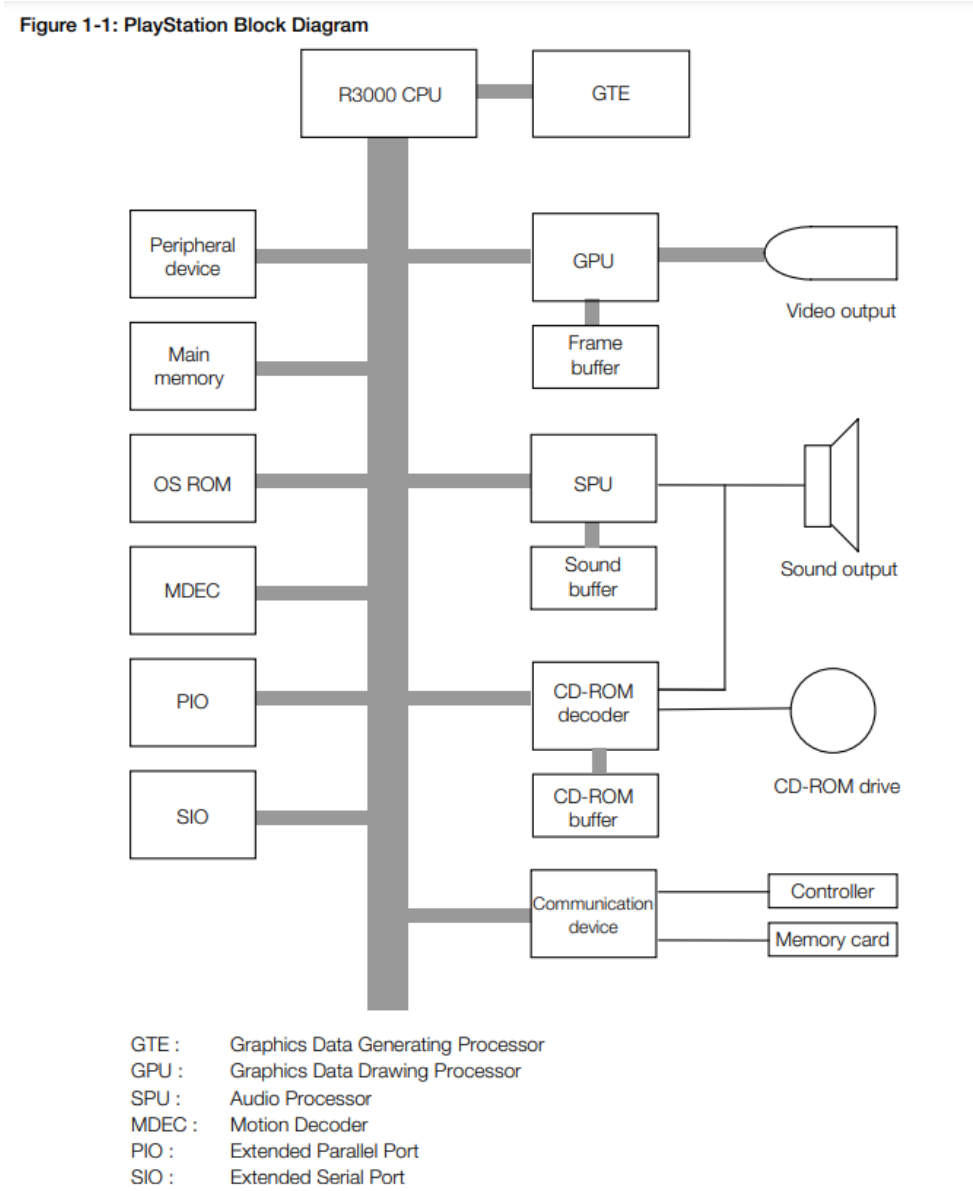


Figure 3.14: Block diagram of PlayStation.

The console is equipped with a 33.86 MHz 32-bit MIPS R3000A CPU, a significant technological advancement compared to previous generations of consoles that operated at about 7 MHz (Sources: [75]) and, even earlier, at 1.7 MHz (Sources: [76]).

Among the coprocessors, we find the CP0 for system control, the CP2 (Geometry Transformation Engine - GTE) for geometric transformation, and the MDEC (Motion Decoder) for video decoding.

The main memory consists of 2 MB of EDO RAM, while the VRAM amounts to 1 MB, used to store textures, frame buffers, images, fonts, and more. The graphics system includes a GPU based on the SCPH-9000 chip, responsible for rasterization.

The SPU (Sound Processing Unit) manages audio processing, offering 24 ADPCM channels at 16 bits. Finally, the CD Subsystem is equipped with a DSP (Digital Signal Processing) dedicated to controlling the motor and laser for data reading.

3.2.2 CPU Introduction

The CPU is the heart of the system and is designed with memory and an interrupt controller, based on a 32-bit RISC architecture. It integrates an instruction cache ("I cache") and a scratchpad memory, allowing direct management of the main memory.

3.2.3 Graphics System Introduction

The graphics system of PlayStation consists of two main elements: a processor for creating graphic data (GTE) and one for graphic drawing (GPU).

The GTE performs coordinate transformations and light source calculations as a coprocessor of the CPU. The GPU, on the other hand, executes the polygon drawing instructions imparted by the CPU. The CPU also has a non-shared two-dimensional addressing space where the frame buffer is mapped. The frame buffer is the portion of memory dedicated to the temporary storage of data that make up the image or frame to be displayed on the screen. It will be analyzed in more detail in the following chapters.

The fundamental principle of PlayStation's graphics system is simple: the CPU transmits the drawing data, such as textures, polygons, and color palettes (CLUT), to the GPU, which saves this information in the frame buffer. Subsequently, the GPU proceeds with the drawing of the polygons based on the color and coordinate information provided by the GTE.

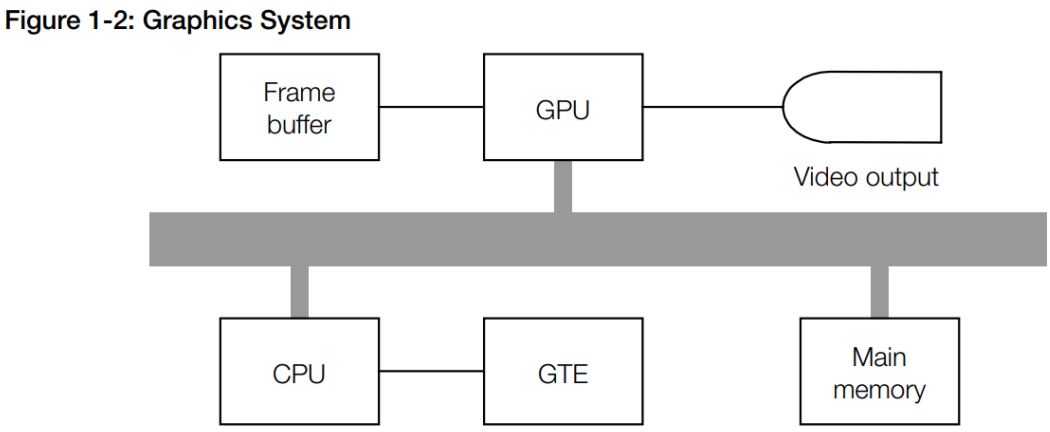


Figure 3.15: PlayStation graphics system

3.2.4 Audio System Introduction

The audio system of PlayStation also consists of two components, a processor for sound reproduction (SPU) and a CD-ROM decoder.

The SPU (Sound Processing Unit) integrates a 24-voice ADPCM sound source, the operation of which is managed by the CPU. This unit has a dedicated addressing space, where an audio buffer is mapped. The CPU sends ADPCM data to this buffer, while the SPU reproduces them using the information related to Key On/Key Off and modulation.

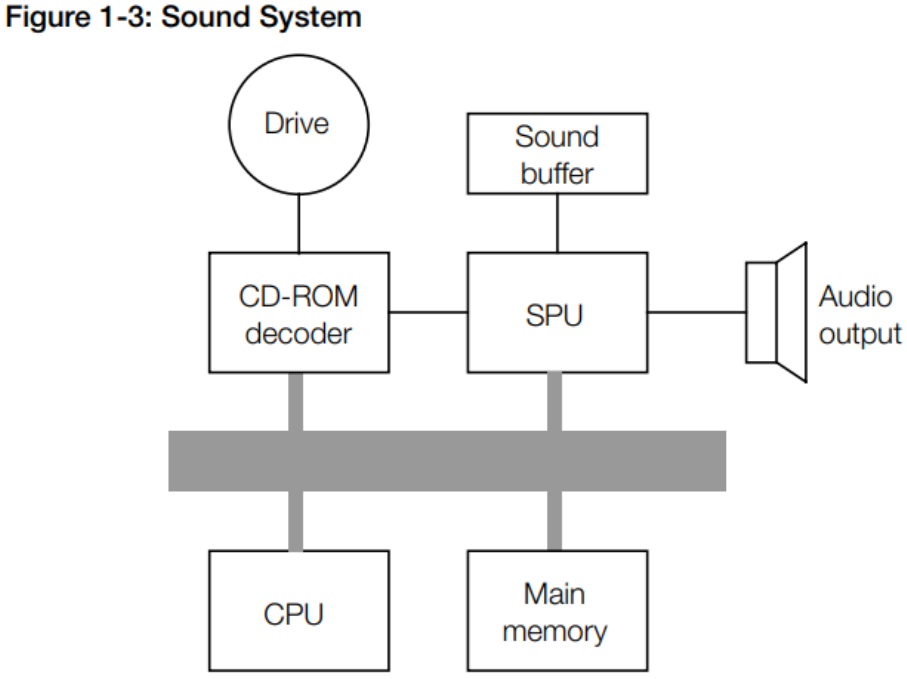


Figure 3.16: PlayStation audio system

The CD-ROM decoder reproduces audio while reading PCM or CD-ROM XA ADPCM data present on the disc. The audio output of the decoder is temporarily transferred to the SPU, where it is mixed with the output of the SPU to generate the final track.

3.2.5 Additional Supports

The CD-ROM system consists of the components necessary to read CD-ROMs, such as the reader and decoder, and supports CD-DA, CD-ROM XA, and PlayStation formats.

The Data Expansion Engine performs high-speed inverse DCT transformations and manages the expansion of JPEG and MPEG (frame compressed only) data.

The PlayStation controller is an interface that transmits the player's intentions to the applications and allows connecting numerous controllers via the MultiTap peripheral. The memory card is a device for saving data after "reset" or power off, with the possibility of connecting multiple cards via.

Finally, there are two types of expansion ports available: serial and parallel.

Figure 3.17: An example of the MultiTap unit for Multiplayer gaming [31][32]

3.2.6 Motherboard Analysis

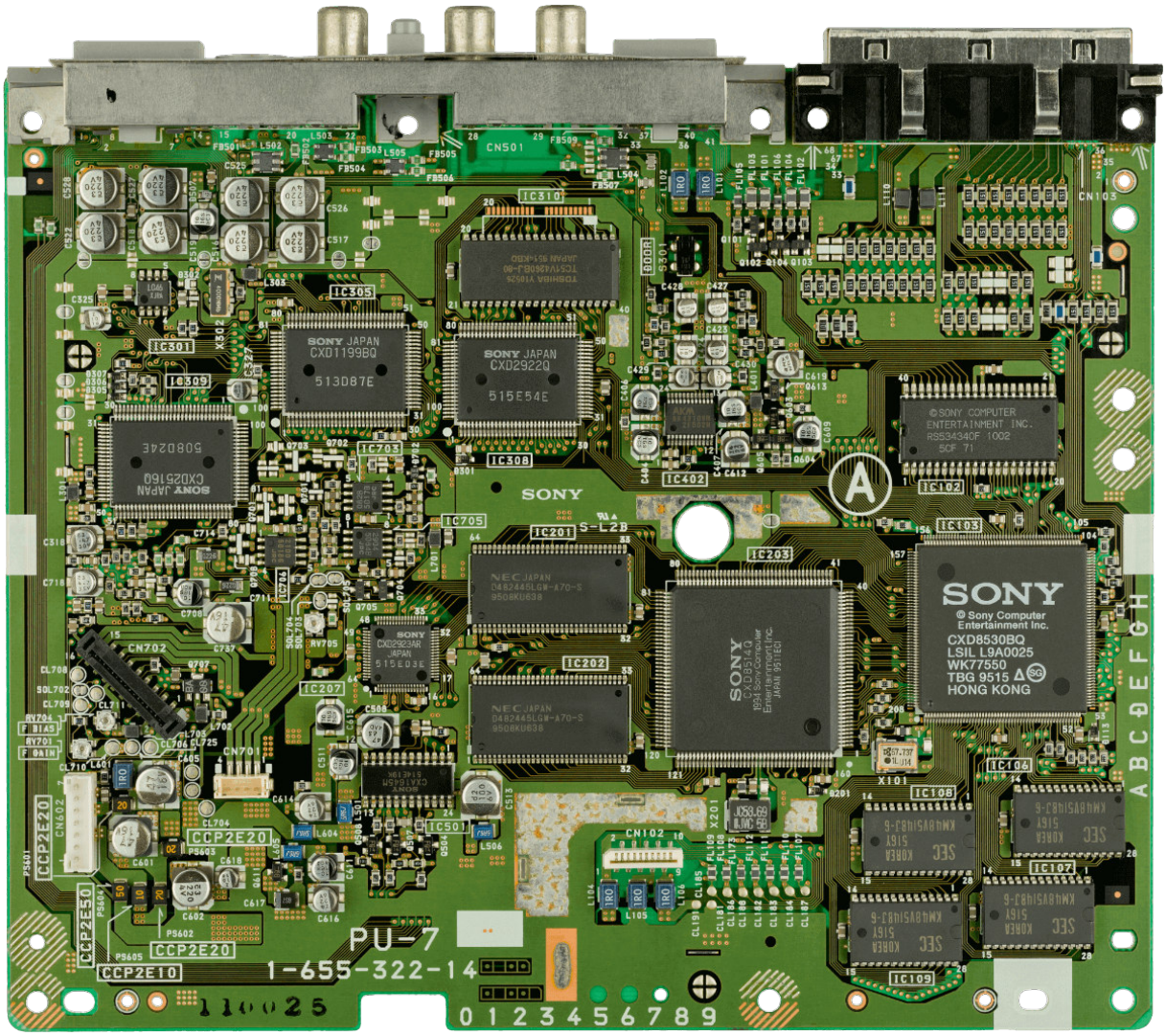


Figure 3.18: Motherboard of the SCPH-1000 model

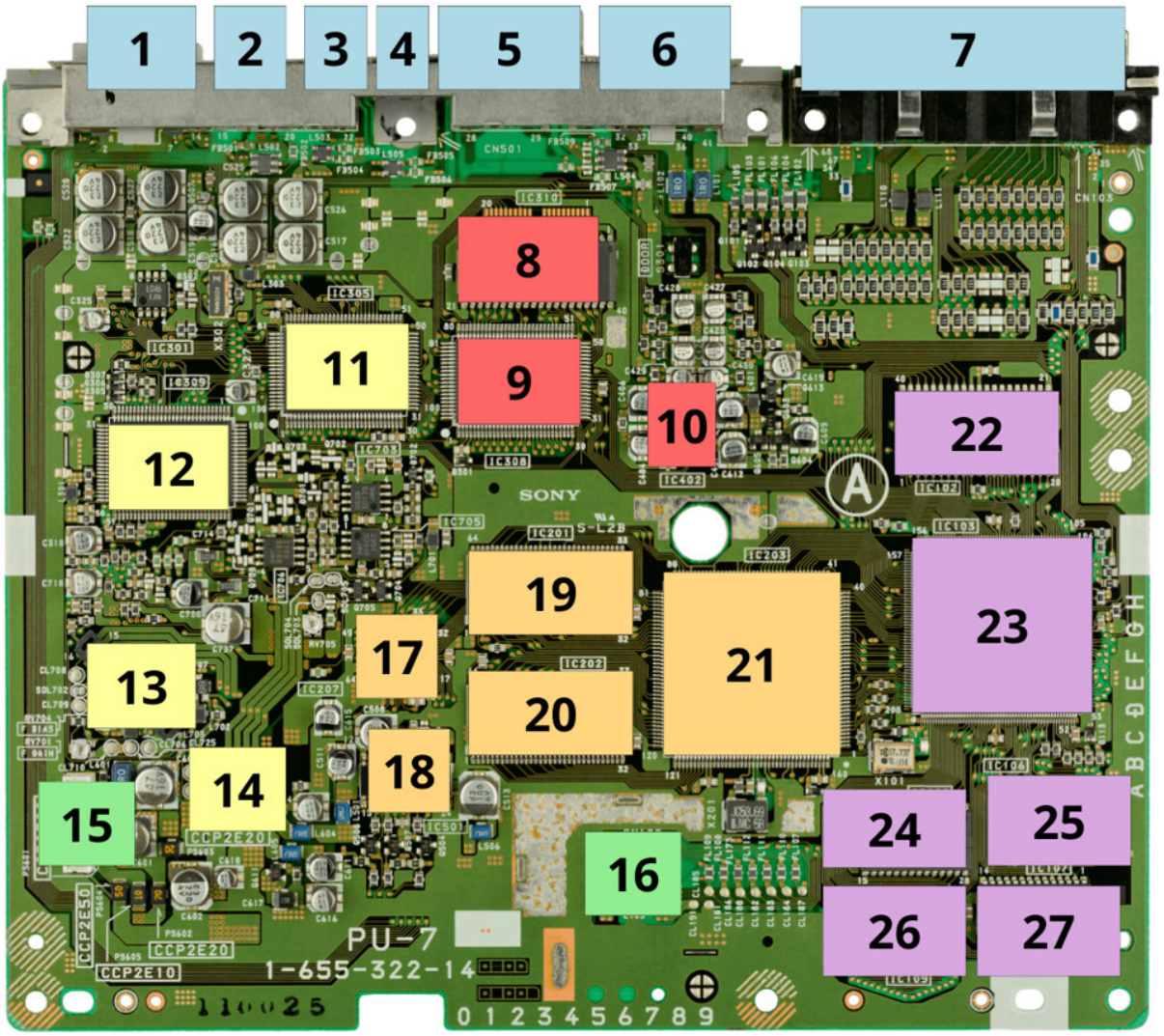


Figure 3.19: Motherboard of the SCPH-1000 model with details

1. AV Multi Out
2. S-Video Out
3. Composite RCA Out
4. RFU DC Out
5. Stereo RCA Out
6. Serial I/O Port
7. Parallel I/O Port
8. 512 KB DRAM
9. Sound Processing Unit
10. Serial DAC
11. CD-ROM Controller
12. CD-ROM DSP
13. CD Drive Data
14. CD Drive Power
15. PSU board Connector
16. Front panel Connector
17. RGB Encoder
18. Composite Encoder
19. 512 KB VRAM
20. 512 KB VRAM
21. Sony GPU
22. 512 KB BIOS ROM
23. Sony CPU SoC
24. 512 KB EDO DRAM
25. 512 KB EDO DRAM
26. 512 KB EDO DRAM
27. 512 KB EDO DRAM

3.3 The PSX CPU

3.3.1 Processor Characteristics

The processor of the PlayStation 1, identified by the code ”Sony CXD8530BQ”, is classified as a ”System on a Chip” (SoC) due to the presence of integrated co-processors that support the execution of different tasks, all within the same chip.

Figure 3.20: Sony CXD8530BQ
[33]

Following are its main characteristics:

- It has a clock frequency of 33.86 MHz.
- The processor model is an R3000A developed by MIPS and LSI Logic.
- It is a CPU with a 32-bit RISC architecture. RISC stands for Reduced Instruction Set Computer.
- It is an ISA MIPS I. MIPS stands for Microprocessor without Interlocked Pipelined Stages. Among its main features, the words are 32 bits long and the instruction set includes multiplication and division operations.

  ISA (Instruction Set Architecture) is a specification that defines the aspects of the CPU visible to a programmer in machine language or to an assembler, such as registers, memory model, input/output, and exceptions. Although the hardware implementation may vary, it must ensure compatibility with the ISA, allowing a program compiled for a given ISA to be executed on any hardware that implements it.

- It has 32 general-purpose registers and 2 registers for multiplication and division. All registers are 32 bits, and one register (R0) is always zero.

- 32-bit data bus: In PlayStation, the data bus is divided into two:
  - Main Bus (32-bit): Connects the MDEC unit and the GPU.
  - Sub Bus (16/8-bit): Connects the other chips and I/O interfaces. This bus is managed by the Bus Interface Unit, which also allows access to special ports of the GPU and SPU.
- 32-bit address bus: Allows access to up to 4 GB of physical memory, including RAM and memory-mapped I/O.
- 5-stage pipeline: Allows simultaneous execution of up to five instructions. These instructions are respectively Fetch, Decode, Execute, Memory, and Write. It is not necessary to wait for one stage to finish before starting the next. Sources: [77].
- 4 KB instruction cache: It can be ”isolated”, allowing the program to directly manipulate the content of the instruction cache.
- Absence of data cache: There is no cache for data. However, 1 KB of memory normally reserved for data cache is mapped to a fixed address. This area, known as Scratchpad, is made up of high-speed SRAM.
- 2 MB of RAM for general use. Curiously, on the motherboard, EDO-type chips have been mounted, which are slightly more efficient than traditional DRAM, thanks to lower latency.

At some point, subsystems like graphics, audio, or CD will require large amounts of data at high speeds, exceeding the CPU's ability to meet such demand.

To meet this need, the CD-ROM Controller, MDEC, GPU, SPU, and parallel port can use a dedicated DMA controller to manage data transfers. DMA takes control of the main bus, allowing faster transfers than using the CPU, which is still necessary to configure the transfer. However, when DMA is active, the CPU cannot access the main bus and remains inactive, unless it is working on data in the Scratchpad.

3.3.2 Processor Origins

In the 1990s, the landscape of CPUs saw a significant transformation. The 8-bit processors, which had dominated the market for a long time, were now obsolete and out of the spotlight. Similarly, the Motorola 68000, along with other 16-bit processors that had been successful at the end of the 1980s, was beginning to make way for new designs destined to replace them.

At first, it seemed that technological development had reached a standstill. However, a new generation of relatively unknown CPUs began to enter the mass market. These new designs

were, in many cases, the result of academic research, aimed at testing and demonstrating particular design ideas. Among these new processors, significant examples include MIPS, PowerPC, SPARC, and ARM.

These chips had a common characteristic: they all followed the RISC (Reduced Instruction Set Computer) architecture, which radically changed the approach to CPU design and programming. A key principle of the RISC architecture established that instructions should not combine register operations with memory operations, allowing designers to simplify the circuit that executed the instructions and exploit parallelism to improve performance.

The first commercial processor to implement a RISC design was the "MIPS R2000" from MIPS Computer Systems, which was adopted in several UNIX workstations. However, it was only in 1987 that MIPS chips gained more visibility, when Silicon Graphics Incorporated (SGI) adopted them to power their systems. SGI became a key player in the field of computer graphics, thanks to the development of hardware-accelerated vertex pipeline, an innovation that had previously been done by software. The acquisition of MIPS by SGI consolidated the company's position in both the CPU and advanced graphics markets.

Before the PlayStation design began, MIPS had already adopted a licensing-based business model for their designs, selling their CPUs in the form of licenses that allowed licensees to customize and produce them independently. Among the options offered was the R3000A CPU, which appeared as a cost-effective solution, but did not belong to the company's flagship line (unlike the R4000, chosen by other manufacturers later).

Meanwhile, Sony, while internally designing its own audio and graphics chips, needed a processor capable of supporting both components. The CPU had to be powerful enough to exploit the potential of the graphics and audio chips developed by Sony, but also cheap enough to keep the console's price competitive.

In this context, LSI Logic, a semiconductor manufacturer that had obtained a license from MIPS, offered a program called "CoreWare", which allowed companies to develop customized CPUs using a series of modular blocks. Among these, was the 'CW33300' block, a core derived from the LSI LR33300, a chip already marketed by LSI.

It was found that the LSI LR33300 and CW33300 chips were binary compatible with the MIPS R3000A family, although they presented slight architectural differences in some areas. However, the programming interface (MIPS I ISA) remained unchanged, allowing compatibility between the different models.

In the end, Sony commissioned LSI to produce the CPU package for the PlayStation. The Japanese company chose the CW33300 core, made some modifications, and combined it with other blocks to create the chip that would be used on the motherboard of the PlayStation console.

3.3.3 Coprocessors Analysis

System Control Coprocessor

Identified as CP0, the System Control Coprocessor is a common element of MIPS CPUs. It controls the cache implementation, allowing direct access to the Scratchpad and Instruction Cache. Moreover, it is also responsible for managing interrupts, exceptions, and breakpoints, the latter being useful during the debug process.

Geometry Transformation Engine

The ”CP2” or Geometry Transformation Engine (GTE) is a math processor that accelerates vector and matrix calculations. It offers operations useful for 3D graphics such as:

- Multiplication and addition of vectors or matrices
- Perspective transformations
- Clipping operations
- Interpolation functions
- Functions for lighting and color operations

Motion Decoder

The task of the Motion Decoder, also known as ”MDEC” or ”Macroblock Decoder”, is to decompress ”Macroblock” into a format understandable by the GPU. A Macroblock is a data structure that represents an image with a coding similar to that of the JPEG format.

MDEC decompresses bitmap images made up of 8x8 pixels at 24 bpp (bits per pixel) at a time. This allows it to stream full-motion video (FMV) with a resolution of 320x240 px at 30 frames per second. DMA (Direct Memory Access) is used to transfer the compressed data from the CD-ROM to the RAM and then to the MDEC. The path is also followed in reverse, but in this case, the destination is the VRAM.

The Missing Unit

Unfortunately, Sony did not provide the SoC with a CP1, that is, a ”Floating Point Unit” (FPU). This does not mean that the CPU cannot perform arithmetic operations with decimal numbers, it simply cannot do it fast or accurately enough.

Game logics that include operations such as physics implementation and collision detection can still be managed using ”Fixed-point” arithmetic. With this system, decimal numbers are

represented with a fixed number of decimals, which results in a loss of precision after some operations. A deeper insight into this topic will be covered in chapter 5.6.

3.3.4 Memory with Addressable Access

When talking about memory, it can be conceived as a one-dimensional array of addresses, where each address corresponds to a memory location. Each memory address is represented by a 32-bit number, but each address refers to a single unit of memory, namely a byte.

To represent these addresses, hexadecimal notation is used, allowing a more compact reading of the numbers within the addresses themselves.

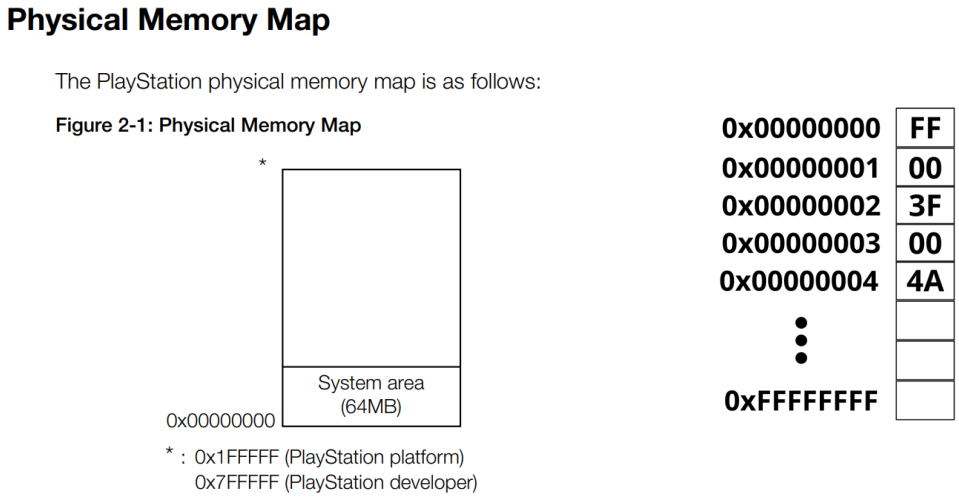


Figure 3.21: Graphic representation of PlayStation memory

The question that arises spontaneously is: how many addresses are available?

The PlayStation's address bus is 32 bits, which allows addressing a maximum of 2^{32} bytes of memory, equal to 4 GB. In this context, it does not refer exclusively to RAM, but to the entire memory of the PlayStation system.

In modern systems, CPUs and operating systems use virtual memory, which allows the use of virtual addresses that do not directly correspond to physical memory. These addresses are managed by the operating system and mapped to physical addresses through a memory management unit (MMU). However, in the case of PlayStation, the absence of an MMU means that virtual memory is not used, but only physical memory. The hardware is not designed to abstract addresses through a memory management system.

Endianess Concept

In memory management, CPUs adopt different approaches for byte ordering, choosing different modes to save data in memory. Sources: [78].

A useful example to illustrate this variation is saving a hexadecimal number, for example, 0x12345678. An intuitive way to represent it in memory might be to save it in order from left to right: the first byte would be 0x12, followed by 0x34, 0x56, and finally 0x78. This approach is known as Big Endian, where the most significant byte (MSB) is saved first compared to the least significant byte (LSB). Some architectures, like the Motorola 68000 family, adopt this mode of representation. Sources: [79].

PlayStation, on the other hand, uses a different approach, called Little Endian. In this case, the least significant byte is saved first, followed by the others in increasing order of significance. Returning to the example of the number 0x12345678, the order of saving in memory would be 0x78, 0x56, 0x34, and finally 0x12. The MIPS R300 CPU, used in the PlayStation, is configured to adopt the Little Endian format.

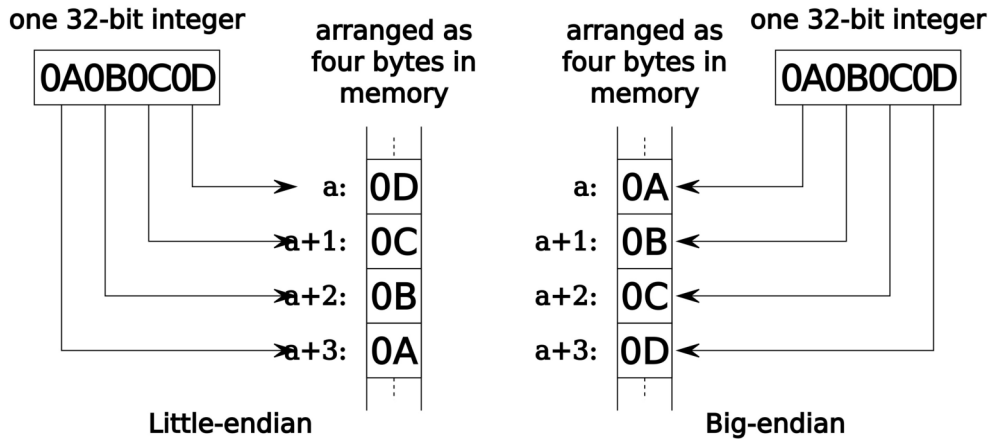


Figure 3.22: Example of byte ordering
[34]

It is important to note that MIPS architectures, in general, do not impose a mandatory endianess. Many MIPS processors can be configured to operate in either Big Endian or Little Endian mode. However, in the specific case of PlayStation, the CPU is designed to use exclusively the Little Endian format, where the least significant byte of a number is saved first in memory.

Memory Mapping

A crucial aspect in systems like PlayStation is memory mapping, which associates memory addresses with various hardware components. The CPU needs to directly access devices like I/O ports and peripherals to interact with the system. This is done by mapping these devices to specific memory addresses.

For example, to access the GPU, check the status of the CD Controller, or read from RAM, it is necessary that these devices have defined memory addresses. This way, the read and write operations are simplified, improving the efficiency of communication between the CPU and hardware.

Table 3.2: PlayStation Memory Map pt.1

Description	KUSEG	KSEG0	KSEG1	Space
Main RAM (first 64K reserved for BIOS)	00000000h	80000000h	A0000000h	2048K
Expansion Region 1 (ROM/RAM)	1F000000h	9F000000h	BF000000h	8192K
Scratchpad (D-Cache used as Fast RAM)	1F800000h	9F800000h	–	1K
I/O Ports	1F801000h	9F801000h	BF801000h	8K
Expansion Region 2 (I/O Ports)	1F802000h	9F802000h	BF802000h	8K
Expansion Region 3 (whatever purpose)	1FA00000h	9FA00000h	BFA00000h	2048K
BIOS ROM (Kernel) (4096K max)	1FC00000h	9FC00000h	BFC00000h	512K

Table 3.3: PlayStation Memory Map pt.2

Description	KSEG2	Space
I/O Ports (Cache Control)	FFFE0000h	0.5K

Table 3.4: Additional memory not mapped to the CPU bus

Size	Description
1024K	VRAM (Framebuffers, Textures, Palettes) (with 2KB Texture Cache)
512K	Sound RAM (Capture Buffers, ADPCM Data, Reverb Workspace)
0.5K	CDROM controller RAM
16.5K	CDROM controller ROM (Firmware and Bootstrap for CPU MC68HC05)
32K	CDROM Buffer (IC303) (32Kx8)
128K	External Memories (Memory Card)

- **Kernel Memory**: KSEG1 is the normal physical memory (without cache), while KSEG0 is a copy of it (but with the cache enabled). KSEG2 is generally intended to contain the kernel's virtual memory but in the case of PSX contains the cache control I/O ports.
- **User Memory**: KUSEG is intended to contain 2 GB of virtual memory (on extended MIPS processors) but PSX as has been reiterated previously does not support virtual memory. In this case, KUSEG simply contains a copy of KSEG0/KSEG1 (in the first 512 MB). If you try to access memory in the remaining 1.5 GB you get an error.
- **Code Cache**: The Code Cache operates in the regions with cache (KUSEG and KSEG0).
- **Data Cache (Scratchpad)**: MIPS CPUs usually have a Data Cache but in the case of PSX Sony exploited this functionality differently, using it as a "Scratchpad". In other words, the "Data Cache" is mapped to a fixed memory position between 1F800000h and 1F8003FFh (thus used as "Fast Ram", rather than as a cache).

Table 3.5: KUSEG, KSEG0, KSEG1, and KSEG2 memory regions

Address	Name	Size	Privileges	Code-Cache	Data-Cache
00000000h	KUSEG	2048M	Kernel/User	Yes	(Scratchpad)
80000000h	KSEG0	512M	Kernel	Yes	(Scratchpad)
A0000000h	KSEG1	512M	Kernel	No	No
C0000000h	KSEG2	1024M	Kernel	(No code)	No

- **Memory Mirrors**: The KUSEG, KSEG0, and KSEG1 regions of 512MB are mirror images of each other. Within these regions, there are further mirrors:

  - 2MB of RAM and 512K of BIOS ROM can be mirrored in specific memory areas, with default active options for RAM and inactive for BIOS.

  - The Expansion Hardware, if present, can be mirrored in the respective region, as well as the seven DMA control registers (1F8010x8h mirrored in 1F8010xCh).

  - The sizes of the RAM, BIOS, and Expansion regions can be configured via software, and the Scratchpad is mirrored only in KUSEG and KSEG0, not in KSEG1.

Some examples of register usage can be:
- 0x1F801814 READ : Reads the GPU status
- 0x1F801814 WRITE : Sends commands to the GP1 (Display Control)
- 0x1F801820 WRITE : Sends commands to the MDEC

3.4 The PSX GPU

The GTE manages a significant part of the graphics pipeline, taking care of operations like perspective transformation, which converts three-dimensional space into a two-dimensional plane following the camera's perspective, and lighting processing. The data obtained is then transferred to Sony's proprietary GPU for rendering.

Figure 3.23: Graphics processor (GPU) used in the SCPH-100X series of the Sony PlayStation [35]

3.4.1 VRAM

The system's VRAM, with a capacity of 1 MB, is used to store the frame buffer, textures, and resources necessary for the GPU to process the scene. The CPU can use DMA to transfer data to this memory area.

In summary, the CPU provides the GPU with geometric data, such as vertices, which may include rendering requests, setting changes, or VRAM manipulations.

The GPU, responsible for drawing the geometry, performs various processes including applying clipping algorithms to exclude polygons that are outside the camera's field of view.

At the end of its operations, the result is transferred from the GPU to the frame buffer of the VRAM, from where the video encoder takes care of its transmission to the screen.

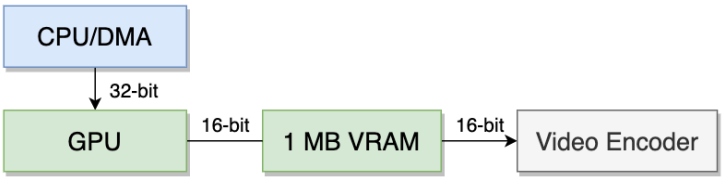


Figure 3.24: Graphic layout of the VRAM

3.4.2 Video Outputs

- **AV Multi Out**: Supported all previous signals, except RFU, and included RGB and a 5+ Volt power line.
- **RCA**: Provided composite video signals.
- **S-Video**: Allowed the output of Luma + Sync (combined) and Chroma.

- **RFU DC**: Designed for connection to an RF modulator, was eliminated shortly after launch.

The subsequent revisions of the console simplified this configuration, progressively eliminating all ports except the AV Multi Out.

3.5 The PSX SPU

Sony's Sound Processing Unit (SPU) takes care of the audio component of PlayStation. This chip supports 24 channels of 16-bit ADPCM samples with a sampling rate of 44.1 kHz (Standard audio quality for CDs).

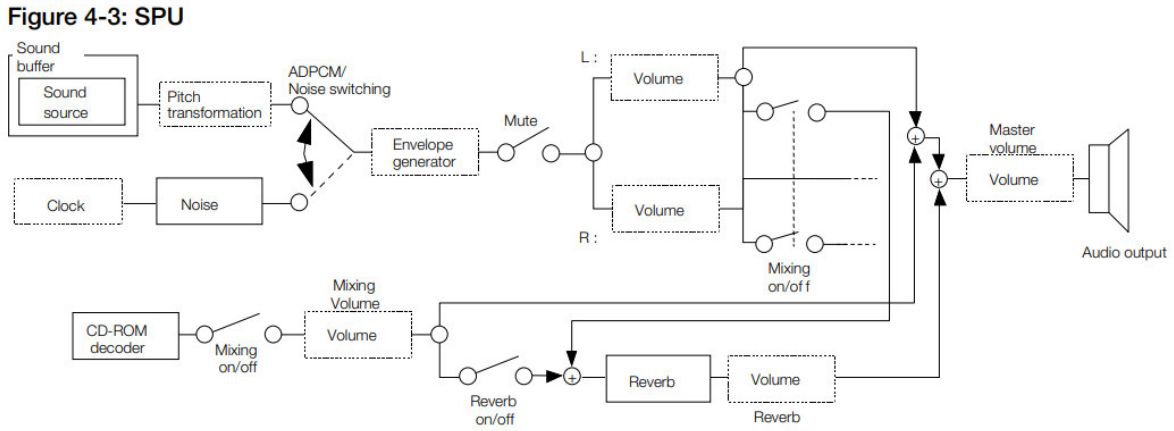


Figure 3.25: Layout of the Sound Processing Unit. Image taken from the original documentation.

This chip offers the following functionalities :

- **Pitch Modulation**: Allows automatic alteration of the pitch of audio tracks, useful in transitioning from one track to another.
- **Frequency modulation**: Allows alteration of the track frequency.
- **ADSR Envelope**: Package of audio properties available to modulate the amplitude.
- **Looping**: Allows the repetition of the track in a loop.
- **Digital reverb**: Digital simulation of reverberation in a given environment.

PlayStation has 512 KB of DRAM (Sound Ram) available as audio buffer. The use of this memory is allowed only to the CPU via DMA and to the CD controller. 4KB are reserved for the SPU to process audio tracks from the CD while the remaining 508 KB are used by games to store samples. If reverb is activated, the latter amount is further reduced.

The CD controller can directly send samples to the audio mixer, without having to go through the DRAM or involve the CPU. Moreover, they can be compressed with the 'XA' encoding, which the SPU can decode in real-time.

Furthermore, thanks to the large amount of space available on CD-ROMs, the music tracks can be sent directly to the audio chip. An operation that, in the past, was not possible due to the space and hardware limitations of the previous generation consoles, which were forced to rearrange the tracks in sequences or to use predefined waveforms.

3.6 Management of PSX I/O Interfaces

3.6.1 CD Module

The section that manages the CD drive can be considered as a computer integrated within the console. This subsystem is composed of the following elements:

- **DSP (Digital Signal Processing)**: controls the motor and laser of the unit, as well as processes the RF signal of the latter.

- **SUB-CPU**: a CPU made up of a Motorola 68HC05 microcontroller, 512 B of RAM, and 16 KB of ROM. In short, the Sub-CPU executes a local program stored in ROM and controls the DSP. This program checks the integrity and originality of the disc.

- **CD-Controller**: acts as an intermediary between the CD Subsystem and the rest of the console, receiving commands from the main CPU in FIFO mode. Communicates directly with the Sub-CPU and receives CD data from the DSP. Additionally, it integrates a DMA unit and is connected to the SPU to allow direct audio streaming.

- **SRAM**: 32 KB of memory connected to the controller, used as a buffer for reading data from the disc.

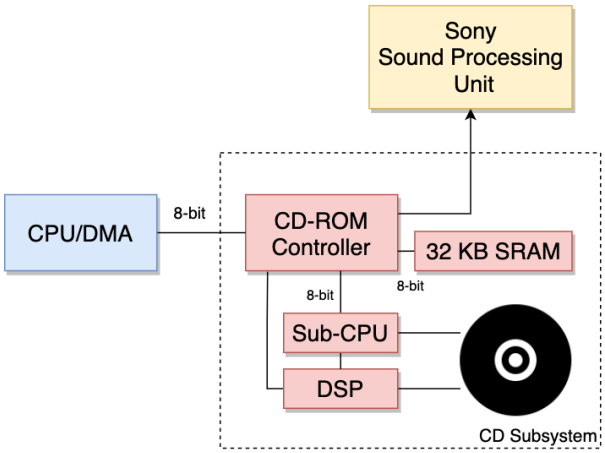


Figure 3.26: CD subsystem of PlayStation
[33]

3.6.2 Front Ports

The PlayStation controller and Memory Card slots are electrically identical, so the address of each one is permanently coded. Sony changed the physical shape of the ports to avoid confusion and connection errors.

Communication with these devices occurs via a serial interface. The commands sent to the console are addressed to one of the two slots (respectively ”mem. card 0” and ”controller 0” / ”mem. card 1” and ”controller 1”). Afterwards, both accessories are assigned an ID allowing the console to distinguish the two types of devices (memory card or controller) and to concentrate operations on the requested one.

This similarity between the two accessories, with completely different uses, caused significant bugs for developers at the time. Dave Baggett, working on Crash Bandicoot, discovered that a hardware interference between the controller and memory card, caused data corruption, leading Sony to acknowledge the problem. Sources: [80].

3.6.3 Rear Ports

As mentioned in chapter 3.2.5, the SCPH-1000 model has two I/O ports, a Serial and a Parallel one. These ports were then removed in later models due to lack of use and because they could be used to bypass the copy protection system.

Part II

Development and Programming Fundamentals on PS1

Chapter 4

Programming in MIPS Assembly

This section introduces the fundamentals of MIPS Assembly programming in relation to the PSX. Topics such as memory manipulation, inserting values into registers, and familiarization with basic instructions will be covered. This language is essential in this context as it allows direct communication with the PlayStation CPU, executing low-level operations that directly interact with the hardware.

It is worth emphasizing that the overview will be general in nature, as the C language will primarily be used for the development of demos. Nonetheless, addressing this topic is considered important.

4.1 The 32 General-Purpose Registers

The R3000 SoC registers can be divided into general-purpose registers and ”special” registers.

One of these is the Program Counter (PC), a special processor register that contains the address of the next instruction to be executed. In this chip, it is automatically incremented during code execution, pointing to the next instruction in the sequence.

The general-purpose registers of this processor consist of 32 registers, each 32 bits wide, with a specific use. When developing assembly code, these registers are not simply called R1, R2, etc., but are identified by more understandable code names or aliases for programmers. They are used to store data, addresses, and intermediate results during the execution of a program. They are essential for performing arithmetic and logical operations, managing the passing of parameters in subroutines, and manipulating memory addresses. The use of general-purpose registers allows the processor to quickly access the necessary information without having to interact with the main memory, thus improving system performance. Sources: [72].

Register #	Alias	Description
0	Zero	Constant zero (fixed value)
1	AT	Reserved for the assembler
2	V0	Function return value
3	V1	Return value (for double type)
4	A0	First function argument
5	A1	Second function argument
6	A2	Third function argument
7	A3	Fourth function argument
8	T0	Temporary value
9	T1	Temporary value
10	T2	Temporary value
11	T3	Temporary value
12	T4	Temporary value
13	T5	Temporary value
14	T6	Temporary value
15	T7	Temporary value
16	S0	Saved register (preserved)
17	S1	Saved register (preserved)
18	S2	Saved register (preserved)
19	S3	Saved register (preserved)
20	S4	Saved register (preserved)
21	S5	Saved register (preserved)
22	S6	Saved register (preserved)
23	S7	Saved register (preserved)
24	T8	Temporary value
25	T9	Temporary value
26	K0	Reserved for the kernel
27	K1	Reserved for the kernel
28	GP	Global pointer
29	SP	Stack pointer
30	FP	Frame pointer
31	RA	Return address

Table 4.1: General-purpose registers of the R3000 SoC

Zero Register This register constantly contains the value 0 and cannot be modified. It is used to simplify operations that require this value, avoiding the need to explicitly load the value into another register.

AT Register The AT register is used as a workspace by the assembler and is reserved (thus cannot be used by the programmer or the C compiler).

SP and FP Registers The R3000 processor does not have the concept of a stack. Therefore, the compiler creates its own stack by storing a pointer in the SP register. To make function frames (areas for automatic/local variables and workspace) efficient, the initial address of the

frame is stored in the FP register, calculated from SP. During module activation, FP is set to the same value as SP.

A0, A1, A2, A3 Registers Reserved for the arguments of C language functions. If a function has up to four arguments, they are stored in registers A0-A3. If there are five or more arguments, they are stored on the stack; however, the first four arguments are still passed through the registers, while the stack space for them remains unused.

RA Register In this processor, calls to subroutines are managed using jump instructions that save the return address in a special register, the RA register. This allows the program to know where to return after executing a subroutine.

GP Register For efficient access with 16-bit offsets, the compiler groups variables up to 64 KB in an area called the ”bss section” and stores the central address in the GP register. This allows accessing variables with short instructions; the GP remains constant in the module.

4.2 Elementary MIPS Instructions

In assembly code, instructions have a fixed length of 32 bits and define the operations that the CPU must perform. This section will analyze the main operations.
Sources: [74] [81].

4.2.1 Data Transfer Instructions

These instructions are fundamental for managing data in registers. Keep in mind that each register has a size of 4 bytes, or 32 bits.

Instruction	Example	Description
`li`	`li $t0, 10`	Load an immediate value (a constant) into the register ($t0 = 10).
`la`	`la $t1, var`	Load the address of a variable ($t1 = address of `var`).
`lui`	`lui $t2, 0x1F80`	Load the immediate value 0x1F80 into the upper 16 bits of register $t2, setting the lower 16 bits to zero ($t2 = 0x1F800000).
`move`	`move $t3, $t4`	Copy the value from one register to another ($t3 = $t4).

Table 4.2: Instructions `li`, `la`, `lui` and `move` with examples and descriptions.

4.2.2 Load Instructions

In the context of data loading, these can be managed in different sizes:

- **BYTE**: 8 bits
- **HALF**: 16 bits
- **WORD**: 32 bits
- **DWORD**: 64 bits

If one wanted to make a comparison between the data formats of MIPS and the data types of the C language, the following correspondences could be highlighted:

```
1       int main(){
2           char        v1; // 1 byte OR 8 bits
3           short       v2; // 2 bytes OR 16 bits
4           int         v3; // 4 bytes OR 32 bits
5           long        v4; // 4 bytes OR 32 bits
6           long long   v5; // 8 bytes OR 64 bits
7       }
```

Load instructions read data from memory and copy them into a register. These operations serve to transfer information from the main memory (RAM) to the CPU registers to allow processing.

It is not possible to load a DWORD as the PSX SoC uses a 32-bit architecture.

Instruction	Example	Description
`lw`	`lw $t0, 0($t1)`	Load a word (32 bit) from memory at address `$t1 + 0` into register `$t0`.
`lh`	`lh $t0, 4($t1)`	Load a halfword (16 bit, sign-extended) from memory at address `$t1 + 4` into register `$t0`.
`lb`	`lb $t0, 8($t1)`	Load a byte (8 bit, sign-extended) from memory at address `$t1 + 8` into register `$t0`.
`lhu`	`lhu $t0, 4($t1)`	Load an unsigned halfword (16 bit) from memory at address `$t1 + 4` into register `$t0`.
`lbu`	`lbu $t0, 8($t1)`	Load an unsigned byte (8 bit) from memory at address `$t1 + 8` into register `$t0`.

Table 4.3: Load instructions: `lw`, `lh`, `lb`, `lhu`, `lbu` with examples and descriptions.

4.2.3 Store Instructions

Store instructions transfer data from a register to a memory address. These operations are used to transfer calculated results from the CPU registers to the main memory (RAM), where they can be stored or used by other parts of the program.

Instruction	Example	Description
`sw`	`sw $t0, 0($t1)`	Store a word (32 bit) from register `$t0` to memory address `$t1 + 0`.
`sh`	`sh $t0, 4($t1)`	Store a halfword (16 bit) from register `$t0` to memory address `$t1 + 4`.
`sb`	`sb $t0, 8($t1)`	Store a byte (8 bit) from register `$t0` to memory address `$t1 + 8`.

Table 4.4: Store instructions: `sw`, `sh`, `sb` with examples and descriptions.

4.2.4 Differences between Load and Store

Type	Purpose	Example	Direction
Load	Read data from memory to registers	`lw $t0, 0($t1)`	Memory → Register
Store	Write data from registers to memory	`sw $t0, 0($t1)`	Register → Memory

Table 4.5: Summary of the difference between Load and Store operations.

4.2.5 Jump Instructions

These instructions alter the program's execution flow, allowing a jump to another position in the code. They are used to implement control structures such as loops, function calls, and returns from subroutines.

Instruction	Example	Description
`j`	`j target`	Unconditionally jump to the label `target`.
`jal`	`jal target`	Jump to the label `target` and save the return address in the `$ra` register.
`jr`	`jr $t0`	Jump to the address stored in the `$t0` register.
`jalr`	`jalr $t1, $t0`	Jump to the address in `$t0` and save the return address in `$t1`.

Table 4.6: Jump instructions in MIPS with examples and descriptions.

4.2.6 Branch Instructions

These instructions are used to implement control structures such as conditional blocks (if) and loops (for, while).

Instruction	Example	Description
`beq`	`beq $t0, $t1, label`	Jump to `label` if `$t0` is equal to `$t1`.
`bne`	`bne $t0, $t1, label`	Jump to `label` if `$t0` is different from `$t1`.
`blez`	`blez $t0, label`	Jump to `label` if `$t0` is less than or equal to zero (signed).
`bgez`	`bgez $t0, label`	Jump to `label` if `$t0` is greater than or equal to zero (signed).
`bltz`	`bltz $t0, label`	Jump to `label` if `$t0` is less than zero (signed).
`bgtz`	`bgtz $t0, label`	Jump to `label` if `$t0` is greater than zero (signed).
`blt`	`blt $t0, $t1, label`	Jump to `label` if `$t0` is less than `$t1` (signed).
`ble`	`ble $t0, $t1, label`	Jump to `label` if `$t0` is less than or equal to `$t1` (signed).

Table 4.7: Branch instructions in MIPS with examples and descriptions.

4.2.7 The NOP Concept

In software written for MIPS architectures, it is common practice to insert a NOP (No Operation) instruction immediately after a Branch or Jump. This is due to the internal functioning of the MIPS processor pipeline, which divides instruction execution into parallel phases.

As will be discussed in chapter 4.5, the processor pipeline requires time to determine the exact destination address of a branch or jump. During this time, the CPU might incorrectly attempt to execute the subsequent instructions before knowing where to actually jump.

This behavior is known as the ”branch delay slot”.

The NOP instruction, which has no practical effect, is inserted to occupy the branch delay slot, preventing unintended instructions from being executed. Alternatively, the slot can be used to insert a useful instruction that does not interfere with the jump.

```
1 beq $t0, $t1, branch_label  // Branch to branch_label if $t0 == $t1
2 nop                         // NOP instruction for delay slot
3 // Other instructions
4 branch_label:
5     // Code executed if the condition is true
```

Codice 4.1: Example with conditional branch instruction

```
j target_label  ; Salta incondizionatamente a target_label
nop             ; Istruzione NOP per gestire il delay slot
; Altre istruzioni
target_label:
    ; Codice eseguito dopo il salto
```

Codice 4.2: Example with jump instruction

4.2.8 Arithmetical Instructions

Instruction	Example	Description
add	`add $t0, $t1, $t2`	Adds the values in registers $t1 and $t2, storing the result in $t0. Generates an exception in case of overflow.
addu	`addu $t0, $t1, $t2`	Adds the values in registers $t1 and $t2 without generating exceptions for overflow, storing the result in $t0.
addi	`addi $t0, $t1, 10`	Adds the immediate value 10 to the content of $t1, storing the result in $t0. Generates an exception in case of overflow.
addiu	`addiu $t0, $t1, 10`	Adds the immediate value 10 to the content of $t1 without generating exceptions for overflow, storing the result in $t0.
sub	`sub $t0, $t1, $t2`	Subtracts the value in $t2 from that in $t1, storing the result in $t0. Generates an exception in case of overflow.
subu	`subu $t0, $t1, $t2`	Subtracts the value in $t2 from that in $t1 without generating exceptions for overflow, storing the result in $t0.
mult	`mult $t1, $t2`	Multiplies the values in registers $t1 and $t2, storing the 64-bit result in the special registers HI and LO.
multu	`multu $t1, $t2`	Multiplies the unsigned values in registers $t1 and $t2, storing the 64-bit result in the special registers HI and LO.
div	`div $t1, $t2`	Divides the value in $t1 by that in $t2, storing the quotient in LO and the remainder in HI.
divu	`divu $t1, $t2`	Divides the unsigned value in $t1 by that in $t2, storing the quotient in LO and the remainder in HI.

Table 4.8: Arithmetic instructions in MIPS with examples and descriptions.

The multiplication and division operations in MIPS save their results in two special registers, HI and LO. To access the values contained in these registers, the mfhi and mflo instructions can be used.

```
; Example of using HI and LO registers
li $t0, 10          ; Load 10 into $t0
li $t1, 20          ; Load 20 into $t1

mult $t0, $t1       ; Multiply $t0 and $t1; result goes to HI and LO
mflo $t2            ; Copy the value of LO into $t2
mfhi $t3            ; Copy the value of HI into $t3

div $t0, $t1        ; Divide $t0 by $t1; quotient in LO, remainder in HI
mflo $t4            ; Copy the quotient (LO) into $t4
mfhi $t5            ; Copy the remainder (HI) into $t5
```

Codice 4.3: Example of using mfhi and mflo operations

4.3 Console Emulation

This section will use two tools to execute assembly code on a PlayStation emulator. The first tool is ARMIPS, an assembler designed to generate executable binary code for the PSX [82]. The second is a PlayStation emulator, PCSX-Redux [83].

4.3.1 Programming Example

Below is a simple program, written in MIPS Assembly language, for calculating the factorial of a number. The development of this software is based on the information provided in the previous chapters.

The following code will be saved in .s format, a commonly used extension for assembly source files.

```
.psx ; Indica all assembler l'architettura di riferimento
.create "Fattoriale.bin", 0x80010000
.org 0x80010000  ; Creazione di un file.bin in questo preciso punto
   d'ingresso
Main:
li $t0, 6; num = 6 (numero per cui calcolare il fattoriale)
li $t3, 1; temp = 1 (valore temporaneo)
li $t4, 1; sum = 1 (somma intermedia)
li $t1, 1; i = 1 (contatore esterno)
OuterLoop:
ble $t0, $t1, EndLoop; Se i > num, esce dal ciclo esterno
nop
move $t4, $zero; sum = 0 (reset della somma)
move $t2, $zero; j = 0 (reset del contatore interno)
InnerLoop:
```

```
blt $t1, $t2, EndInnerLoop ; Se j >= i, esce dal ciclo interno
nop
add $t4, $t4, $t3     ; sum += temp (accumula temp in sum)
addi $t2, $t2, 1      ; j++
j InnerLoop     ; Torna all inizio del ciclo interno
nop
    EndInnerLoop:
move $t3, $t4   ; temp = sum (aggiorna temp)
addi $t1, $t1, 1      ; i++
j OuterLoop     ; Torna all'inizio del ciclo esterno
nop
    EndLoop:
move $v0, $t4   ; Salva il risultato finale in $v0
InfiniteLoop:
j InfiniteLoop ; Ciclo infinito per terminare il programma in modo sicuro
nop
.close
```

Codice 4.4: Calculating the factorial of a number in MIPS Assembly

4.3.2 The PS-EXE Format

Once the code is assembled, a .bin file will be created, representing raw binary data.

Before testing the code on an emulator or a PlayStation console, conversion to a compatible format is necessary. The PlayStation BIOS accepts files in 'PSX-EXE' format. This format represents a 32-bit MIPS executable specific to the PS1 and includes both the code and the data necessary for execution. Sources: [84].

To generate a file in PSX-EXE format, a Python script can be used [85] that converts, for example, the Fattoriale.bin file into the corresponding Fattoriale.ps-exe.

Loading the .PS-EXE file into the emulator displays the following information:

- **Red box**: General registers.
- **Yellow box**: Memory position where the program was loaded.
- **Green box**: Program code. The yellow arrow highlights the starting point of execution, while the red dot represents a breakpoint, inserted to make the result of the operations readable after execution.

Once the program is executed, calculating the factorial of the number 6, the general register $v0 will contain the hexadecimal value 000002d0, corresponding to the decimal value 720, confirming that the calculation was performed correctly.

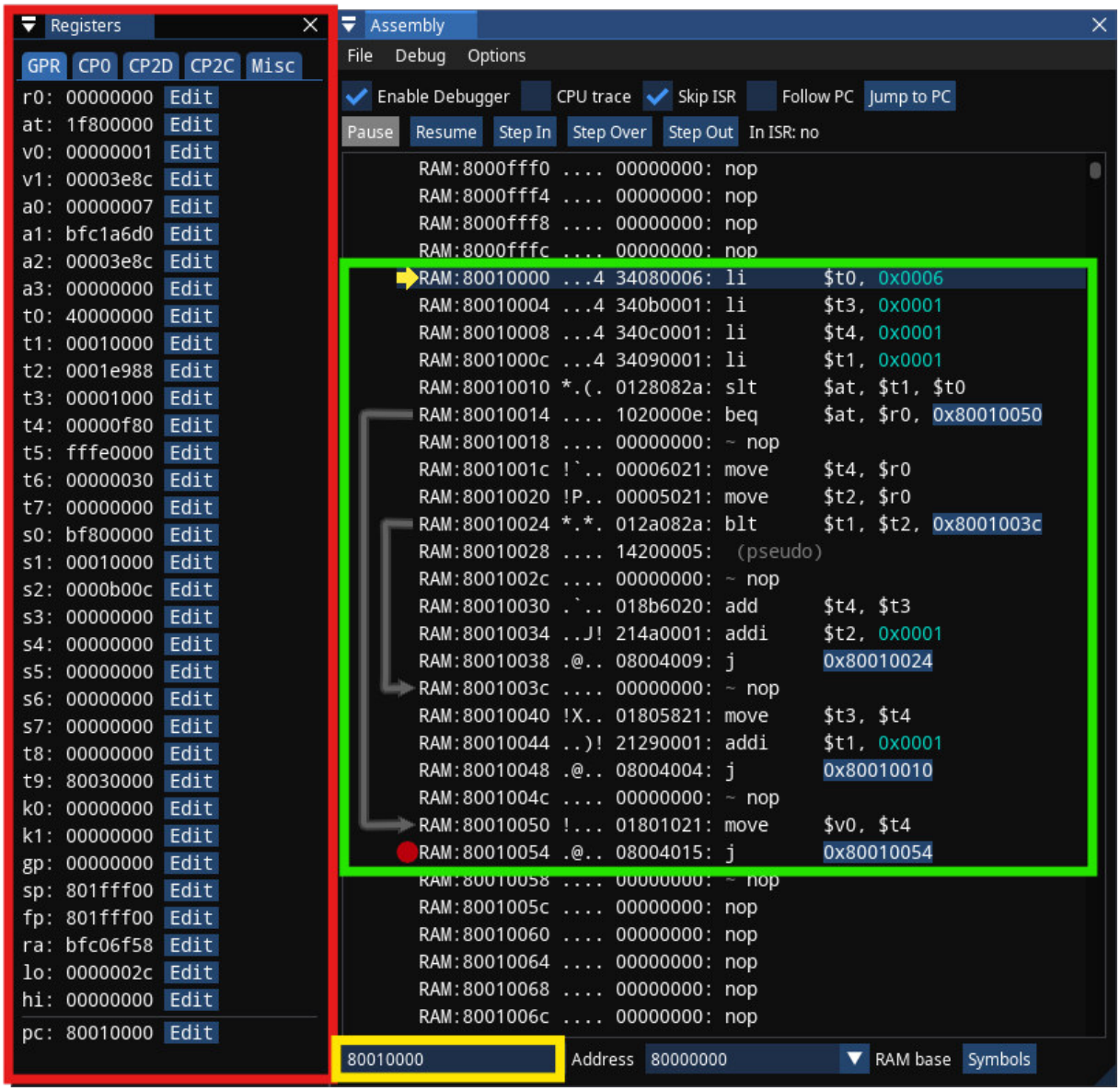


Figure 4.1: Screen taken from the PSX emulator before execution

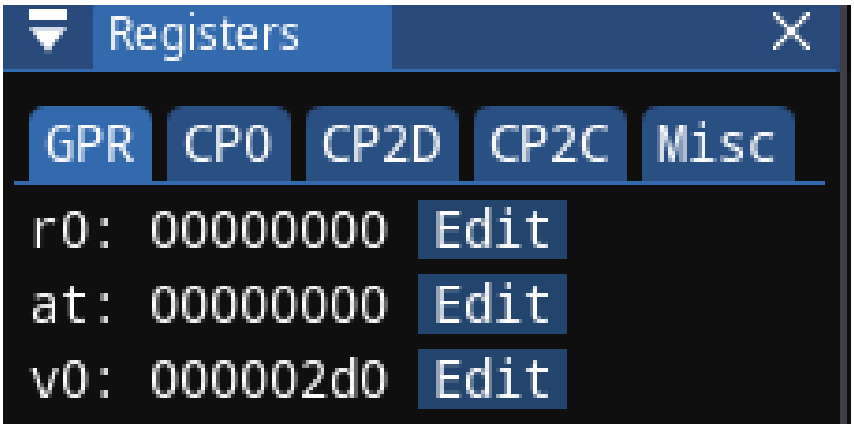


Figure 4.2: Screen taken from the PSX emulator after execution

4.3.3 The Pseudo-Instruction Concept

In the code displayed on the emulator (figure 4.1), some differences can be seen compared to the code shown in chapter 4.2.1. This discrepancy is due to the use of pseudo-instructions in the MIPS Assembly language. The assembler translates these pseudo-instructions into one or more actual instructions from the MIPS instruction set. It is important to note that, although pseudo-instructions simplify programming, they may result in the execution of more machine instructions than a single instruction, affecting the software's efficiency. Sources: [86].

4.3.4 The Sub-Routine Concept

As described in the paragraph dedicated to "Jump Instructions", subroutines can be used to improve code structure, making it more readable and flexible. In assembly language, subroutines are similar to functions in high-level programming languages. Below is a version of the factorial calculation program that uses subroutines. Sources: [87].

```
1  .psx    // Specify target architecture (PlayStation)
2  .create "factorial.bin", 0x80010000 // Create a binary file with entry
       point at 0x80010000
3
4  .org 0x80010000 // Set code origin to address 0x80010000
5
6  Main:
7    li $a0 , 0x0006   // Load number 6 into register $a0 (number to
         calculate factorial for)
8    jal Factorial     // Call the Factorial subroutine and save return
         address in $ra
9    nop               // NOP for jump delay slot
10
11 LoopForever:
12   j LoopForever     // Jump to LoopForever, creating an infinite loop
13   nop               // NOP for jump delay slot
14
15 Factorial:
16   li $t3 , 0x0001   // Initialize $t3 (temp) to 1
17   li $t4 , 0x0001   // Initialize $t4 (sum) to 1
18   li $t1 , 0x0001   // Initialize $t1 (i) to 1
19
20 OuterWhile:
21   ble $a0 , $t1 , EndOuterWhile // If $t1 (i) >= $a0 (NUM), exit outer loop
22   nop               // NOP for delay slot
23   move $t4 , $zero  // Set $t4 (sum) to 0
24   move $t2 , $zero  // Set $t2 (j) to 0
25
26 InnerWhile:
```

```
27    blt $t1, $t2, EndInnerWhile   // If $t2 (j) >= $t1 (i), exit inner loop
28    nop                // NOP for delay slot
29    add $t4, $t4, $t3 // Add $t3 (temp) to $t4 (sum)
30    addi $t2, $t2, 0x0001 // Increment $t2 (j) by 1
31    b InnerWhile         // Jump to start of inner loop
32    nop                  // NOP for delay slot
33
34  EndInnerWhile:
35    move $t3 , $t4       // Update $t3 (temp) with value of $t4 (sum)
36    addi $t1 , $t1 , 0x0001   // Increment $t1 (i) by 1
37    b OuterWhile         // Jump to start of outer loop
38    nop                  // NOP for delay slot
39
40  EndOuterWhile:
41    move $v0 , $t4       // Save result (factorial) in $v0
42    jr $ra               // Return to address saved in $ra (subroutine
          finished)
43
44  .close  // Close the binary file
```

Codice 4.5: Calculating the factorial using Subroutine

4.4 Handling Binary Data

4.4.1 Managing Negative Numbers

PlayStation, as a digital machine, operates using binary data represented by zeroes and ones. Data that are generally organized in bytes. As described in the basic operations of chapter 4.2, it is possible to handle data as either signed (with sign) or unsigned (without sign).

A byte always consists of 8 bits, but its interpretation can vary. For example, the binary number 01101101 corresponds to the value 109 in decimal as a positive value. To perform this conversion, 2^n is summed for each bit set to one, where n represents the bit position (starting from 0 for the least significant bit).

An unsigned byte (8 bits) represents values from 0 to 255, an unsigned halfword (16 bits) from 0 to 65,535, and an unsigned word (32 bits) from 0 to 4,294,967,295.

In the case of signed data, the most significant bit (the leftmost bit) is reserved to represent the sign. A value of 0 will indicate a positive number, while 1 indicates a negative number. However, this representation modifies the previous logic of calculation, and that is why the concept of ”two’s complement” is adopted. Sources: [88].

With ”two’s complement”, the leftmost bit represents a negative value equal to $-2^{(n-1)}$, where n

is the total number of bits. The rest of the bits contribute to the total value as positive powers of 2. Thus a signed byte can represent values from -128 to +127. A signed halfword can represent values from -32,768 to +32,767. A signed word can represent values from -2,147,483,648 to +2,147,483,647.

Some fundamental properties of this representation include the fact that zero is always indicated with all bits set to 0, the most significant bit determines the sign of the number, and the operations of addition and subtraction work identically for both signed and unsigned numbers.

4.4.2 Sign Extension

This procedure ensures that the value of the number remains consistent regardless of the data size. It is emphasized that the programmer does not have to manage these operations manually, as the MIPS ISA automatically performs them during operations that require this extension.

In the context of signed numbers, a 32-bit architecture does not limit itself to handling single bytes, but also includes the already mentioned halfword (16 bit) and word (32 bit). When a signed number is extended from 8 bits to 16 bits, it is essential to preserve its sign: positive numbers are extended by filling the most significant bits (to the left) with zeroes, while for negative numbers, the most significant bits are filled with ones. Sources: [81].

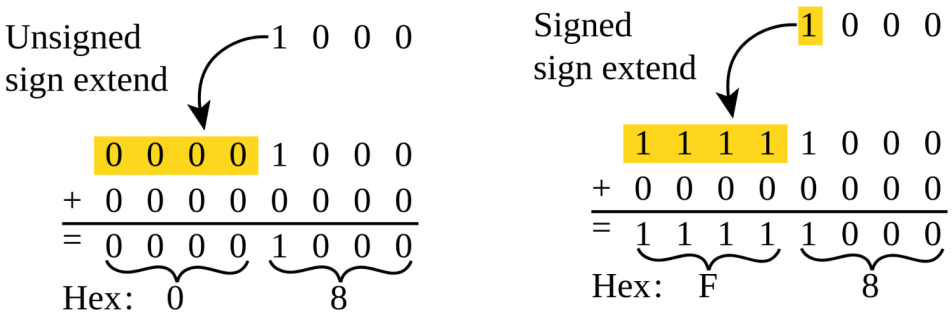


Figure 4.3: Examples of sign extension
[36]

4.4.3 Logical Operations

Logical operations are useful tools for manipulating and analyzing binary data. The most commonly used operations in this context are the following. Sources: [81][74].

Logical AND

The AND operation returns a true result only if both inputs are true. This property lends itself particularly well to the concept of ”masking”, an operation that allows extracting specific information by applying a binary mask.

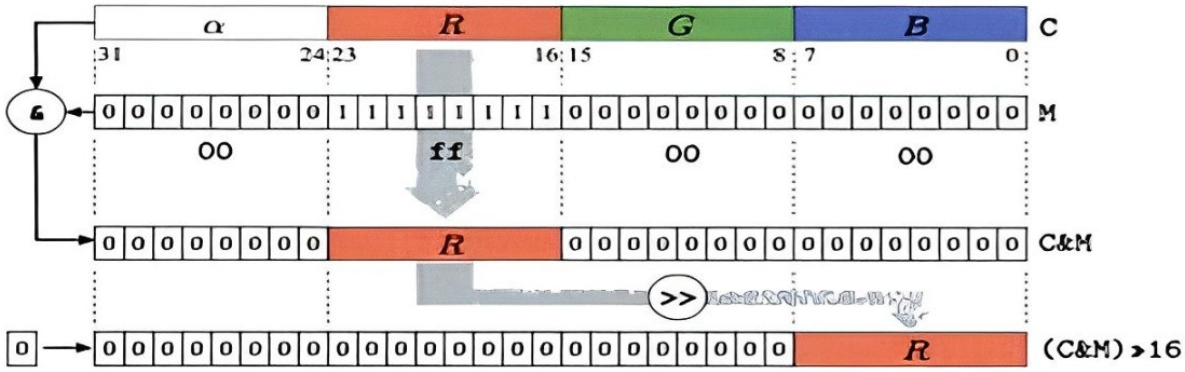


Figure 4.4: Example of ”masking”
[37]

For example, to isolate the values related to the RGB channels (ALPHA, RED, GREEN, BLUE), each represented by 8 bits, a mask can be applied that sets to zero the bits to be hidden and to one those of interest.

Logical OR

The OR operation returns a true result if at least one of the two inputs is true.

For example, OR can be used to activate specific bits in a binary sequence, keeping unchanged those already set. This is particularly relevant when flags in a binary mask need to be set without altering their current state.

Exclusive OR (XOR)

The XOR operation is similar to the previous OR, with the difference that it returns a false result when both inputs are true. This property is useful in applications that require verifying differences between two binary values.

Instruction	Example	Description
and	`and $t0, $t1, $t2`	Performs a bitwise AND operation between the registers $t1 and $t2, storing the result in $t0.
andi	`andi $t0, $t1, 0xFF`	Performs a bitwise AND operation between the register $t1 and the immediate value 0xFF, saving the result in $t0.
or	`or $t0, $t1, $t2`	Performs a bitwise OR operation between the registers $t1 and $t2, storing the result in $t0.
ori	`ori $t0, $t1, 0xFF`	Performs a bitwise OR operation between the register $t1 and the immediate value 0xFF, saving the result in $t0.
xor	`xor $t0, $t1, $t2`	Performs a bitwise XOR operation between the registers $t1 and $t2, storing the result in $t0.
xori	`xori $t0, $t1, 0xFF`	Performs a bitwise XOR operation between the register $t1 and the immediate value 0xFF, saving the result in $t0.

Table 4.9: Logical instructions AND, OR, and XOR with examples and descriptions.

4.4.4 The Bit-Shifting Concept

Bit shifting allows moving the bits of a value to the right or left, or rotating them in the same manner.

This concept is applied in operations such as multiplications and divisions, where a left shift is equivalent to multiplying a number by a power of 2, while a right shift corresponds to dividing it by a power of 2.

The shift operation is particularly advantageous because it is significantly faster and more efficient than traditional multiplications or divisions. This technique will be further explored in the chapter dedicated to ”Fixed Point Math”.

Shift Rotation

In the case of a right shift, the least significant bit (the rightmost one) is lost, while a zero is inserted in the most significant position (to the left). Conversely, with a left shift, the most significant bit is eliminated, and a zero is added as the new least significant bit.

During a rotation, the bits are not lost. The bit removed from one end is reinserted at the opposite end, creating a rotation effect. In the MIPS R3000 environment, the concept of rotation can be simulated, but there is no dedicated instruction for it, unlike shift operations. Sources: [74] [81].

Preserving the Sign

When shifting signed numbers, the bit representing the sign can be lost. To address this issue, arithmetic shifts are used, which preserve the sign by keeping the most significant bit unchanged during the shift. However, this consideration is only valid for right shifts. In the case of left shifts, both types behave the same way, as the MSB is simply shifted out of range without the need to preserve the sign. It should also be noted that there is no arithmetic version of rotation, as this operation implies reusing the bits lost during the operation, making the distinction between signed or unsigned values irrelevant. Sources: [89].

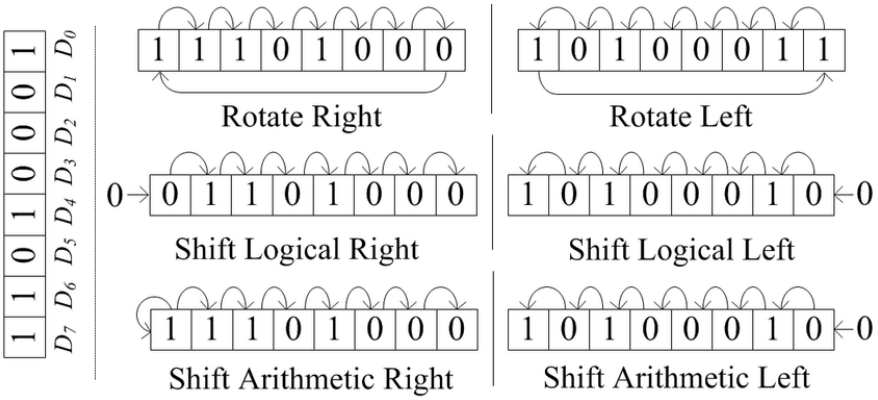


Figure 4.5: Examples of bit shifting
[38]

Instruction	Example	Description
`sll`	`sll $t0, $t1, 2`	Performs a logical left shift on the bits of $t1 by 2 positions, storing the result in $t0.
`srl`	`srl $t0, $t1, 2`	Performs a logical right shift on the bits of $t1 by 2 positions, filling the most significant bits with zeroes and storing the result in $t0.
`sra`	`sra $t0, $t1, 2`	Performs an arithmetic right shift on the bits of $t1 by 2 positions, preserving the sign, and stores the result in $t0.
`sllv`	`sllv $t0, $t1, $t2`	Performs a logical left shift on the bits of $t1 by a number of positions specified by $t2, storing the result in $t0.
`srlv`	`srlv $t0, $t1, $t2`	Performs a logical right shift on the bits of $t1 by a number of positions specified by $t2, filling the most significant bits with zeroes and storing the result in $t0.
`srav`	`srav $t0, $t1, $t2`	Performs an arithmetic right shift on the bits of $t1 by a number of positions specified by $t2, preserving the sign, and stores the result in $t0.

Table 4.10: Shift and rotation instructions in MIPS with examples and descriptions.

4.5 In-depth Study of the MIPS Pipeline

As mentioned in chapters 3.3.1 and 4.2.7, this chapter will explore the functioning of the RISC MIPS R3000 processor, with a particular focus on the five-stage pipeline.
Sources: [90][91][92].

4.5.1 MIPS Pipeline Structure

In the processors of the previous generation of consoles, lacking a pipeline, instruction execution occurred sequentially, completing an instruction entirely before starting a new one.

The introduction of the pipeline in RISC processors, such as the MIPS R3000, fragments operations into smaller and overlapping stages, improving execution efficiency. Each instruction goes through five main stages: fetch, decode, execute, memory, and write.

This design allows for overlapping stages, enabling the processor to execute multiple instructions simultaneously.

In theory, a processor with a pipeline can approach a performance of one instruction per clock cycle (1 CPI), even though each instruction typically requires three to five cycles.

This approach significantly increases the processor's efficiency but can introduce synchronization problems if not managed correctly.

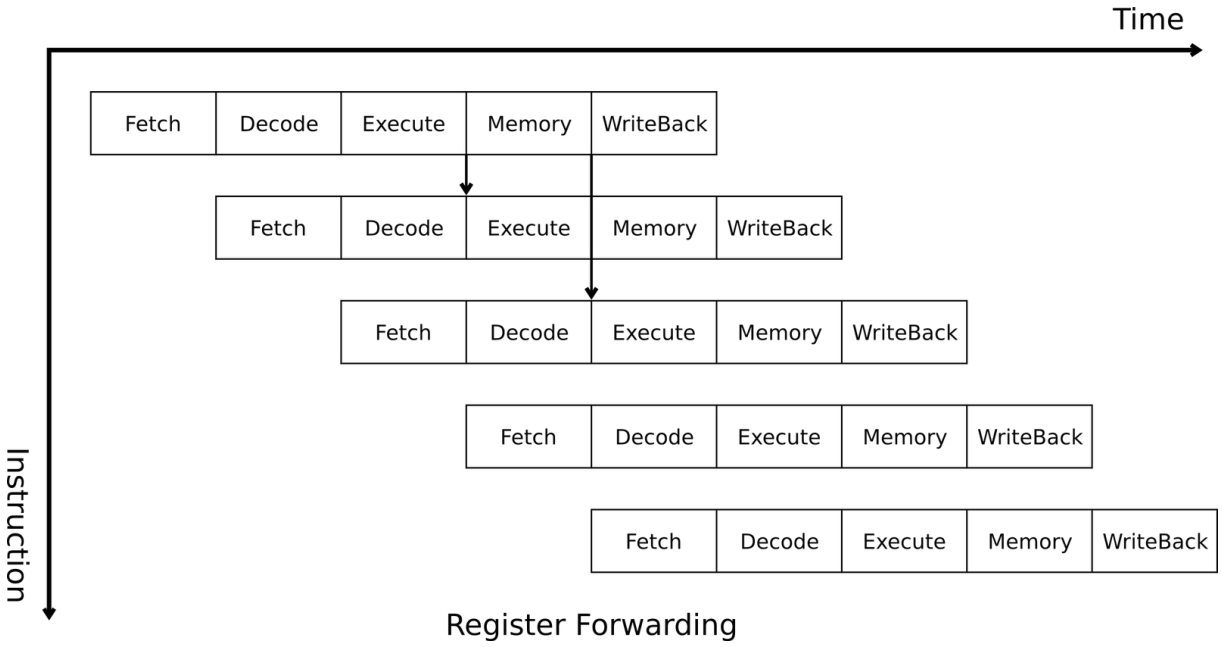


Figure 4.6: Pipeline representation
[39]

4.5.2 Limits of the MIPS Pipeline

- **Flow changes**: Branch or jump instructions can cause problems because the immediately following instruction, already in the pipeline, might be incorrectly executed before the jump is actually made.
- **Load delay**: When loading a value from memory with instructions like lw (load word), there might be an attempt to use the loaded value before it is fully available. This problem does not occur with operations like li (load immediate), which happen very quickly.

4.5.3 Managing Delay Slots

To solve these problems, the MIPS pipeline introduces the concept of ”delay slot”, a solution adopted by the first RISC processors. In simple terms, operations that serve to ”consume” the clock cycles necessary to complete the previous operation.

A common example is the use of the NOP (No Operation) instruction, explained in chapter 4.2.7, which occupies the delay slot without performing any operation.

This instruction is a pseudo-instruction (see chapter 4.3.3) in the MIPS instruction set, equivalent to an instruction that performs a logical left shift of 0 bits on the $r0 register.

Since $r0 is fixed at the value 0 and cannot be modified, the instruction has no effects, other than consuming a clock cycle.

4.5.4 Optimizing Delay Slots

Although using NOP is the simplest solution, it is possible to fill delay slots with useful instructions that do not depend on the previous operation, thus improving software efficiency.

Modern compilers, like those for the C language, automatically manage these optimizations, relieving the programmer of this responsibility.

4.6 In-depth Study of the RISC Processor

In the previous chapters, the importance of the RISC architecture has been emphasized. But what do the terms Reduced Instruction Set (RISC) and Complex Instruction Set (CISC) really mean, and how does RISC differ from traditional CISC processors?

4.6.1 Historical Evolution and Context

Until the fifth generation of consoles, such as NES, Sega Genesis, Super Nintendo (SNES), and Atari 2600, the processors used were mainly based on CISC architecture. These processors,

although flexible, often required many clock cycles to complete a single instruction. Towards the end of the 1980s, a shift in design philosophy occurred with the introduction of RISC architectures. This technological revolution led to the development of several families of processors based on RISC. Among the major contributions of the time:

- **Acorn Computers** developed ARM processors.
- **Sun Microsystems** introduced the SPARC family.
- **Apple, IBM, and Motorola** created PowerPC processors.
- **Silicon Graphics** designed MIPS processors, mainly used in workstations dedicated to computer graphics.

MIPS technology not only became the heart of the PlayStation processor but was also adopted by competitors like Nintendo with its Nintendo 64 (Sources: [93]). This evolution marked a transition to faster and more efficient processors, specifically designed for multitasking and parallel processing. Sources: [73].

4.6.2 Differences between RISC and CISC

In CISC processors, instructions were designed to be more complex and powerful, often capable of performing elaborate operations in a single pass.

This approach somewhat simplified the writing of assembly code, making it more intuitive for programmers. However, the complexity of the instructions often required more clock cycles for their execution, significantly limiting the speed compared to the leaner and faster approach of RISC architectures.

RISC, on the other hand, adopts a minimalist approach with a reduced set of simple and fast instructions. Some of the main features of RISC include:

- Few and simple addressing modes.
- Fixed-size instructions.
- Separation between load and store operations.
- Aim to achieve 1 cycle per instruction (1 CPI).

Although CISC instructions were more ”user-friendly”, the advantage of RISC is that modern compilers, like those used for the PlayStation, automatically translate high-level code (e.g., in C language) into efficient instructions for the RISC architecture.

4.6.3 Advantages of RISC Architecture

The adoption of RISC offered numerous benefits:

- **Optimized instruction set**: RISC instructions, being simpler and more uniform, can be executed more quickly, reducing the complexity of the processor and improving overall performance.
- **More efficient pipeline**: The regular structure of RISC instructions facilitates pipeline implementation, allowing parallel execution of multiple instructions and greater throughput.
- **Flexibility in compilers**: The simplicity of RISC instructions allows compilers to generate more efficient code, optimizing the use of processor resources.

Such advantages have made RISC processors an ideal choice for devices requiring high performance and low latency, like game consoles. Sources: [94][95].

4.7 Graphics System

After exploring the functioning of the CPU, it is appropriate to examine that of the GPU. As discussed in chapters 3.2.3 and 3.4, this is a dedicated chip with access to two units of VRAM of 512 KB each, for a total of 1 MB.

The GPU has its own registers, similar to those of the CPU, but designed for different purposes, mainly related to graphics processing. It has access to the frame buffer, a portion of memory dedicated within the VRAM, used for rendering images. It is important to emphasize that the frame buffer is not directly accessible by the CPU, as it is not memory-mapped, which implies that the interaction between the CPU and GPU occurs exclusively through specific commands.

Communication between the GPU and CPU occurs through ”packets” containing graphics drawing and configuration instructions. These packets can be transmitted in two ways:

- **Word by Word**, packet after packet, single words of data are sent sequentially through the GPU's data port. A simple but less efficient approach.
- **Via DMA**. Data is transferred in bulk, using the DMA controller for faster and more efficient communication.

Introduced in chapter 3.3.1, DMA allows components like GPU, SPU, CD-ROM drive, and parallel port to access a dedicated DMA controller. This controller takes control of the main bus allowing very fast data transfer. However, during its operation, the CPU is unable to access the main bus, as it is entirely occupied by the ongoing transfer. Sources: [73] [72].

4.7.1 The Frame Buffer

The frame buffer is an essential component of the GPU, used to display frames on the screen and store graphic resources such as textures, fonts, and color tables. It has a size of 1024x512 pixels and supports color depths of 16 to 24 bits.

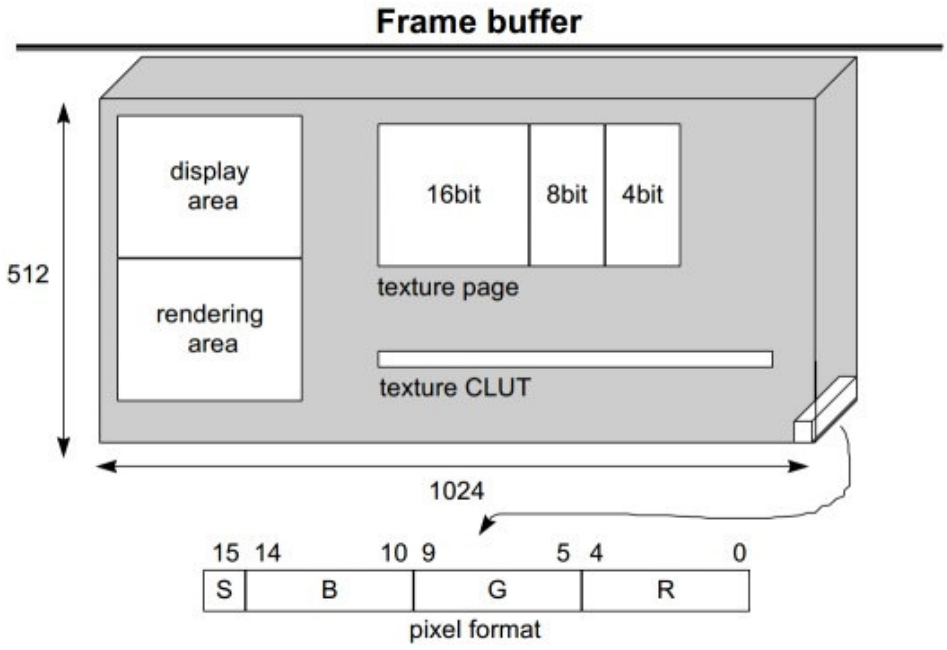


Figure 4.7: Graphic representation of the Frame Buffer
[40]

The coordinate system of the frame buffer is defined by a rectangular area with the upper left corner located at (0,0) and the lower right corner at (1023,511). Within this area, there is freedom to place different graphic resources. However, a portion of the frame buffer must be dedicated to the display area, a map that defines what will be visible on the screen.

Typically, two display areas are used through a concept called ”double buffering”, which helps reduce flickering and improve visual fluidity by alternating buffers during rendering. This concept will be further explored in the upcoming chapters. It is important to manage this memory efficiently to avoid overloads and ensure visual quality.

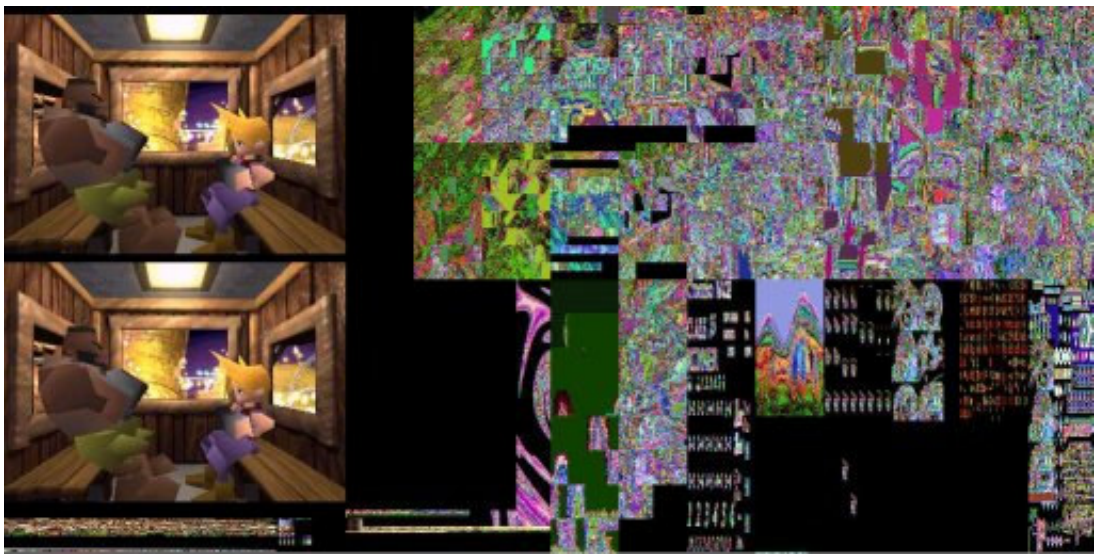


Figure 4.8: Frame Buffer during the playback of Final Fantasy 7
[41]

The PlayStation GPU uses a coordinate system encoded in 11-bit units, where the values for X and Y range from -1024 to 1023. Although the frame buffer size is limited to 1024x512 pixels, it is possible to define an arbitrary drawing origin by setting an offset in the coordinates.

This allows the GPU to draw only within defined rectangular areas, avoiding rendering in unwanted regions. The GPU's automatic clipping ensures that drawings are confined to the visible area, preventing visual errors. The concept of Clipping will be explored in a dedicated chapter.

4.7.2 Display Configuration Parameters

The PlayStation GPU supports different configurations for the display, each with specific resolutions and characteristics. These parameters determine how graphic content is rendered and displayed on the monitor. The main resolution configurations for the NTSC standard are listed in the following table:

Each mode has specific implications, varying according to the chosen resolution. For example, higher resolutions like 640x480 require more VRAM memory and GPU processing power, reducing the available space for other graphic resources, such as textures or color maps. On the other hand, modes with lower resolutions, such as 256x240, reduce the load on the GPU and allow for greater availability of resources for graphic assets.

Another relevant aspect is the choice between interlaced and non-interlaced modes. Interlaced modes allow for doubling the vertical resolution (e.g., from 240 to 480 lines), but may introduce a visible and annoying flickering effect. Non-interlaced modes offer more stable images, but with

Mode	Standard Resolution (NTSC)	Notes
0	256(H) x 240(V)	Non-interlaced
1	320 x 240	Non-interlaced
2	512 x 240	Non-interlaced
3	640 x 240	Non-interlaced
4	256 x 480	Interlaced
5	320 x 480	Interlaced
6	512 x 480	Interlaced
7	640 x 480	Interlaced
8	384 x 240	Non-interlaced
9	384 x 480	Interlaced

Table 4.11: Screen modes supported by the PlayStation GPU.

a lower vertical resolution. The chosen configuration depends on the type of graphic content and the performance requirements of the game. Sources: [96].

Most games operate at native resolutions ranging from $256x224$ in progressive mode to $640x480$ in interlaced mode. The most commonly used resolution is $320x240$. Lower resolutions offer better performance, but some games exploit higher resolutions for certain sections or for the entire gameplay. Sources: [97].

4.7.3 Color and Depth

PSX supports two color modes, each with specific characteristics: the 15-bit mode and the 24-bit mode. The choice of color mode depends on the balance between visual quality and GPU processing capability.

15-bit Mode

Allows the display of a maximum of 32,768 colors. Although the number of available colors is lower than in the 24-bit mode, the GPU performs color calculations using 24-bit precision.

Thanks to dithering, a technique that introduces visual noise to simulate smoother transitions between colors, it is possible to achieve a visual quality close to ”true color” (24 bit).

The samples of red, green, and blue follow the typical behavior of colors defined in the RGB space, while the STP (Special Transparency Processing) bit performs specific functions. Depending on the set transparency management mode, this bit determines whether pixels of a certain color are to be considered transparent.

When transparency is active, pixels with the STP bit enabled are interpreted as transparent. A special case is represented by black pixels (RGB 0,0,0), which PlayStation treats as transparent by default, unless the STP bit is disabled.

24-bit Mode

The 24-bit mode allows the display of up to 16,777,216 colors, offering a complete color representation. However, this mode has some significant limitations. The GPU can only display image data that has already been transferred to the frame buffer, as it is unable to perform direct drawing operations with 24-bit pixels.

Even in this mode, the coordinates and display positions in the frame buffer are still managed on a 16-bit basis. For example, a 24-bit image with a resolution of 640x480 is treated in the frame buffer as 960x480, to align with the 16-bit management. Additionally, the horizontal display sizes must be multiples of 8, which implies that the minimum screen size in this mode is 8x2 pixels.

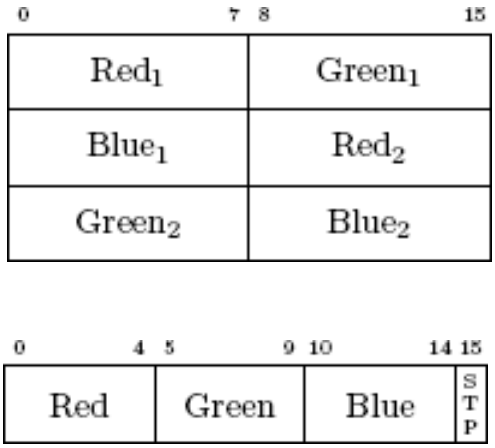
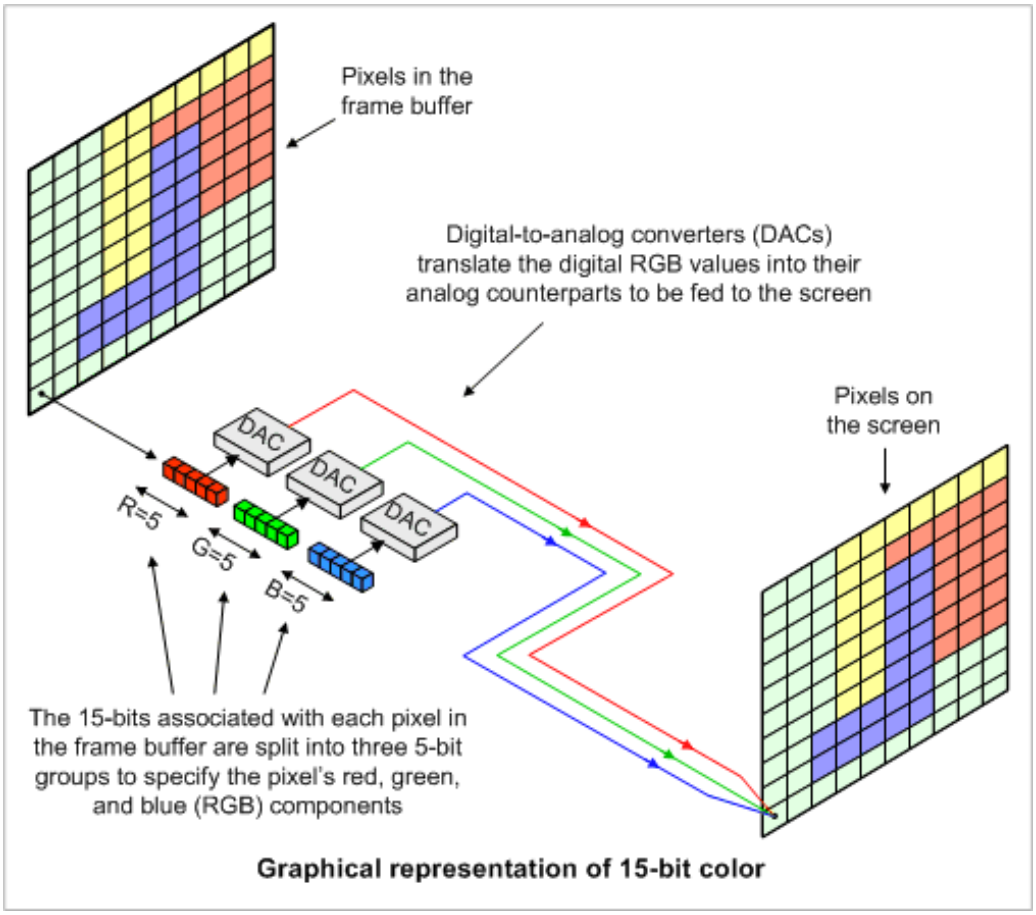


Figure 4.9: Above, the 24-bit mode. Below, the 16-bit mode.
[42]



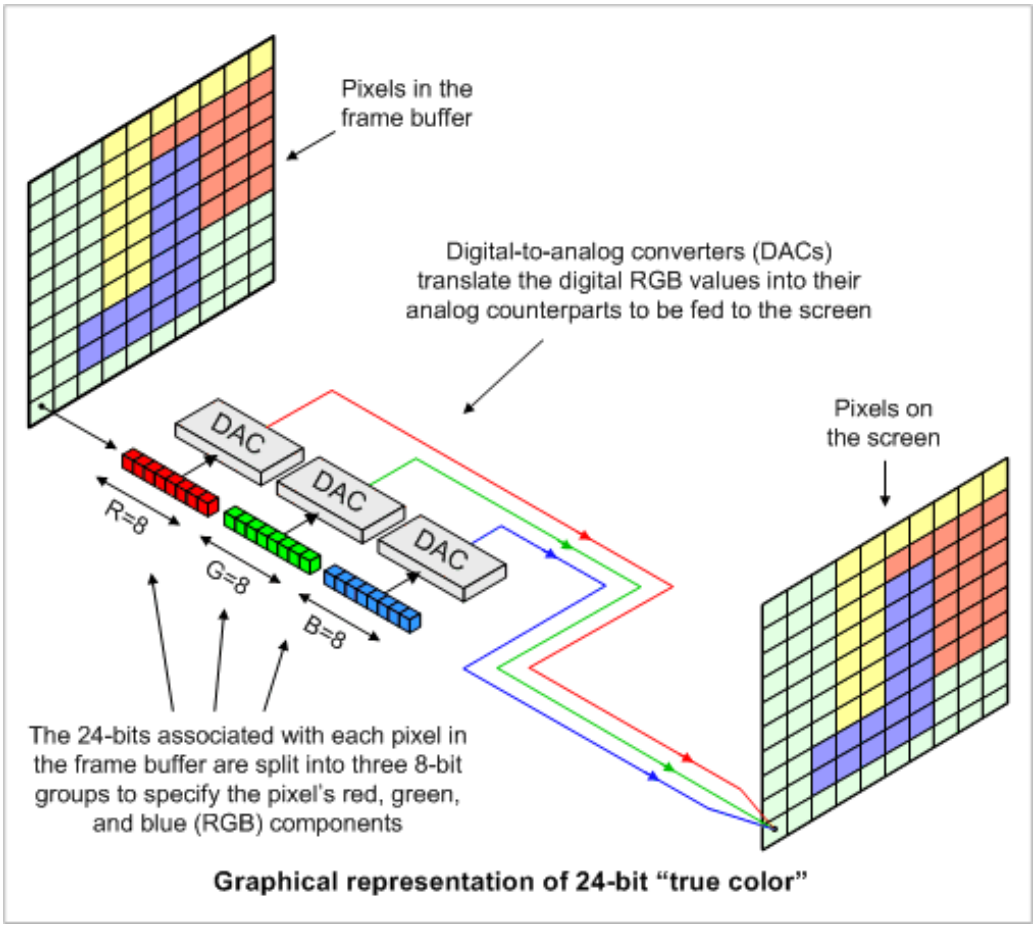


Figure 4.10: 15-bit and 24-bit color modes

[43]

4.7.4 PSX Primitives

Concept of primitive in computer graphics

In computer graphics, a primitive represents the basic geometric unit used to build complex models and images. Primitives include elements like points, lines, polygons, and are processed by the GPU to represent graphic scenes on the screen.

A polygon, in particular, is a geometric figure composed of closed lines that enclose an area. A triangle is considered the ideal primitive in 3D graphics because it possesses three fundamental properties: it is convex, simple (its sides do not intersect), and planar (its vertices are coplanar). This ensures greater stability and ease of calculation during rendering, eliminating ambiguities that could arise with more complex polygons. Sources: [98].

Primitives supported by the console

The PSX GPU is designed to draw a series of fundamental primitives, including:

- **Lines**: Segments that connect two points and are used to represent outlines or simple details.
- **Flat-Shaded Polygons**: Triangles and quadrilaterals with uniform coloring. It should be noted that quadrilaterals, in the case of the PSX, are internally managed as two adjacent triangles.
- **Gouraud-Shaded Polygons**: Polygons that use shading interpolated between the colors of the vertices to create a gradient effect.
- **Textured Polygons**: Polygons onto which textures are applied to add graphic details and enhance realism.
- **Sprite/Tile**: Two-dimensional images used mainly for moving objects or backgrounds.

Figure 4.11: Model: Crash Bandicoot (about 500 polygons).

Figure 4.12: Model: Leon, Resident Evil 2 (about 730 polygons).

4.7.5 Packet Management and Communication between CPU and GPU

Communication between the CPU and GPU inside the PSX occurs through the sending of structured packets, which contain commands and parameters necessary for managing rendering and display control. These packets travel along buses of specific sizes, following a precise hierarchy:

- The CPU, or DMA controller, sends 32-bit packets to the GPU.
- The GPU transfers 16-bit data to the VRAM.
- The VRAM, finally, sends 16-bit data to the Video Encoder, responsible for converting the content for display.

As illustrated in figure 3.24 (chapter 3.4.1), this structure allows for a clear separation of tasks between hardware components, ensuring an organized and optimized data flow.

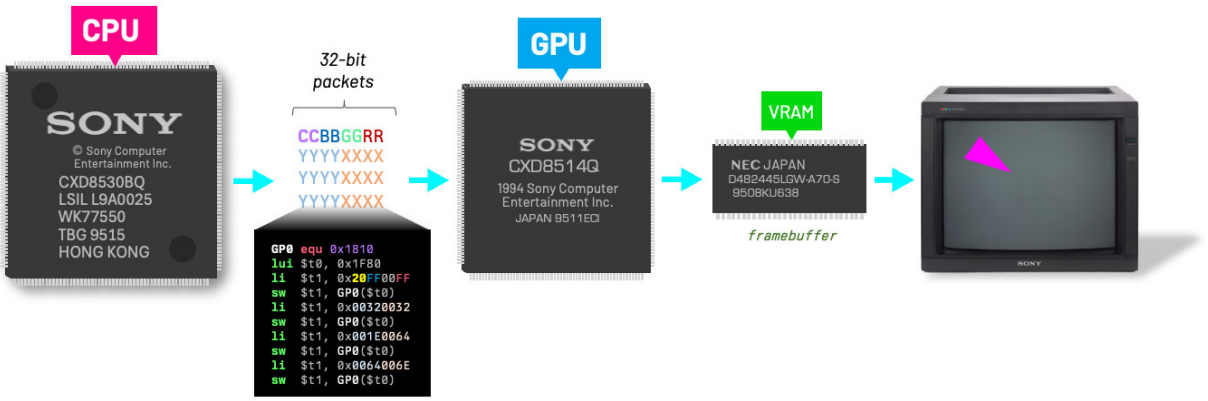


Figure 4.13: Communication between CPU, GPU, VRAM, and Video Encoder [44]

Structure and Destination of GPU Packets

The packets sent to the GPU are addressed to its two main control registers:

- **GP0**: used for rendering and accessing VRAM. This register manages commands related to drawing graphic primitives.
- **GP1**: used for display configuration and graphic environment control.

Sending Packets to the GP0 Register

The packets sent to the GP0 register serve to manage rendering and primarily work with graphic primitives, such as polygons, lines, and sprites. Each packet consists of a series of commands and parameters, each containing specific information.

For example, to draw a flat-shaded polygon, the following packets need to be sent:

- Packet 1: command and color (format CCBBGGRR).
- Packet 2: coordinates of the first vertex (format YYYYXXXX).
- Packet 3: coordinates of the second vertex.
- Packet 4: coordinates of the third vertex.
- Packet 5: coordinates of the fourth vertex (if necessary).

Sending Packets to the GP1 Register

The packets sent to the GP1 register are responsible for display configuration. Each packet contains an 8-bit command (MSB) and up to 24 bits of parameters (LSB). These packets follow the COMMAND+PARAM (CCPPPPPP) format, where the most significant bits represent the command and the remaining three bytes contain the parameters. Some examples of commands are:

- 0x00: Reset the GPU.
- 0x03: Enable the display.
- 0x08: Configure the video mode.

4.7.6 First Example of Basic Rendering in MIPS Assembly

This example represents a simple rendering process. The code illustrates how to configure the display, define the drawing area, and create graphic primitives such as triangles and quadrilaterals.

The instructions interact directly with the GP0 and GP1 registers to manage rendering and configure the graphic environment, showing how to send command packets to achieve graphic output on the screen.

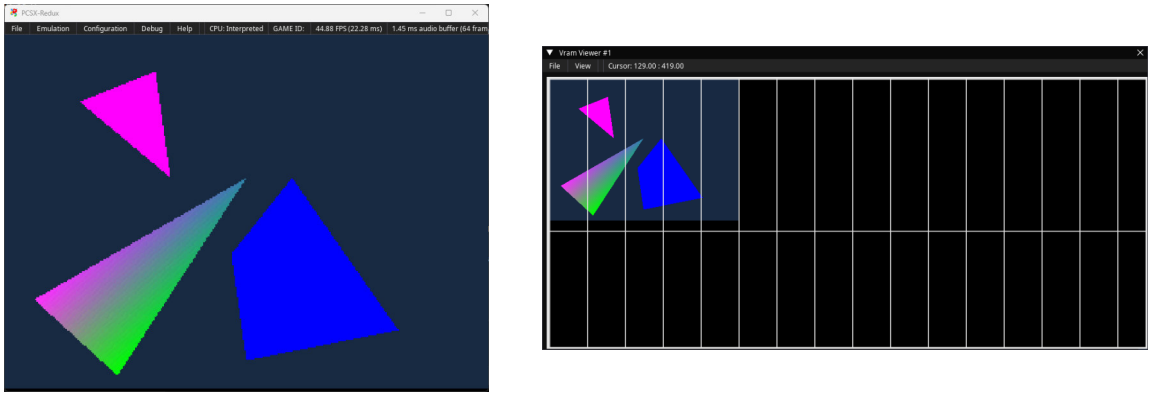


Figure 4.14: Screenshots taken from the emulator during the program execution. On the right, it is possible to see how the frame buffer is being occupied.

```
.psx
.create "First_Rendering_Example.bin", 0x80010000
.org 0x80010000

IO_BASE_ADDR equ 0x1F80      ; Base address for memory-mapped I/O ports
GP0 equ 0x1810               ; GP0 register (addr. $1F801810): Rendering
    data and VRAM access
GP1 equ 0x1814               ; GP1 register (addr. $1F801814): Display
    control and environment configuration
Main:
  lui $a0, IO_BASE_ADDR      ; Global variable. Load the base address of
     I/O ports into $a0
  // ----------
  // Send commands to GP1 register (0x1F801814) for display configuration
  // (Command = 8-Bit MSB, Parameter = 24-Bit LSB)
  // CCPPPPPP: CC=Command PPPPPP=Parameter
  // ----------
  li $t1, 0x00000000         ; 00 = Reset GPU
  sw $t1, GP1($a0)           ; Write to GP1 register

  li $t1, 0x03000000         ; 03 = Enable display
  sw $t1, GP1($a0)           ; Write to GP1 register

  li $t1, 0x08000001         ; 08 = Display mode (320x240, 15-bit, NTSC)
  sw $t1, GP1($a0)           ; Write to GP1 register

  li $t1, 0x06C60260         ; 06 = Horizontal display range -
     0bxxxxxxxxxxXXXXXXXXXX (3168..608)
  sw $t1, GP1($a0)           ; Write to GP1 register

  li $t1, 0x07042018         ; 07 = Vertical display range -
     0byyyyyyyyyyYYYYYYYYYY (264..24)
  sw $t1, GP1($a0)           ; Write to GP1 register
  // ----------
  // Send commands to GP0 register (0x1F801810) to configure drawing area
  // ----------
  li $t1, 0xE1000400         ; E1 = Drawing mode settings
  sw $t1, GP0($a0)           ; Write to GP0 register

  li $t1, 0xE3000000         ; E3 = Drawing area top left vertex -
     0bYYYYYYYYYYXXXXXXXXXX (10 bits for Y and X)
  sw $t1, GP0($a0)           ; Write to GP0 register

  li $t1, 0xE403BD3F         ; E4 = Drawing area bottom right vertex -
     0bYYYYYYYYYYXXXXXXXXXX (10 bits for X=319 and Y=239)
```

```
sw $t1, GP0($a0)            ; Write to GP0 register

li $t1, 0xE5000000          ; E5 = Drawing offset -
   0bYYYYYYYYYYYYXXXXXXXXXXXX (X=0, Y=0)
sw $t1, GP0($a0)            ; Write to GP0 register
// ----------
// Clear the screen (draw a filled rectangle in VRAM)
// ----------
li $t1, 0x02422E1B          ; 02 = Create rectangle in VRAM (Color:
   0xBBGGRR)
sw $t1, GP0($a0)            ; Write to GP0 register

li $t1, 0x00000000          ; Top left corner coordinates (0,0) -
   0xYYYYXXXX
sw $t1, GP0($a0)            ; Write to GP0 register

li $t1, 0x00EF013F          ; Rectangle size (Height=239, Width=319) -
   0xHHHHWWWW
sw $t1, GP0($a0)            ; Write to GP0 register
// ----------
// Draw a flat-shaded triangle
// ----------
li $t1, 0x20FF00FF          ; 20 = Flat-shaded triangle (Color: 0xBBGGRR)
sw $t1, GP0($a0)            ; Write to GP0 register

li $t1, 0x00320032          ; Vertex 1: Coordinates (50, 50) (Parameter
   0xYyyyXxxx)
sw $t1, GP0($a0)            ; Write to GP0 register

li $t1, 0x001E0064          ; Vertex 2: Coordinates (100, 30) (Parameter
   0xYyyyXxxx)
sw $t1, GP0($a0)            ; Write to GP0 register

li $t1, 0x0064006E          ; Vertex 3: Coordinates (110, 100) (Parameter
   0xYyyyXxxx)
sw $t1, GP0($a0)            ; Write to GP0 register
// ----------
// Draw a flat-shaded quadrilateral
// ----------
li $t1, 0x28FF0000          ; 28 = Flat-shaded quadrilateral (Color:
   0xBBGGRR)
sw $t1, GP0($a0)            ; Write to GP0 register

li $t1, 0x00960096          ; Vertex 1: Coordinates (150, 150) (Parameter
   0xYyyyXxxx)
sw $t1, GP0($a0)            ; Write to GP0 register
```

```
76
77    li $t1, 0x006400BE          ; Vertex 2: Coordinates (190, 100) (Parameter
          0xYyyyXxxx)
78    sw $t1, GP0($a0)            ; Write to GP0 register
79
80    li $t1, 0x00DC00A0          ; Vertex 3: Coordinates (160, 220) (Parameter
          0xYyyyXxxx)
81    sw $t1, GP0($a0)            ; Write to GP0 register
82
83    li $t1, 0x00C80104          ; Vertex 4: Coordinates (260, 200) (Parameter
          0xYyyyXxxx)
84    sw $t1, GP0($a0)            ; Write to GP0 register
85    // ----------
86    // Draw a Gouraud-shaded triangle
87    // ----------
88    li $t1, 0x30FF31FF          ; 30 = Gouraud-shaded triangle (Vertex 1
          color: 0xBBGGRR)
89    sw $t1, GP0($a0)            ; Write to GP0 register
90
91    li $t1, 0x00B40014          ; Vertex 1: Coordinates (20, 180) (Parameter
          0xYyyyXxxx)
92    sw $t1, GP0($a0)            ; Write to GP0 register
93
94    li $t1, 0x00A88332          ; Vertex 2 color: 0xBBGGRR
95    sw $t1, GP0($a0)            ; Write to GP0 register
96
97    li $t1, 0x006400A0          ; Vertex 2: Coordinates (160, 100) (Parameter
          0xYyyyXxxx)
98    sw $t1, GP0($a0)            ; Write to GP0 register
99
100   li $t1, 0x0000FF00          ; Vertex 3 color: 0xBBGGRR
101   sw $t1, GP0($a0)            ; Write to GP0 register
102
103   li $t1, 0x00E6004B          ; Vertex 3: Coordinates (75, 230) (Parameter
          0xYyyyXxxx)
104   sw $t1, GP0($a0)            ; Write to GP0 register
105 LoopForever:
106   j LoopForever               ; Infinite loop
107   nop
```

Codice 4.6: First example of rendering

4.8 Memory Management

4.8.1 The MIPS Application Binary Interface

The ABI (Application Binary Interface) acronym defines a set of rules and interfaces that allow compiled programs to communicate with the underlying operating system or hardware. In other words, the ABI specifies how data and computational routines are handled at the machine code level, thus representing a format strongly dependent on the hardware architecture.

A particularly important aspect of an ABI is the ”calling convention”. This defines how programs should pass arguments during a function call, who is responsible for saving registers, and how return values should be handled. The standardization of these rules ensures that compiled programs can reliably function on a specific hardware platform.

The ABI for the MIPS architecture precisely defines register usage and memory management. Some examples are:

- **Passing arguments**: Registers $a0-$a3 are used to pass up to four arguments. If more than four arguments are required, they will be passed via the stack.
- **Return values**: A function’s return value is stored in the $v0 register. An optional second return value is stored in $v1.
- **Special Registers**: Registers like $ra and $k0-$k1 are reserved for storing the return address of a function call and for the operating system kernel, respectively.
- **Stack management**: The stack can be used to save temporary data and local variables. However, argument registers are not automatically saved to the stack by the caller. The $sp register is reserved for the stack pointer.
- **Preserved registers**: Registers $s0-$s7 must be preserved across a function call. This means that if a function modifies them, it is obligated to save their initial content to the stack and restore it before terminating.

Sources: [99].

4.8.2 The Concepts of Heap and Stack

Heap and stack represent two fundamental memory areas available to a program during execution.

These areas, although both essential for memory management, have different characteristics and uses.

Stack

The stack is a fixed-size memory area, primarily used to declare local variables, manage static arrays, pass parameters to functions, and store return addresses during subroutine calls.

This memory portion operates according to the LIFO model, where the last data inserted is the first to be retrieved. To manage this structure, a special register called the stack pointer ($sp) is used, which keeps track of the current address of the stack.

Each time a ”push” operation (inserting data) is performed, the stack pointer moves by decrementing its value (in architectures that grow downwards). Similarly, a ”pop” operation (retrieving data) advances the pointer, freeing the memory.

Heap

The heap, unlike the stack, is a memory area that does not have predefined sizes or fixed limits, but is managed dynamically during the program's execution. It is an ideal space for allocating dynamic memory and managing objects or data structures of variable sizes.

Unlike the stack, the programmer (or the execution system) must manually allocate and deallocate memory, avoiding wastage or errors, such as memory leaks.

Uses

The stack proves particularly useful in contexts where subroutines require the storage of parameters or temporary values. In MIPS systems, it is possible to take advantage of dedicated registers, like $s1, $s2, etc., to save the necessary parameters.

However, it is often preferable to save such values on the stack to preserve the state of the registers and improve code readability.

The stack pointer, acting as a dynamic pointer, allows easy access to the current position in the stack, simplifying the storage and retrieval operations.

Sources: [100].

4.8.3 Second Example of Basic Rendering in MIPS Assembly

This code example puts into practice the concepts of Stack, calling conventions, bit shifting, logical and arithmetic operations covered in the previous chapters.

Starting from the code presented in chapter 4.7.6, the section related to drawing the three graphic primitives (flat triangle, flat quadrilateral, and Gouraud triangle) will be modified to use a subroutine dedicated exclusively to drawing a flat triangle. The subroutine will receive as parameters, through the memory stack, the three vertices and the color.

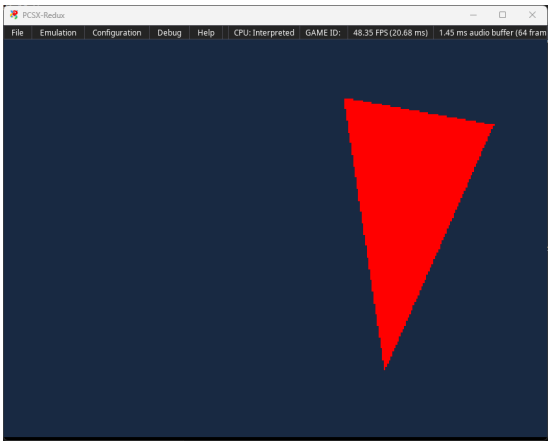


Figure 4.15: Screen taken from the emulator during the program execution.

```
[....] variable declaration, display configuration, drawing area and
    screen clear

  ; ---------------------
  ; Set the stack pointer
  ; ---------------------
  la $sp, 0x00103CF0         ; Initialize the stack pointer (SP)
  ; ---------------------
  ; Draw a flat-shaded triangle using a subroutine
  ; ---------------------
  addiu $sp, -(4 * 7)        ; Reserve space on the stack for 7 parameters
  li $t0, 0x0000FF           ; Color: 0xBBGGRR
  sw $t0, 0($sp)             ; Save parameter on stack
  li $t0, 200                ; x1
  sw $t0, 4($sp)             ; Save parameter on stack
  li $t0, 40                 ; y1
  sw $t0, 8($sp)             ; Save parameter on stack
  li $t0, 288                ; x2
  sw $t0, 12($sp)            ; Save parameter on stack
  li $t0, 56                 ; y2
  sw $t0, 16($sp)            ; Save parameter on stack
  li $t0, 224                ; x3
  sw $t0, 20($sp)            ; Save parameter on stack
  li $t0, 200                ; y3
  sw $t0, 24($sp)            ; Save parameter on stack
  jal DrawFlatTriangle       ; Jump to subroutine to draw the triangle
  nop

```

```
LoopForever:
  j LoopForever              ; Infinite loop to block execution
  nop

; ---------------------
; Subroutine to draw a flat-shaded triangle
; Arguments:
; $sp+0  = Color
; $sp+4  = x1 ; $sp+8  = y1
; $sp+12 = x2 ; $sp+16 = y2
; $sp+20 = x3 ; $sp+24 = y3
; ---------------------
DrawFlatTriangle:
  lui $t0, 0x2000            ; Command: Flat-shaded triangle
  lw  $t1, 0($sp)            ; Color
  or  $t8, $t0, $t1          ; Command | Color
  sw  $t8, GP0($a0)          ; Write command to GP0

  lw   $t1, 4($sp)           ; x1
  lw   $t2, 8($sp)           ; y1
  sll  $t2, $t2, 16          ; y1 <<= 16
  or   $t8, $t1, $t2         ; x1 | y1
  sw   $t8, GP0($a0)         ; Write vertex 1

  lw   $t1, 12($sp)          ; x2
  lw   $t2, 16($sp)          ; y2
  sll  $t2, $t2, 16          ; y2 <<= 16
  or   $t8, $t1, $t2         ; x2 | y2
  sw   $t8, GP0($a0)         ; Write vertex 2

  lw   $t1, 20($sp)          ; x3
  lw   $t2, 24($sp)          ; y3
  sll  $t2, $t2, 16          ; y3 <<= 16
  or   $t8, $t1, $t2         ; x3 | y3
  sw   $t8, GP0($a0)         ; Write vertex 3

  addiu $sp, $sp, (4 * 7)    ; Restore the stack pointer
  jr $ra                     ; Return to caller
  nop

.close
```

Codice 4.7: Second example of rendering

4.8.4 The Concept of Variable and Vector Alignment

In this context of programming, we can distinguish between local and global variables. Local variables, also known as ”automatic variables”, are temporarily allocated in registers like $t0, $t1, etc. On the other hand, global variables, or external variables, require space in the main memory and must be explicitly declared.

A crucial aspect of declaring variables in memory is alignment. The fundamental rule establishes that the starting address of a variable must be a multiple of its size. For example, word-type variables (32 bit) must be aligned to addresses that are multiples of 4, while halfword-type variables (16 bit) require alignment to multiples of 2. Byte-type variables (8 bit), on the other hand, do not require specific alignment, as they occupy only one byte. Directives like .hword and .word in MIPS Assembly automatically ensure correct alignment in memory, making the management process simpler for the programmer.

When working with vectors, each element follows the same alignment rule, while the entire vector must start from a correctly aligned address. On PSX, alignment becomes even more relevant, as operations like DMA transfer and the use of the frame buffer often require data aligned to 64-byte blocks to optimize speed and ensure correct processing. Sources: [101].

```
.org 0x80010000 // Starting point in memory
// Declaration of a byte with value 0x1F (31 decimal)
var_byte: .byte 0x1F
// Declaration of a halfword (2 bytes) with value 0x1234
var_hword: .hword 0x1234
// Declaration of a word (4 bytes) with value 0x12345678
var_word: .word 0x12345678
// Declaration of a .word vector with 3 values (4,5,6)
vector_word_3: .word 4, 5, 6
// Declaration of a .word vector with 256 undefined values
.align 2                          // Alignment to 4 bytes (2^2 = 4)
vector_word_256: .space 256 * 4  // Reserve space for 256 words (256 * 4
    bytes)
// Declaration of a .hword vector with 256 undefined values
.align 1                           // Alignment to 2 bytes (2^1 = 2)
vector_hword_256: .space 256 * 2 // Reserve space for 256 halfwords (256 *
    2 bytes)
// Declaration of a .byte vector with 256 undefined values (no alignment
    needed)
vector_byte_256: .space 256      // Reserve space for 256 bytes
.close
```

Codice 4.8: Examples of variable and vector alignment

4.8.5 Third Example of Basic Rendering in MIPS Assembly

This code example illustrates how to load a 24-bit per pixel (bpp) texture into the PSX frame buffer. After configuring the display and drawing area, the image data is transferred from the main memory to the VRAM using DMA for faster transfer.

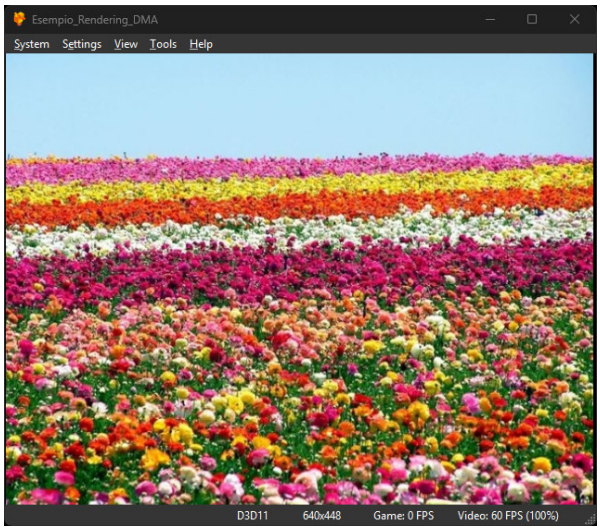


Figure 4.16: Screenshot taken from the emulator during program execution. In this case, [102] was used, which correctly handles the 24bpp mode, unlike the previous emulator.

The code applies data alignment in memory to optimize access and ensure faster transfers. In addition, it uses bit shifting operations to dynamically calculate the size of data to be transferred, improving the overall efficiency of the program.

```
.psx
.create "03_Rendering_DMA.bin", 0x80010000
.org 0x80010000

// Definition of constants and IO_BASE_ADDR, GP0, GP1 registers as in
   previous examples

// Constants for DMA transfer
DMA_BASE_ADDR    equ 0x1F801080    // Base address of the DMA controller
DMA_GPU_CHCR     equ 0x70          // Offset of the GPU channel control
   register
DMA_GPU_MADR     equ 0x74          // Offset of the memory address register
DMA_GPU_BCR      equ 0x78          // Offset of the block control register
DMA_CONTROL      equ 0x7C          // General DMA control register

// Image constants
IMG_WIDTH        equ 640
IMG_HEIGHT       equ 480
IMG_SIZE_BYTES   equ 921600        // 640 x 480 x 3 bytes per pixel (24bpp)

Main:
    lui $a0, IO_BASE_ADDR

```

```
    // Send GPU RESET and Display Enable commands as in previous examples

    li $t1, 0x08000037                // Command 08: Display mode
       (640x480, 24bpp, NTSC)
    sw $t1, GP1($a0)                  // Write to GP1 register

    li $t1, 0x06C60260                // Command 06: Horizontal display
       range (3168..608)
    sw $t1, GP1($a0)                  // Write to GP1 register

    li $t1, 0x0707E018                // Command 07: Vertical display
       range (504..24)
    sw $t1, GP1($a0)                  // Write to GP1 register

    // Send commands to GP0 register to configure drawing area as in
       previous examples

    // DMA transfer configuration
    // 1. Set up the DMA controller for transfer to GPU (Channel 2)
    // 2. Configure DMA transfer to send the image to VRAM

    // Base address of the DMA controller
    lui $t5, (DMA_BASE_ADDR >> 16)    // Load upper part of DMA base
       address into $t5

    // Set the MADR (Memory Address Register) with the image address
    la  $t1, Image                    // Load image address
    ori $t0, $t5, DMA_GPU_MADR        // Address of MADR register for GPU
       channel
    sw  $t1, 0($t0)                   // Write image address to MADR
       register
    nop                               // Delay slot

    // Calculate number of words to transfer (IMG_SIZE_BYTES / 4)
    li  $t1, IMG_SIZE_BYTES           // Load total image size in bytes
    srl $t1, $t1, 2                   // Divide by 4 (shift right logical
       by 2 bits)
    nop                               // Delay slot

    // Set the BCR (Block Control Register) with the transfer length
    ori $t0, $t5, DMA_GPU_BCR         // Address of BCR register for GPU
       channel
    sw  $t1, 0($t0)                   // Write length to BCR register
    nop                               // Delay slot

    // Set the CHCR (Channel Control Register) to start the transfer
```

```
59      li   $t1, 0x01000200                // Set channel control (Normal mode,
            CPU->GPU)
60      ori $t0, $t5, DMA_GPU_CHCR         // Address of CHCR register for GPU
            channel
61      sw   $t1, 0($t0)                   // Write to CHCR register
62      nop                                // Delay slot
63
64      // Set the start bit in CHCR register to begin the transfer
65      lw   $t1, 0($t0)                   // Read current CHCR value
66      lui $t2, 0x0100                    // Load 0x01000000 into $t2 (bit 24)
67      or   $t1, $t1, $t2                 // Set start bit (bit 24)
68      sw   $t1, 0($t0)                   // Write to CHCR register to start
            transfer
69      nop                                // Delay slot
70
71  // Wait for DMA transfer to complete
72  WaitDMADone:
73      lw   $t1, 0($t0)                   // Read current CHCR value
74      lui $t2, 0x0100                    // Load 0x01000000 into $t2
75      and $t1, $t1, $t2                  // Check start bit
76      bne  $t1, $zero, WaitDMADone       // If DMA is still in progress, keep
            waiting
77      nop                                // Delay slot
78
79  LoopForever:
80      j LoopForever                      // Infinite loop
81      nop                                // Delay slot
82
83  // Data Section (Image)
84      .align 2                           // Align to multiple of 4 bytes (2^2)
85  Image:
86      .incbin "campo-di-fiori_640x480_24bit.bin"   // Include 24bpp binary
            image (921600 bytes)
87
88  .close
```

Codice 4.9: Example of uploading a 24bpp image via DMA

```
.psx
.create "03_Rendering_DMA.bin", 0x80010000
.org 0x80010000

// Definition of constants and IO_BASE_ADDR, GP0, GP1 registers as in
   previous examples

// Constants for DMA transfer
DMA_BASE_ADDR   equ 0x1F801080   // Base address of the DMA controller
DMA_GPU_CHCR    equ 0x70         // Offset of the GPU channel control
   register
DMA_GPU_MADR    equ 0x74         // Offset of the memory address register
DMA_GPU_BCR     equ 0x78         // Offset of the block control register
DMA_CONTROL     equ 0x7C         // General DMA control register

// Image constants
IMG_WIDTH       equ 640
IMG_HEIGHT      equ 480
IMG_SIZE_BYTES  equ 921600       // 640 x 480 x 3 bytes per pixel (24bpp)

Main:
    lui $a0, IO_BASE_ADDR

    // Send GPU RESET and Display Enable commands as in previous examples

    li $t1, 0x08000037               // Command 08: Display mode
       (640x480, 24bpp, NTSC)
    sw $t1, GP1($a0)                 // Write to GP1 register

    li $t1, 0x06C60260               // Command 06: Horizontal display
       range (3168..608)
    sw $t1, GP1($a0)                 // Write to GP1 register

    li $t1, 0x0707E018               // Command 07: Vertical display
       range (504..24)
    sw $t1, GP1($a0)                 // Write to GP1 register

    // Send commands to GP0 register to configure drawing area as in
       previous examples

    // DMA transfer configuration
    // 1. Set up the DMA controller for transfer to GPU (Channel 2)
    // 2. Configure DMA transfer to send the image to VRAM

    // Base address of the DMA controller
```

```
    lui $t5, (DMA_BASE_ADDR >> 16)    // Load upper part of DMA base
        address into $t5

    // Set the MADR (Memory Address Register) with the image address
    la  $t1, Image                    // Load image address
    ori $t0, $t5, DMA_GPU_MADR        // Address of MADR register for GPU
        channel
    sw  $t1, 0($t0)                   // Write image address to MADR
        register
    nop                               // Delay slot

    // Calculate number of words to transfer (IMG_SIZE_BYTES / 4)
    li  $t1, IMG_SIZE_BYTES           // Load total image size in bytes
    srl $t1, $t1, 2                   // Divide by 4 (shift right logical
        by 2 bits)
    nop                               // Delay slot

    // Set the BCR (Block Control Register) with the transfer length
    ori $t0, $t5, DMA_GPU_BCR         // Address of BCR register for GPU
        channel
    sw  $t1, 0($t0)                   // Write length to BCR register
    nop                               // Delay slot

    // Set the CHCR (Channel Control Register) to start the transfer
    li  $t1, 0x01000200               // Set channel control (Normal mode,
        CPU->GPU)
    ori $t0, $t5, DMA_GPU_CHCR        // Address of CHCR register for GPU
        channel
    sw  $t1, 0($t0)                   // Write to CHCR register
    nop                               // Delay slot

    // Set the start bit in CHCR register to begin the transfer
    lw  $t1, 0($t0)                   // Read current CHCR value
    lui $t2, 0x0100                   // Load 0x01000000 into $t2 (bit 24)
    or  $t1, $t1, $t2                 // Set start bit (bit 24)
    sw  $t1, 0($t0)                   // Write to CHCR register to start
        transfer
    nop                               // Delay slot

// Wait for DMA transfer to complete
WaitDMADone:
    lw  $t1, 0($t0)                   // Read current CHCR value
    lui $t2, 0x0100                   // Load 0x01000000 into $t2
    and $t1, $t1, $t2                 // Check start bit
    bne  $t1, $zero, WaitDMADone      // If DMA is still in progress, keep
        waiting
```

```
77      nop                                 // Delay slot
78
79  LoopForever:
80      j LoopForever                       // Infinite loop
81      nop                                 // Delay slot
82
83  // Data Section (Image)
84      .align 2                            // Align to multiple of 4 bytes (2^2)
85  Image:
86      .incbin "campo-di-fiori_640x480_24bit.bin"   // Include 24bpp binary
            image (921600 bytes)
87
88  .close
```

Codice 4.10: Example of uploading a 24bpp image via DMA

Chapter 5

Programming in C

This chapter delves into development for PlayStation using the C language, a high-level programming solution supported by the console that can significantly simplify the workflow. The goal is to investigate how the C language interacts with hardware and official libraries, as well as to explore strategies and resources useful for fully leveraging the system's potential.

Unlike previous generations of consoles such as the Sega Genesis, NES, or SNES, where programming relied primarily on assembly, the advent of PlayStation marked an important shift towards higher-level languages. The previous chapter, dedicated to MIPS Assembly, provides the theoretical and practical foundations for understanding how C instructions are ultimately translated into machine language.

To recreate a development environment that reflects the operating conditions of the era, a C compiler, a debugger, and Sony's official PSY-Q library will be used. The latter, originally designed for 32-bit systems, has been integrated into an operational context rebuilt through a virtual machine running Windows XP.

Within this environment, the 32-bit version of the PSY-Q library is available, along with a GNU (DJGPP) C compiler and 32-bit Windows development tools from the historical period in question. The necessary resources were obtained from enthusiast websites specializing in PlayStation development, such as: [103][104][105]. For console emulation, the No$psx emulator was chosen, available at [106], while Code::Blocks[107] was adopted as the IDE.

Detailed installation and configuration instructions for the various components will not be provided here, as these procedures are already extensively documented in the cited sources.

5.1 History of the PSY-Q SDK

The history of the Psy-Q SDK began with the collaboration between the companies SN Systems and Cross Products. Founded in 1988, SN Systems developed a fast and efficient assembler for Atari and Amiga called SNASM, which was later modified by Cross Products into SNASM68K. The latter became the standard tool for Sega Mega Drive development. In 1993, SN Systems and Psygnosis launched the Psy-Q kit. This SDK was designed to support multiple platforms, including PlayStation, Super Nintendo, Sega Genesis, and Sega Saturn.

This SDK introduced significant improvements over previous versions, such as support for source-level C debugging and advanced features for large-scale projects. Psy-Q became the primary development tool for PlayStation 1 games, and many successful titles of the era were created using this kit.

The Psy-Q kit included optimized assemblers for the R3000 processor, a high-speed linker, and a debugger for Windows 95 and DOS. Integration with the GNU C compiler enabled developers to convert C source code into optimized MIPS Assembly for PlayStation. In the following years, Psy-Q was rebranded as SDevTC (Sony Developer Toolchain) and continued to be used until the end of the PlayStation's lifecycle. Today, although Psy-Q remains a historically significant resource, open-source alternatives such as PSn00bSDK, CandyK-PSX, and PSXSDK are available. Nevertheless, Psy-Q's role remains central to understanding the historical development on the PSX. Sources: [108].

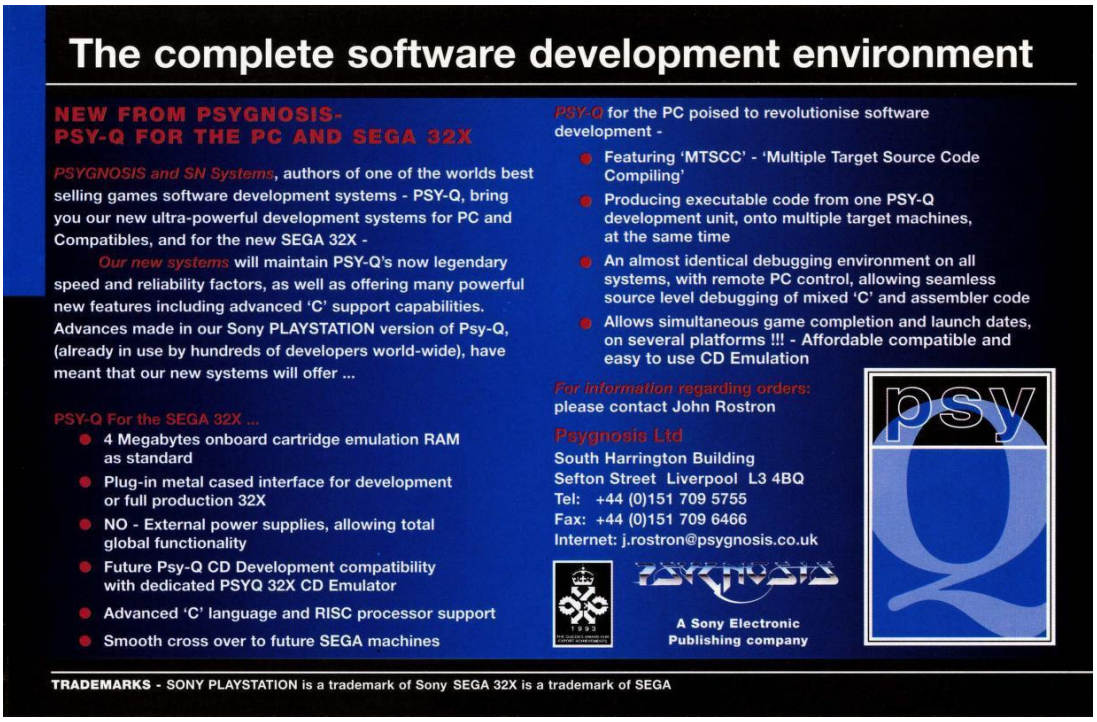


Figure 5.1: Magazine advertisement for the PSY-Q SDK

5.2 Key Programming Concepts

5.2.1 Double Buffering

Double buffering, as described in Chapter 4.7.1, is a technique frequently used on the PS1 to ensure smooth graphical processing free from visual artifacts. Its implementation relies on the combined use of two main components: the frame buffer (see Chapter 4.7.1) and the ordering tables. This setup allows the separation of the area dedicated to image creation from the one used for display, reducing issues such as ”tearing” (a graphical artifact occurring when the on-screen frame contains information from two or more frames) or flickering.

It is common practice to structure the frame buffer to contain two images: one for the processing phase (drawing area) and one for display. The latter is positioned exactly beneath the first image (see Figure 4.7, Chapter 4.7.1). This approach enables the system to continue processing the next frame while the current one is being displayed.

Synchronization between these two areas occurs during a time interval called vertical blanking (VBlank). During this period, the system halts video signal updates to allow the drawing and display areas to be swapped for a smoother transition. This scheme thus maintains a continuous rendering pipeline and optimizes hardware resource usage. At the same time, it guarantees a stable and disturbance-free visual experience.

5.2.2 Z-Sorting

Z-sorting is a technique used to order the objects in a scene based on their depth relative to the camera’s viewpoint. This process is essential to correctly represent a 3D scene on a 2D screen, ensuring that, as in reality, closer objects occlude those farther away. In the PSX environment, this technique is based on an algorithm that sorts polygons according to their depth. Also known as the Painter’s Algorithm [109], it involves sequentially drawing objects starting from those farthest from the camera, ensuring that nearer objects overwrite those behind them. Compared to other techniques such as Z-buffering, Z-sorting is better suited to the console’s hardware architecture, as it requires less memory and enables faster processing. An important aspect is the relationship between scene complexity and resource requirements. As the number of objects to sort increases, computation time grows significantly. However, thanks to the use of ordering tables, the PSX implements this technique efficiently.

5.2.3 Ordering Tables

Ordering tables (OT) are typical data structures used to organize and manage graphic commands in the PlayStation environment. These tables function as linked lists that sort graphical

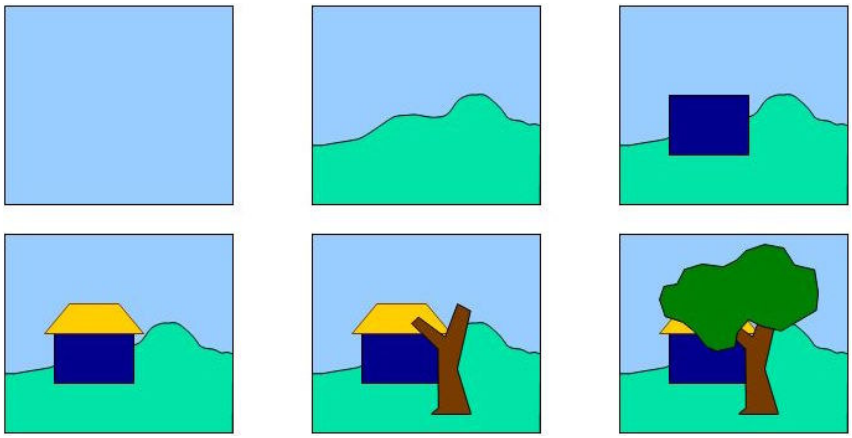


Figure 5.2: Example of the Painter's Algorithm [45]

elements such as polygons, sprites, and line segments based on their visual depth (Z-depth), thus facilitating the Z-sorting process.

Each element in an OT consists of a pointer linking one element to the next and a field specifying its size. Initially, the array is initialized with a default value indicating the end of the list (usually 0xFFFFFF). This approach simplifies adding, removing, and modifying elements without reallocating memory.

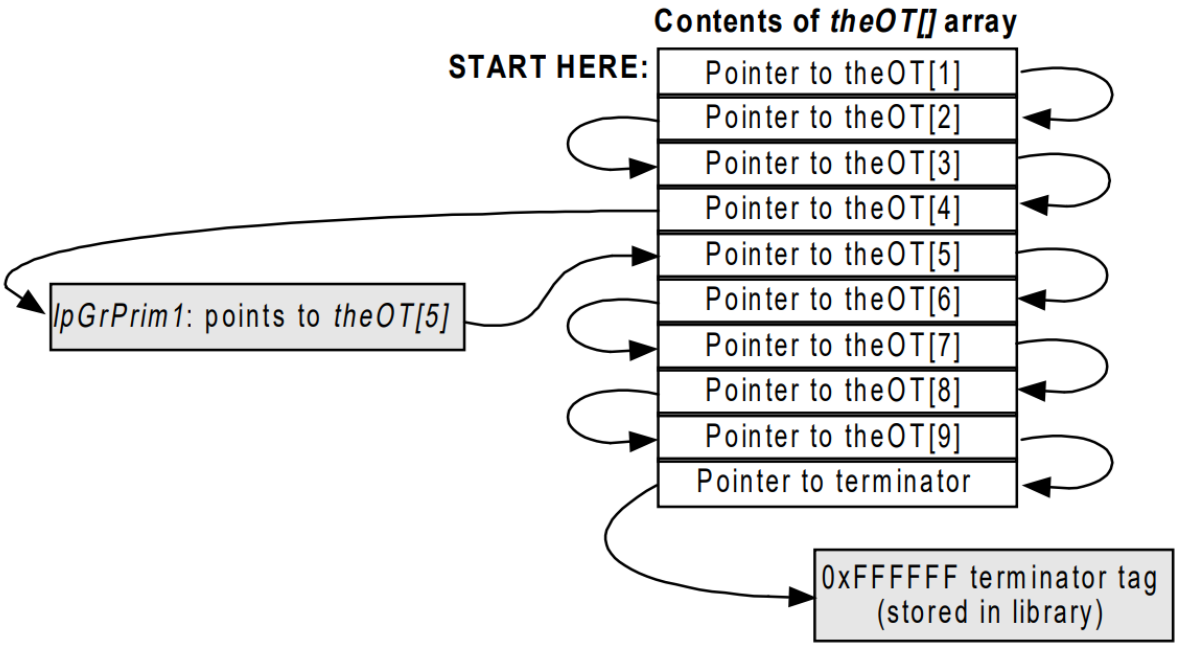


Figure 5.3: Example of an Ordering Table

Depending on the needs, it is also possible to create a reverse ordered table using functions like ClearOTagR(). This, known as a ”Reverse Ordering Table,” is useful for rendering polygons in reverse depth order, thus enabling the implementation of the Painter's Algorithm mentioned in the previous chapter.

The addition of graphic primitives to an OT is done through functions such as addPrim(). Once the OT is complete, its content is sent to the GPU for rendering via the DrawOTag() function, which draws the elements in the established order. The integration of OTs with libraries like LIBGPU, LIBGTE, and LIBGS provides programmers with precise control over rendering modes. The concept of Ordering Tables is extremely flexible regarding graphic resource management. A single position can contain one or multiple primitives. Additionally, using multiple local OTs allows the subdivision of complex scenes into simpler units, which can later be merged into a single global table. Despite these advantages, using this mechanism

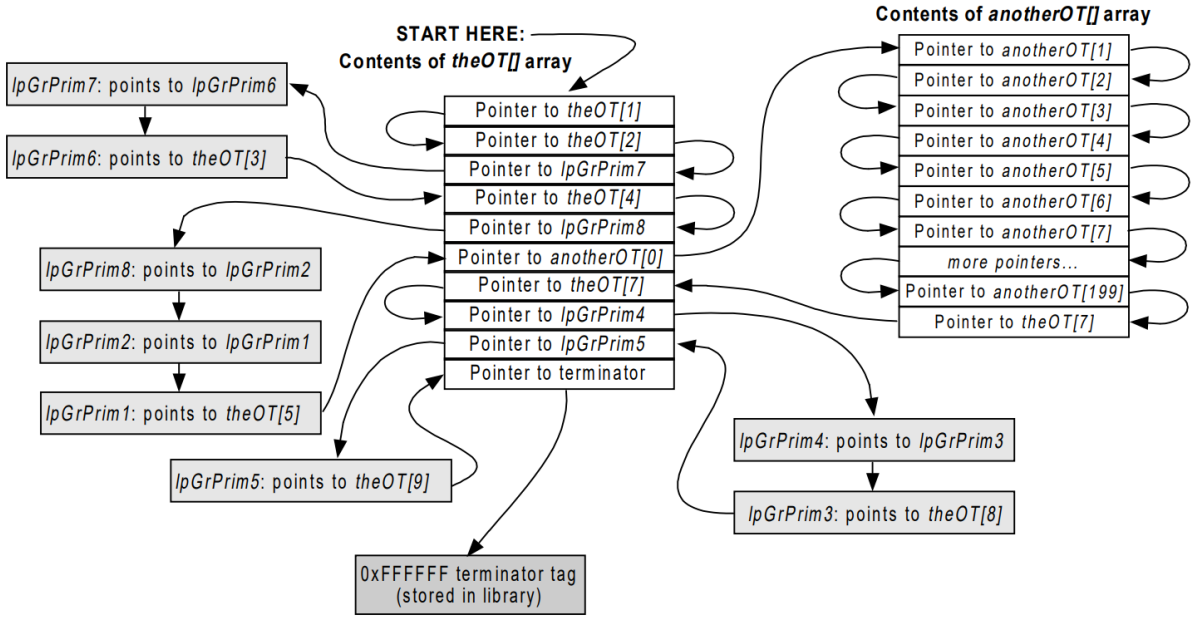


Figure 5.4: Example of using multiple Ordering Tables

can lead to substantial memory usage depending on the level of detail in a scene. To mitigate this issue, two solutions are possible: The first consists of reducing the table size by decreasing the number of represented depth levels (a solution that may cause visual artifacts such as polygon flickering). The second, more efficient solution involves using an offset parameter in the GsClearOt() function, which allows ignoring depth levels below a predefined threshold (enabling lower memory usage without resolution compromises). In conclusion, Ordering Tables allow efficient management of 3D rendering, balancing hardware limitations with the need to represent complex visual scenarios. Sources: [110][111].

5.3 Main System Libraries

The PlayStation operating system libraries are divided into high-level and low-level libraries. This library structure allows choosing the most appropriate level according to one's needs, with the possibility of using both levels simultaneously. The table below describes the main libraries. Sources: [112].

Name	Description
libapi	Kernel library providing an interface (API) between the PSX OS and applications.
libc / libc2	Standard C libraries with functions for character handling, memory operations, character class tests, and utilities.
libmath	Mathematical library compliant with ANSI/IEEE754 standards, supporting floating-point calculations.
libcard	Library for Memory Card control, including filesystem and drivers.
libmcrd	High-level interface for Memory Card management.
libpress	Library for compression and decompression of image and audio data.
libgpu	Basic graphics library for handling primitives such as sprites, polygons, and lines.
libgte	Library for controlling the GTE, with matrix and vertex management.
libgs	Extended 3D graphics system utilizing libgpu and libgte to manage complex objects and background surfaces.
libcd	Library for reading data from the CD-ROM and playback of digital audio (DA) and XA sound.
libds	Extended CD-ROM library with advanced control functions and error recovery.
libetc	Library for callback control and low-level peripheral and interrupt handling.
libtap	Library for accessing multiple controllers and Memory Cards via the Multi-Tap.
libgun	Library for input devices such as the Light Pen connected to controller ports.
libpad	Library for accessing controllers, including those using extended protocols such as the DualShock.
libcomb	Library for communication via the Link Cable, supporting communication blocks of 8 bits.
libsnd	Extended audio library for playback of pre-recorded sound sequences.
libspu	Basic library for controlling the SPU (Sound Processing Unit).
libsio	Library for managing serial I/O via SIO 1.
libhmd	Library for handling the HMD format integrating modeling, animation, textures, and MIMe data.
libmcx	Library for accessing PDA functions when inserted into a Memory Card slot.
mcgui	Module supporting loading and saving data on Memory Cards and providing user interface functionalities.

5.4 Geometry Transformation Engine

As explained in Chapter 3, the GTE is a high-speed geometric processor designed to handle vectors and matrices. Thanks to its integrated multiplier, accumulator, and divider units, the GTE supports complex real-time calculations using fixed-point fractional numbers. Its main features include:

- High-speed matrix calculations.
- Fast coordinate transformations.
- Efficient perspective projections.
- Optimized lighting calculations.

Sources: [113][114][115][116].

5.4.1 3D Transformations

Regarding 3D rendering, the GTE follows a well-defined pipeline to transform a three-dimensional world representation into a two-dimensional image on the screen. This process includes the following stages:

- 3D objects are generated with coordinates defined relative to a local coordinate system, where all vertices refer to an origin point (also called the ”pivot point”).
- To position the object in a 3D world, vertices are transformed into the World Coordinate System by multiplying the object's vertices by a world matrix that includes rotation, scale, and translation information.
- The scene is then transformed into a view volume using a view matrix. At this stage, the coordinate system origin becomes the camera position, which looks toward a defined target.
- The final step projects the vertices onto the 2D screen via a ”perspective divide,” an operation that divides the original XY values by the Z component (depth). This process proportionally reduces the size of distant objects, simulating perspective.

Once these transformations are completed, the resulting 2D data is sent to the GPU for rasterization, finalizing the rendering process.

5.4.2 Examples of GTE Instructions

Vertex Transformation (RTPT)

```
SVECTOR vertex = {100, 200, 300};   // Vertex coordinate in local space
VECTOR   translation = {0, 0, 1024}; // Translation in world space
MATRIX   transformMatrix;           // Transformation matrix

// Transformation matrix setup
RotMatrix(&rotation, &transformMatrix);     // Apply a rotation
TransMatrix(&transformMatrix, &translation); // Apply a translation

// Set active matrices
SetRotMatrix(&transformMatrix);
SetTransMatrix(&transformMatrix);

// Load the vertex
gte_ldv0(&vertex);  // Load the first vertex

// Perform the transformation
gte_rtpt();         // Transform and project

// Extract the projected 2D coordinates
long screenX, screenY, screenZ;
gte_stsxy(&screenX);  // 2D coordinates on screen
gte_stsz(&screenZ);   // Depth (Z-buffer)
```

Codice 5.1: This command transforms a set of 3D vertices from world space to screen space coordinates.

Lighting: Normal Calculation (NCLIP)

```
SVECTOR normal = {0, 0, -4096}; // Face normal
VECTOR  light = {0, 0, 4096};   // Light direction

// Load the normal
gte_ldv0(&normal);
// Compute the dot product
gte_nclip();

// Extract the result
long visibility;
gte_stopz(&visibility); // If > 0, the face is visible
```

Codice 5.2: This command determines if a face is visible relative to the camera direction.

Perspective Projection Calculation

```
// 3D point in local space
SVECTOR vertex = {256, 128, 512};

// Translation in world space
VECTOR translation = {0, 0, 1024};

// Matrix setup and transformation
MATRIX transformMatrix;
RotMatrix(&rotation, &transformMatrix);
TransMatrix(&transformMatrix, &translation);
SetRotMatrix(&transformMatrix);
SetTransMatrix(&transformMatrix);

// Load the point and project
gte_ldv0(&vertex);
gte_rtps();

// Extract coordinates and depth
long screenX, screenY, depthZ;
gte_stsxy(&screenX);
gte_stsz(&depthZ);

// Print results
printf("X=%ld, Y=%ld, Z=%ld\n", screenX, screenY, depthZ);
```

Codice 5.3: Projection of a 3D point into 2D screen coordinates.

Shading Interpolation: Gouraud Color Calculation

```
SVECTOR normal = {0, -4096, 0};  // Vertex normal
CVECTOR lightColor = {128, 128, 128}; // Light color

// Set light parameters
SetLightMatrix(&lightMatrix);
SetColorMatrix(&colorMatrix);

gte_ldv0(&normal);  // Load the normal

gte_ncs();          // Compute lighting

// Extract the resulting color
CVECTOR resultColor;
gte_stcl0(&resultColor);
```

Codice 5.4: This command uses normals and light vectors to compute vertex color.

Bounding Box Calculation (AVSZ4)

```
SVECTOR v0 = {0, 0, 512};
SVECTOR v1 = {256, 0, 512};
SVECTOR v2 = {256, 256, 512};
SVECTOR v3 = {0, 256, 512};

// Load the vertices
gte_ldv0(&v0);
gte_ldv1(&v1);
gte_ldv2(&v2);
gte_ldv3(&v3);

// Calculate the average distance
gte_avsz4();

// Extract the distance
long averageZ;
gte_stotz(&averageZ);
```

Codice 5.5: This command calculates the average distance of four vertices from a viewpoint.

Lighting Matrix Setup

```
MATRIX lightMatrix = {
    {0x1000, 0x0000, 0x0000}, // Light direction R
    {0x0000, 0x1000, 0x0000}, // Light direction G
    {0x0000, 0x0000, 0x1000}  // Light direction B
};

MATRIX colorMatrix = {
    {0x0080, 0x0000, 0x0000}, // Light intensity R
    {0x0000, 0x0080, 0x0000}, // Light intensity G
    {0x0000, 0x0000, 0x0080}  // Light intensity B
};

SetLightMatrix(&lightMatrix);
SetColorMatrix(&colorMatrix);
```

Codice 5.6: Configuration to achieve lighting effects.

5.4.3 GTE Register Set

To perform the geometric transformation operations described above, the GTE provides two sets of registers: 32 control registers and 32 general (data) registers.

General Registers

These registers store data operated on by the GTE, such as vertex coordinates, transformation matrices, and lighting parameters. Developers can read from and write to these registers to define processing values.

Table 5.1: GTE (Coprocessor 2) registers and their descriptions.

Register	Type and Name	Description
cop2r0-1	3xS16 VXY0, VZ0	Vector 0 (X, Y, Z).
cop2r2-3	3xS16 VXY1, VZ1	Vector 1 (X, Y, Z).
cop2r4-5	3xS16 VXY2, VZ2	Vector 2 (X, Y, Z).
cop2r6	4xU8 RGBC	Color value/code.
cop2r7	1xU16 OTZ	Average Z value (for Ordering Table).
cop2r8	1xS16 IR0	16-bit accumulator (interpolation).
cop2r9-11	3xS16 IR1, IR2, IR3	16-bit accumulators (vector).
cop2r12-15	6xS16 SXY0, SXY1, SXY2, SXYP	Screen XY coordinate FIFO (3 stages).
cop2r16-19	4xU16 SZ0, SZ1, SZ2, SZ3	Screen Z coordinate FIFO (4 stages).
cop2r20-22	12xU8 RGB0, RGB1, RGB2	CRGB color FIFO/code (3 stages).
cop2r23	4xU8 (RES1)	Reserved.
cop2r24	1xS32 MAC0	32-bit mathematical accumulator (value).
cop2r25-27	3xS32 MAC1, MAC2, MAC3	32-bit mathematical accumulators (vector).
cop2r28-29	1xU15 IRGB, ORGB	RGB color conversion (48-bit vs 15-bit).
cop2r30-31	2xS32 LZCS, LZCR	Leading-zero/one count (sign bit).

Control Registers

These registers contain configuration information that controls GTE behavior, such as projection parameters, scaling, offsets, and lighting calculation settings.

Table 5.2: GTE (Coprocessor 2) control registers and their descriptions.

Register	Type and Name	Description
cop2r32-36	9xS16 RT11, RT12, ..., RT33	Rotation matrix (3x3).
cop2r37-39	3x32 TRX, TRY, TRZ	Translation vector (X, Y, Z).
cop2r40-44	9xS16 L11, L12, ..., L33	Light source matrix (3x3).
cop2r45-47	3x32 RBK, GBK, BBK	Background color (R, G, B).
cop2r48-52	9xS16 LR1, LR2, ..., LB3	Light source color matrix (3x3).
cop2r53-55	3x32 RFC, GFC, BFC	Far color (R, G, B).
cop2r56-57	2x32 OFX, OFY	Screen offset (X, Y).
cop2r58	BuggyU16 H	Projection plane distance.
cop2r59	S16 DQA	Depth queuing parameter A (coefficient).
cop2r60	32 DQB	Depth queuing parameter B (offset).
cop2r61-62	2xS16 ZSF3, ZSF4	Scale factors for average Z.
cop2r63	U20 FLAG	Returns calculation errors.

Dedicated Instruction Set

The Geometry Transformation Engine (GTE), integrated into the CPU, does not use memory mapping like the GPU. Instead, it has a dedicated instruction set specific to Coprocessor 2. Below is an example related to the ”RotTransPers3” instruction:

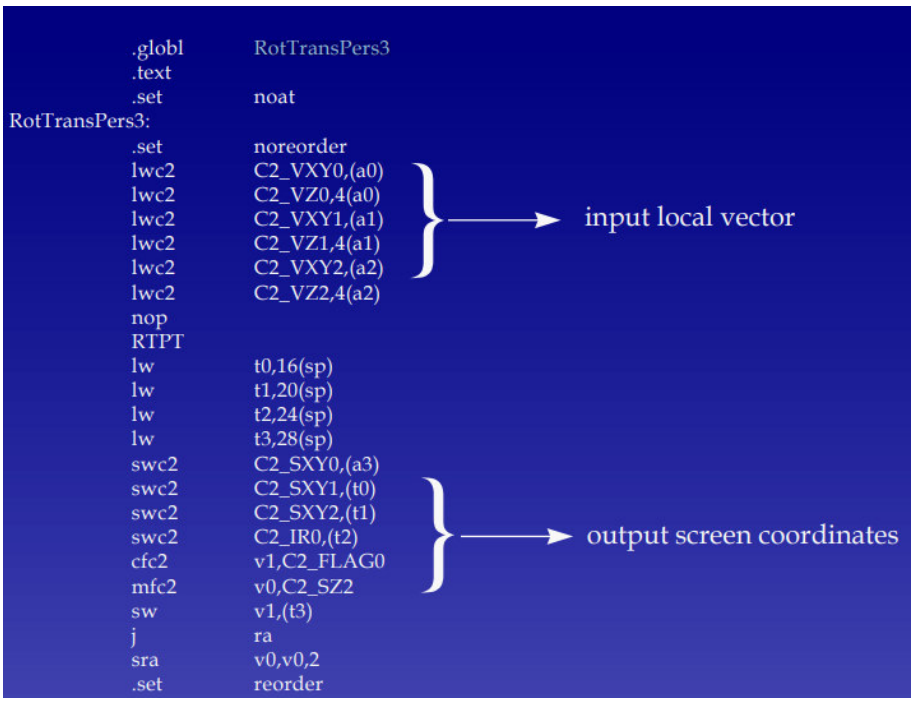


Figure 5.5: GTE instructions used by the RotTransPers3 function.

Instructions such as `lwc2` and `swc2` correspond to ”Load Word Coprocessor 2” and ”Store Word

Coprocessor 2," respectively. In the example, the first part of the code provides the necessary data to the GTE registers, while the second part shows the processed output.

5.5 Clipping

It is not always necessary to render all polygons of an object on the screen. In some cases, rendering every surface can cause visual artifacts such as the *z-fighting* phenomenon.

This issue occurs when two surfaces are located at a similar distance from the camera, and their depth values overlap. This results in continuous polygon "flickering" due to the system's inability to determine which surface should be drawn first. To prevent these problems, clipping techniques can be employed.

Clipping elements outside the camera's view is a very useful step to optimize the use of PSX resources. By eliminating objects that fall outside the visible field, the number of calculations and geometry processing is consequently reduced. Sources: [98].

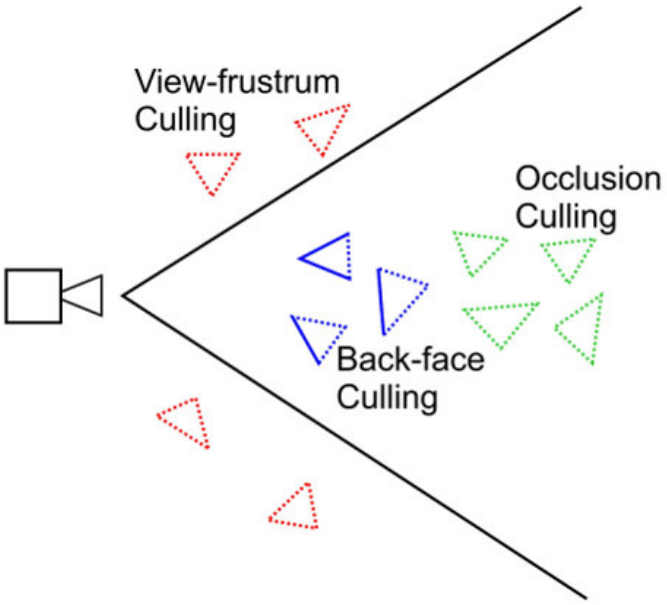


Figure 5.6: Illustration of the "clipping" concept
[46]

5.5.1 Backface Culling

Among the most commonly used clipping techniques is *Normal Clip*, also known as *backface culling* or the "hidden surface removal algorithm". This algorithm allows ignoring the faces

of an object that are not facing the screen, thereby optimizing rendering and avoiding drawing polygons that would not be visible.

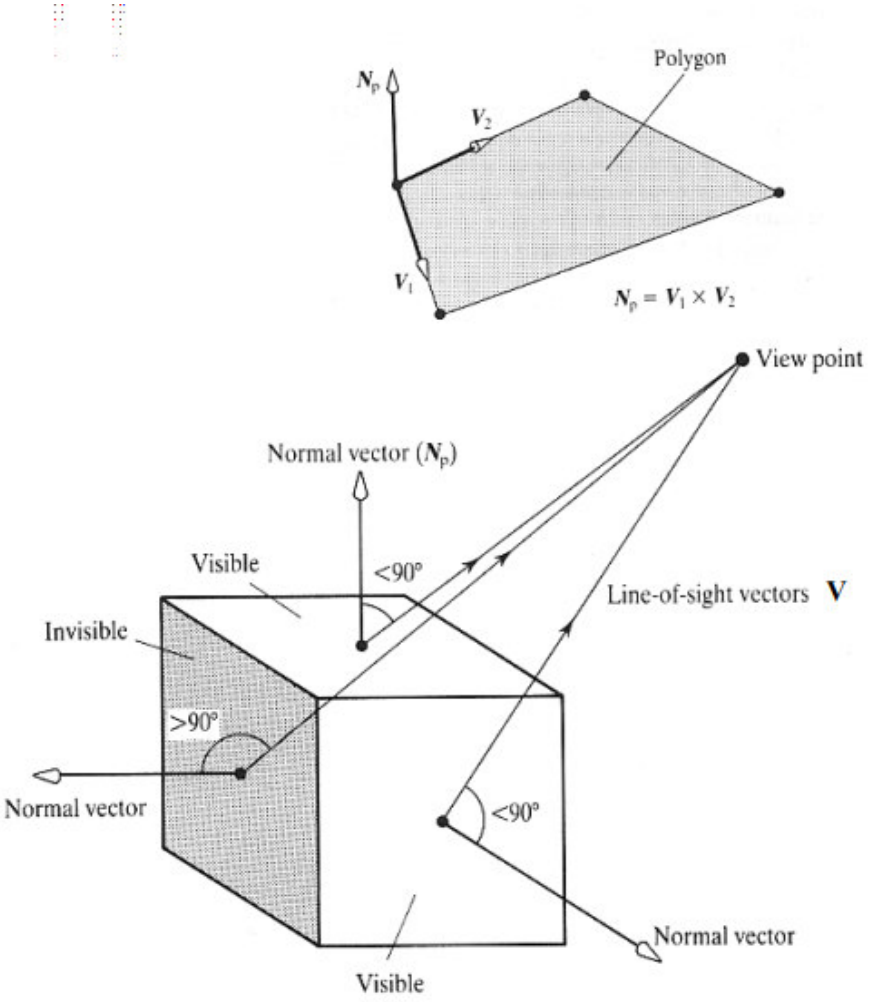


Figure 5.7: Graphical representation of Backface Culling
[47]

This process is based on the orientation of polygon faces relative to the camera position. Each polygon is defined by a normal vector, which is perpendicular to its surface. This *normal vector* is calculated based on the triangle's vertices and represents the direction the face is oriented toward. The technique can be explained through the following steps:

1. Compute the polygon's normal vector (**n**) and consider a camera direction vector (**e**) pointing from the virtual eye (camera) toward the polygon's centroid.

2. To determine if a face is visible, calculate the dot product between the normal (**n**) and the camera vector (**e**). This can be done in two ways:

- Using the non-normalized dot product:

$$\mathbf{e} \cdot \mathbf{n} = e_x n_x + e_y n_y + e_z n_z$$

- Or by calculating the cosine of the angle θ between the two vectors, which includes normalization:

$$\cos(\theta) = \frac{\mathbf{e} \cdot \mathbf{n}}{\|\mathbf{e}\| \cdot \|\mathbf{n}\|}$$

3. If the dot product $\mathbf{e} \cdot \mathbf{n} > 0$ (or $\cos\theta > 0$ if normalized), the polygon is visible. Otherwise, if $\mathbf{e} \cdot \mathbf{n} \leq 0$ (or $\cos\theta \leq 0$), the polygon is not visible.

4. Based on the above calculation, polygons that do not meet the visibility condition are ignored during rendering.

Fortunately, the PlayStation provides a dedicated function called `gte_nclip()`. An example can be seen in code 5.2 of the previous chapter.

5.5.2 Cohen-Sutherland Algorithm

The Cohen-Sutherland algorithm is a clipping technique for 2D line segments. It divides the projection space into nine regions by extending the clipping window borders. Each point in space is identified by a binary code known as an *outcode*, which represents its position relative to the viewport.

After computing the *outcodes* for the two endpoints of the segment, four main cases occur:

- **Case 1: Both outcodes are zero.** The segment lies entirely within the window, and no clipping is necessary.
- **Case 2: The AND operation between outcodes yields a non-zero result.** In this case, the segment lies entirely outside the window and must be discarded.
- **Case 3: Only one outcode is zero.** This means one endpoint is inside the window while the other is outside. Partial clipping is required by calculating intersection points with the window edges.
- **Case 4: The AND operation between outcodes yields zero.** Both endpoints are outside the window, but in different regions. Further checks are necessary to determine whether the segment intersects the window.

Despite its popularity, the Cohen-Sutherland algorithm has a limitation in the fourth case. When the logical AND operation between outcodes o_1 and o_2 returns zero, the segment may cross part of the window without properly accounting for the borders. This can lead to errors in

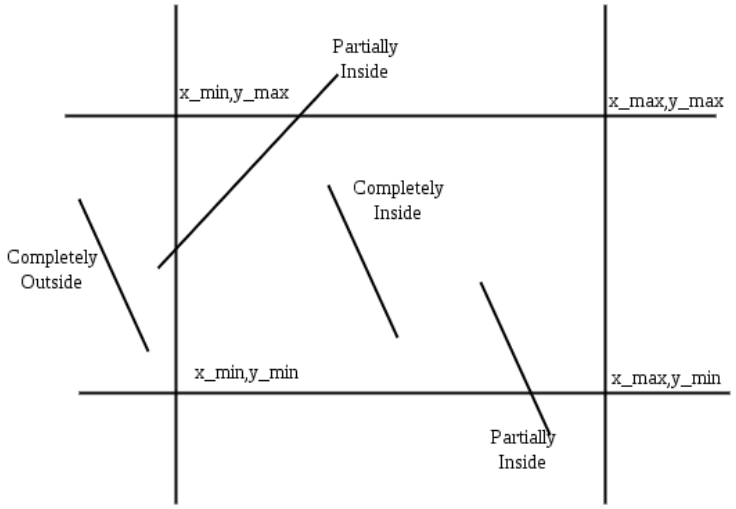


Figure 5.8: Graphical representation of the nine-region subdivision in the Cohen-Sutherland algorithm

[48]

determining which portions of the segment are visible.

5.5.3 Liang-Barsky Algorithm

The Liang-Barsky algorithm addresses the previous algorithm's limitations by providing an alternative approach. It exploits a parametric representation of lines. Given a segment defined by points P_1 and P_2, the algorithm uses a parameter α to represent any point on the line as:

$$p(\alpha) = (1 - \alpha)P_1 + \alpha P_2$$

where α varies between 0 and 1 to describe the segment between P_1 and P_2.

This algorithm calculates the α values corresponding to the intersections of the line with the clipping window edges. Using these values, it determines which portions of the line are visible. This reduces the number of necessary operations and thus the intersection calculations.

5.5.4 Bounding Boxes

Finally, the use of *bounding boxes* will be analyzed. This technique is based on a concept as simple as it is effective. Instead of clipping each individual side or vertex of a polygon, a "rectangle" or volume that encloses the entire object—called the *bounding box*—is used. This box is then compared against the clipping window to quickly determine the object's visibility. This concept excels in scenarios involving complex objects, as it significantly reduces the number of operations required to determine visibility. The cases are as follows:

- If the *bounding box* is completely outside the window, the entire object is discarded.

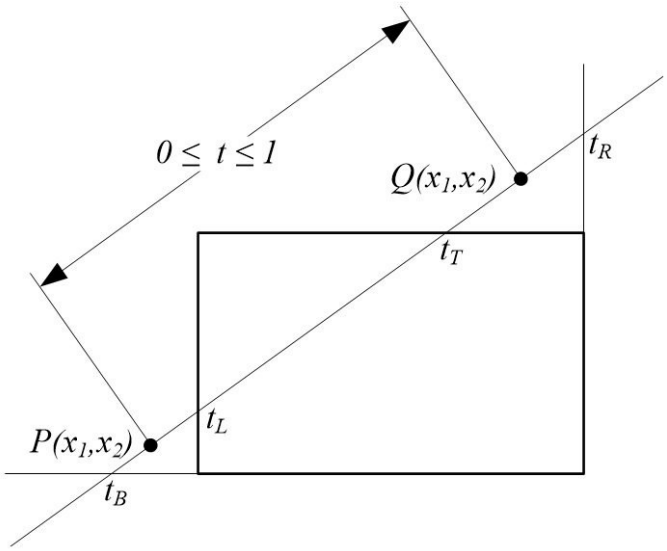


Figure 5.9: Representation of the Liang-Barsky algorithm, where the parameterized segment *t* intersects the clipping window edges, determining the visible portion inside the rectangle. [49]

- If it is completely inside, the object is accepted without further calculations.
- If it intersects the window, more detailed calculations are performed for the actual edges, using the aforementioned algorithms.

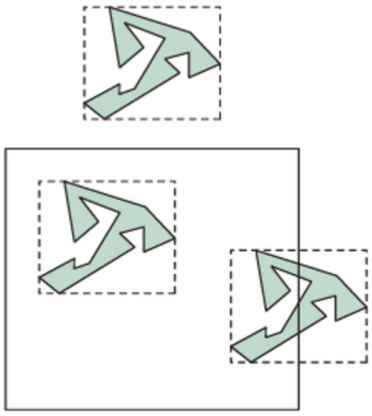


Figure 5.10: Clipping using Bounding Boxes

5.6 Fixed Point Math

Up to this point, the focus has been on using integer values, such as coordinates X and Y and color values RGB. However, when dealing with video games, concepts like movement, acceleration, or rotation require fractional numbers to achieve greater precision.

Fractional numbers allow representing values with finer granularity, making it possible, for example, to express angles in radians. In modern computers, floating-point numbers are natively supported by hardware and can be declared in C language using variables such as `float` (32-bit precision) and `double` (64-bit).

On the PSX platform, the 32-bit limit is not always sufficient to represent real numbers (both positive and negative) and therefore to guarantee a precise representation over a wide range of values. The IEEE754 standard format is designed to represent a broad spectrum of real numbers using:

- **Sign** (S): 1 bit.
- **Exponent** (E): 8 bits.
- **Mantissa** (M): the remaining bits.

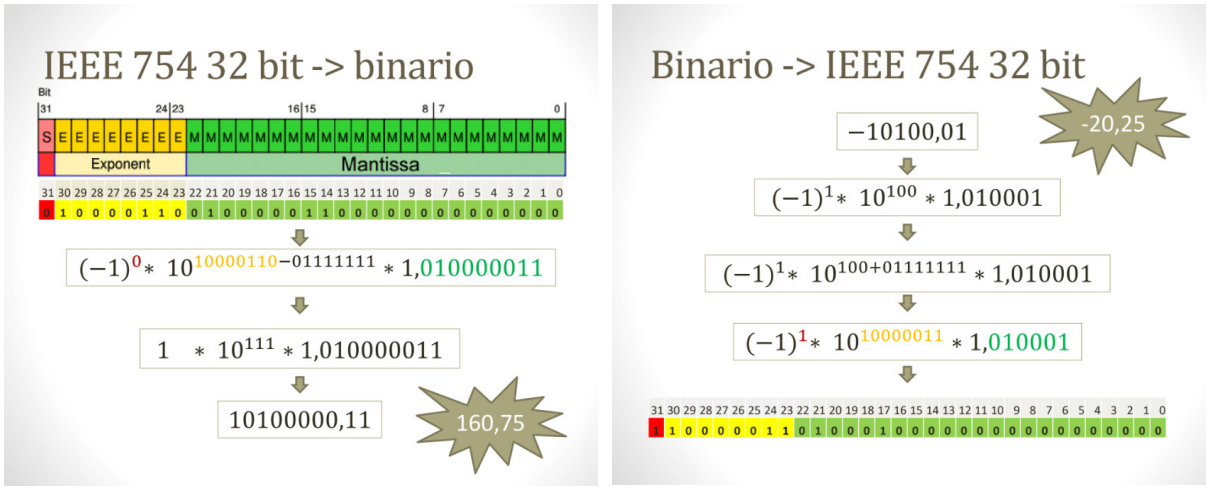


Figure 5.11: Two examples of floating-point number conversion in the IEEE754 standard format [50]

This implementation allows efficient representation of real numbers but introduces a non-constant resolution, also known as *Unit of Least Precision* (ULP), which varies depending on the value. For example:

- 0.004 has a resolution of 0.001
- 24.11 has a resolution of 0.01
- 912.3 has a resolution of 0.1
- 1000 has an integer resolution

The variability of the ULP can lead to inaccuracies in calculations. For instance, an operation

like 0.1 + 0.2 can return 0.30000000000000004.

This issue makes floating-point numbers unsuitable when absolute precision is required, such as in financial calculations or coordinate management in video games.

Fixed Point Math represents a more suitable alternative for systems lacking an FPU, like the PlayStation. This method relies on reserving a fixed number of bits for the integer part of the number and another fixed number for the fractional part. This eliminates the variability of the ULP.

Below are two possible uses of Fixed Point Math with 32 available bits:

- **16.16 Fixed-Point**: 16 bits are dedicated to the integer part and 16 to the fractional part.
- **20.12 Fixed-Point**: 20 bits for the integer part and 12 for the fractional part.

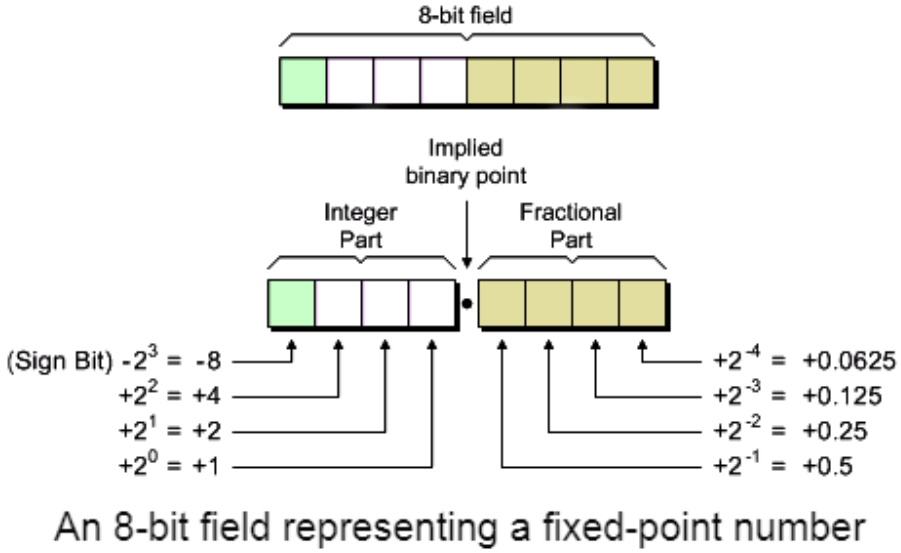


Figure 5.12: Example of a fixed point number. In this case, with 8 bits available. [51]

The 20.12 format is the most commonly used in the PlayStation context as it fits the way the GTE handles data. With 12 fractional bits, the resolution remains constant:

$$\text{Resolution} = \frac{1}{2^{12}} = \frac{1}{4096}.$$

This allows representing 4096 distinct values in the fractional part, where 4096 corresponds to 1.0. This value is also known as the *scaling factor*.

The Psy-Q library defines the constant `ONE` as equivalent to the value 4096. Therefore:

- Adding or subtracting 4096 corresponds to incrementing or decrementing by 1.0.
- Adding 2048 is equivalent to adding 0.5.

- Subtracting 6144 is equivalent to subtracting -1.5.

Addition and subtraction operations are straightforward since they simply involve adding or subtracting the integer values:

```
fix_num1 = 4096; // 1.0
fix_num2 = 2048; // 0.5
result = fix_num1 + fix_num2; // 6144 = 1.5
```

Codice 5.7: Addition of Fixed-Point numbers.

Multiplication Multiplication requires careful handling of the result to avoid overflow and underflow issues. As shown in the figure, when multiplying two fixed-point numbers, the result has more bits than the original numbers. For example, multiplying two 16.15 fixed-point numbers results in a 47-bit number: 1 bit for the sign, 32 bits for the integer part, and 15 additional bits for the fractional part.

After multiplication, the result must be scaled back to the original fixed-point format. To do this, a logical right shift of the excess bits is performed. During this operation:

- Overflow bits (most significant bits) are discarded.
- Underflow bits (least significant bits) are removed.
- The central bits, representing the correct portion of the result, are preserved.

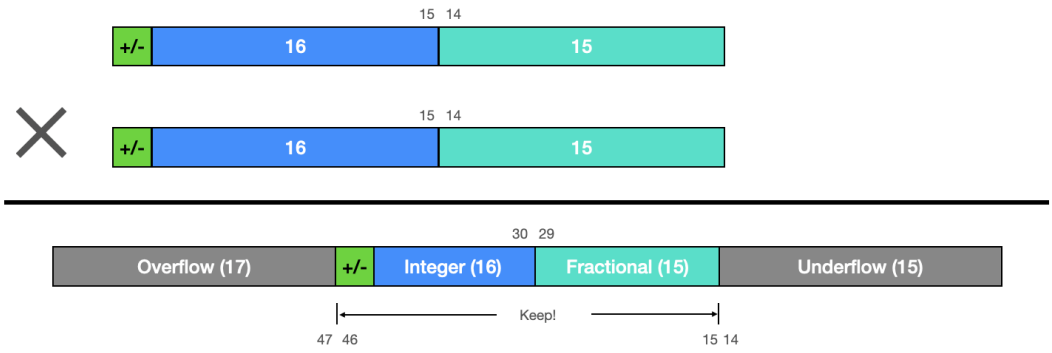


Figure 5.13: Example of multiplication between two fixed-point numbers [52]

```
mul = 2048; // Represents 0.5 in fixed-point
fix_num = (fix_num * mul) >> 15; // Right shift of 15 bits to normalize
```

Codice 5.8: Multiplication of Fixed-Point numbers.

In this case, the logical right shift by 15 bits preserves the 16.15 format, maintaining precision while discarding unnecessary bits. This process ensures the result remains compatible with the original fixed-point system, avoiding errors caused by data size mismatches.

Division Division, on the other hand, involves an intermediate step:

```
fix_num = (fix_num * 4096) / scale;
```

Codice 5.9: Division of Fixed-Point numbers.

Where `scale` represents the desired scaling value, such as 2048 to divide by 0.5 or 6144 to divide by 1.5.

Although the PSX does not have an FPU, its C compiler still allows the use of FLOAT variables. However, these operations are emulated in software, resulting in significantly slower performance compared to the fixed-point approach.

Sources: [117].

5.7 In-depth Analysis of the PSX BIOS

The **BIOS** (Basic Input/Output System) is a set of low-level programs stored in a ROM memory inside the console. These programs are responsible for managing the PlayStation hardware and providing critical functionalities, including:

1. **Hardware Initialization**: When the console is powered on, the BIOS is the first software to execute. It performs a series of system diagnostic checks and initializes the various hardware components. The CPU executes the BIOS ROM code starting from the reset vector located at address `0xBFC00000`. At this stage, the BIOS acts as an intermediary between software and hardware.

2. **Game Loading**: The BIOS handles launching games from the CD-ROM drive. During this process, it verifies the authenticity of the inserted disc and enforces any regional restrictions. Specifically, it executes the `SYSTEM.CNF` file, which contains instructions for loading the game.

3. **Anti-Piracy Measures**: As mentioned above, the system includes protection mechanisms to prevent the execution of pirated copies or games from different regions. This ensures that only official software can be read by the console.

4. **User Interface (UI) Management**: When no CD-ROM is inserted, the BIOS loads a graphical interface commonly known as the *shell*. This allows the user to listen to audio CDs or manage saved data on the Memory Card.

5. **Advanced Features**: The BIOS provides routines for advanced commands, including multithreading management, standard C language functions, and other support functionalities.

6. **Optimization**: Direct access to the ROM is slow since it is connected only via an 8-bit data bus. To improve performance, BIOS APIs are copied into the main RAM during the boot phase. Approximately 64 KB of main RAM is reserved for the BIOS, forming what is called the *Kernel*.

This software is proprietary to **Sony** and is distributed exclusively with official consoles. The binary file, named `SCPH1001.BIN` in the first version, contains the machine code necessary to execute the BIOS functions. Its distribution or use without authorization is, in theory, illegal.

However, in the context of emulation, many emulators require users to provide a BIOS copy extracted from their own console. Additionally, there are *open-source* projects such as **OpenBIOS**, mainly designed for educational purposes.

Sources: [73].

Figure 5.14: Top left: the initial boot screen. Top right: the game boot screen. Bottom: the PSX shell.

5.7.1 Reading Joypad Inputs via BIOS

The PSX BIOS simplifies controller interaction through built-in routines that read the joypad status synchronized with the VSync. These routines allow developers to avoid manual handling of interrupts, serial communication protocols, and decoding of packets sent by the controller.

```c
#include <libetc.h>
#include <libgte.h>
#include <libgpu.h>
#include <stdio.h>

int main() {
    // Initialize the joypad management system
    InitPAD(0, 0, 0, 0); // Configure the controller interface
    StartPAD();          // Start communication with the joypad

    while (1) {
        // Read the state of the first controller
        unsigned short padState = PadRead(0);

        // Check which buttons have been pressed
        if (padState & PAD_UP) {
            printf("UP button pressed!\n"); // Check if the UP button is
                pressed
        }
        if (padState & PAD_DOWN) {
            printf("DOWN button pressed!\n"); // Check if the DOWN button
                is pressed
        }
        if (padState & PAD_CROSS) {
            printf("CROSS button pressed!\n"); // Check if the CROSS
                button is pressed
        }

        // Synchronize execution with VSync to avoid inconsistent reads
        VSync(0);
    }

    return 0;
}
```

Codice 5.10: Example of reading PSX controller inputs.

5.8 Reading the CD-ROM

5.8.1 CD-ROM

The Sony PlayStation was among the first consoles of its generation, alongside the Sega Saturn, to use CD-ROMs as the primary medium for storing game resources such as images, textures, 3D models, audio, and video files. The choice of an optical medium was necessary to overcome the limitations of the PSX's RAM (2 MB). In contrast, a CD-ROM could hold up to 71 minutes / 620 MB of data, providing sufficient space for more elaborate content.

This choice attracted numerous game developers of the era, who preferred to create games for the PlayStation rather than for the Nintendo 64. A notable example was "Final Fantasy VII," developed for PSX due to the space limitations and high costs of Nintendo 64 cartridges. Until then, the saga had been released exclusively on Nintendo consoles. Sources: [118].

A CD-ROM stores its data in a continuous spiral divided into sectors and tracks. Inside, there is also a Table of Contents (TOC) which defines the position of the tracks and their organization within the sectors. Unlike a hard drive, which uses magnetization to represent binary data, CDs operate through an optical disc. Specifically, a laser beam is used to read the information on the disc. When the surface reflects the laser, a "1" is read; if the surface is "scratched" (non-reflective), a "0" is read.

The data stored follows a file system standard called ISO-9660. This format is the only supported and compatible one with the PSX, which expects to find files organized in a directory-like structure with files, folders, and subfolders.

Each sector must be 2048 bytes long (or a multiple thereof). Sectors are organized into tracks (audio or data), which in turn are organized into sessions. Typically, PlayStation CD-ROMs contain a single data track.

To create a PSX-compatible CD image, several tools can be used, some provided directly by the official Sony PlayStation development kit. The main tools used for this thesis were:

- **BUILDCD.EXE**: Generates an image file (`GAME.IMG`) from a configuration file called `CDLAYOUT.CTI`. This latter is a structured file, similar to XML, which must be created manually. It contains information such as folder hierarchy, file names, the CD image name, and other organizational details.
- **PSXLICENSE.EXE**: Adds the official license file `LICENSEE.DAT` to the CD image to comply with Sony's requirements.
- **STRIPISO.EXE**: Converts the `GAME.IMG` file generated by `BUILDCD.EXE` into a standard ISO file (`GAME.ISO`).

To simplify the ISO creation process, the following batch script was used:

```
@ECHO OFF

ECHO Compiling and generating executable...
psymake

ECHO Building IMG file...
BUILDCD -l -i GAME.IMG CDLAYOUT.CTI

ECHO Converting GAME.IMG to GAME.ISO...
STRIPISO S 2352 GAME.IMG GAME.ISO

ECHO Bundling the license into the CD ISO...
PSXLICENSE /eu /i GAME.ISO

ECHO ISO file built successfully!
```

Codice 5.11: BUILDISO.BAT

The executables `BUILDCD.EXE` and `STRIPISO.EXE` are included in the PSY-Q package, located in `\psyq\cdemu\bin`. However, `STRIPISO.EXE` may be incompatible with some operating systems, including Windows XP. Alternatively, a remake called *The Revenge of StripISO* is available on the PSXDev forum.

Another useful tool is `PSXLICENSE.EXE`, also developed by the PSXDev community, which allows including license files in ISO images.

Furthermore, the official PlayStation development kit includes software called *CDGen*, developed by Sony. This tool can create CTI files, add licenses, and even burn discs. However, it was not used in this study.

Sources: [119][120][121][122].

An example of a CDLAYOUT.CTI file can be:

```
Disc CDROMXA_PSX
    CatalogNumber 0000000000000
    LeadIn XA
        Empty 1000
        PostGap 150
    EndTrack
    Track XA
        Pause 150
        Volume ISO9660
            SystemArea .\LCNSFILE\LICENSEE.DAT
            PrimaryVolume
                SystemIdentifier "PLAYSTATION"
                VolumeIdentifier "Game"
                VolumeSetIdentifier "Game"
                PublisherIdentifier "SCEE"
                DataPreparerIdentifier "SONY"
                ApplicationIdentifier "PLAYSTATION"
                LPath
                OptionalLpath
                MPath
                OptionalMpath
                Hierarchy
                    XAFileAttributes Form1 Audio
                    XAVideoAttributes ApplicationSpecific
                    XAAudioAttributes ADPCM_C Stereo
                    File SYSTEM.CNF
                        XAFileAttributes Form1 Data
                        Source [.]\SYSTEM.TXT
                    EndFile
                    File MAIN.EXE
                        XAFileAttributes Form1 Data
                        Source [.]\MAIN.EXE
                    EndFile
                EndHierarchy
            EndPrimaryVolume
        EndVolume
        Empty 300
        PostGap 150
    EndTrack
    LeadOut XA
        Empty 150
    EndTrack
EndDisc
```

Codice 5.12: CDLAYOUT.CTI

In conclusion, every disc contains a `SYSTEM.CNF` file. This file contains the boot information in ASCII/TXT format and can be considered similar to `AUTOEXEC.BAT` or `CONFIG.SYS` files from older MS-DOS systems.

The boot information included is:

- `BOOT=cdrom:\MAIN.EXE`: indicates the main executable to launch.
- `TCB=4`: specifies the number of thread control blocks.
- `EVENT=10`: provides information about event control blocks.
- `STACK=801FFFF0`: defines the initial address for the stack pointer.

In particular, the `STACK` line sets the initial value of the `SP` register (used for C language function calls). During console boot, the BIOS reads this value from the `SYSTEM.CNF` file and loads it into the register before starting the ROM. Sources: [123].

5.8.2 The Unique Design of PSX CDs

PSX discs feature a distinctive black-colored underside, a characteristic feature of the console. This choice by Sony was mainly for marketing purposes, giving the media a unique and recognizable look. Moreover, as explained in a video documentary included in one of the PSX demos, the black color was believed to act as a deterrent against piracy. However, from a technical standpoint, these discs did not differ significantly from standard CD-ROMs. Sources: [124].

Figure 5.15: Backside of an original PSX CD-ROM

5.8.3 Types of PlayStation Files

As mentioned in this chapter, a CD contains various types of files. Below is a brief description of the supported file types. Sources: [125].

Streaming Audio and Video Data

- **STR (Streaming Data)**: Format used for streaming data from the CD-ROM, generally for animations and audio data. Allows sequential reading to ensure continuous playback.
- **XA (CD-ROM Voice Data)**: Format for extended ADPCM audio, mainly used for interleaved audio with video or general data content.
- **BS (MDEC Bitstream Data)**: Compressed data for macroblock-based animation, used with the MDEC unit for video decoding.

3D Graphics

- **RSD (3D Model Data)**: Format for 3D models containing vertex and texture information.
- **TMD (Modeling Data for OS Library)**: Format used to represent modeling data structured for the OS library.
- **PMD (High-Speed Modeling Data)**: An optimized format for fast modeling.
- **HMD (Hierarchical Model Data)**: Complex format integrating hierarchical models with animation data.

2D Graphics

- **TIM (Screen Image Data)**: Format for static images intended for rendering.
- **SDF (Sprite Editor Project File)**: Project file for creating sprites via a dedicated editor.
- **CLT (Palette Data)**: Contains color palette information.
- **ANM (Animation Information)**: File storing data related to 2D animations.

Sound

- **SEQ (PS Sequence Data)**: Data for audio sequences compatible with the PlayStation.
- **VAG (PS Single Waveform Data)**: Format representing a single audio waveform.
- **VAB (PS Sound Source Data)**: Contains multiple audio data in a format optimized for the console.
- **DA (CD-DA Data)**: Audio format derived from standard CDs.

PDA and Memory Card

- **FAT (Memory Card File System Specification)**: Specifies the file system used on Memory Cards, organizing data into blocks and sectors for save data management.

5.8.4 Function to Read Binary Data from the Disc

```
char *FileRead(char *filename, u_long *length) {
    CdlFILE filePosition;
    int sectorCount;
    char *buffer;

    buffer = NULL;

    if (CdSearchFile(&filePosition, filename) == NULL) {
        printf("File %s not found!", filename);
    } else {
        // Calculate the number of sectors to read from the file
        sectorCount = (filePosition.size + 2047) / 2048;

        // Allocate a buffer for the file (must be a multiple of 2048)
        buffer = (char *)malloc(2048 * sectorCount);

        // Set the read position on the CD
        CdControl(CdlSetloc, (u_char *)&filePosition.pos, 0);

        // Start reading from the CD
        CdRead(sectorCount, (u_long *)buffer, CdlModeSpeed);

        // Wait for the read to complete
        CdReadSync(0, 0);

        *length = filePosition.size;
    }

    return buffer;
}

long length;
char *byteBuffer;

byteBuffer = (char *)FileRead("ESEMPIO.BIN;1", &length);
```

Codice 5.13: Example of a C function to read binary files from the disc

5.8.5 Anti-Piracy Mechanisms

As described at the beginning of this chapter, PSX CDs are read by a laser that detects irregularities in the disc's tracks. Conventional discs have minor fluctuations in their tracks that do not affect reading, as the laser can automatically calibrate itself.

Sony based its primary protection system on this principle by using a technology called *Wobble Groove*. This system involved engraving the TOC onto the disc (during mastering) with a specific frequency located in the inner section of the CD (also known as the *Lead-In* area). Moreover, this information was repeated multiple times to increase error tolerance.

The identifying string within the TOC varied depending on the distribution region:

- **SCEA:** Sony Computer Entertainment of America.
- **SCEE:** Sony Computer Entertainment of Europe.
- **SCEI:** Sony Computer Entertainment of Japan.

This technique not only protected discs from unauthorized duplication but also implemented regional locking. Unfortunately, this check could be bypassed in various ways. Since the verification was performed only during boot, it was possible to swap the original disc with a copy immediately after the initial check. This method was effective but risky, as it could severely damage the optical drive. To counteract this practice, some games re-initialized the drive during gameplay to repeat the check.

A second common technique at the time was installing a *modchip* capable of simulating the wobble signal, thus allowing the reading of unauthorized copies. For the development of this project, a console with this hardware modification was used purely for study purposes, to test demos also on real hardware in the absence of a developer console. Later, Sony introduced a library called **Libcrypt** to further strengthen protection.

A noteworthy mention is the anti-piracy system employed in the video game *Metal Gear Solid*, developed by Konami. During one of the key stages, the player was required to enter a radio frequency found only on the back of the original game case. This clever anti-piracy measure lost effectiveness due to word-of-mouth and the rise of the Internet. Sources: [73].

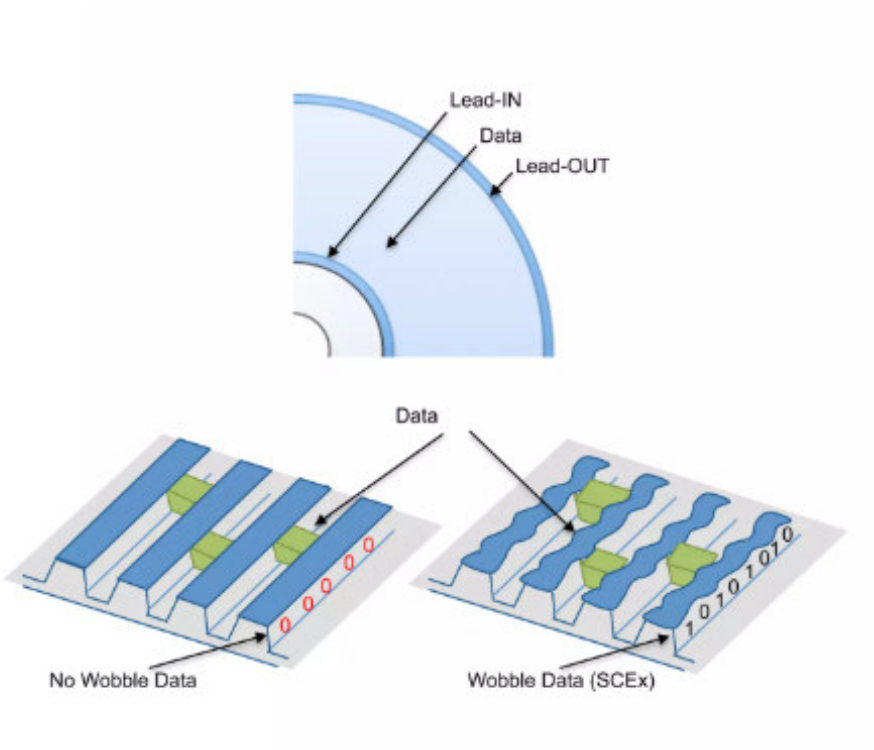


Figure 5.16: Illustration of the Wobble Groove concept [53]

Figure 5.17: Back of the Metal Gear Solid game case, containing the radio frequency required by the game

5.9 Textures

The concept of realism in software is closely related to the complexity of the geometric model. Although the PlayStation was capable of rendering a high number of polygons per second, this ability was insufficient to represent more complex details such as natural surfaces like clouds, rocks, grass, or skin. This concept does not only apply to the PSX but to any console. Sources: [72][73][126][127][128][129].

5.9.1 Foundations of Texture Mapping

The concept of Texture Mapping was developed specifically to avoid detailed modeling of every element. This concept assumes that a geometric object is composed of one or more fragments, each representing one or more pixels.

Mapping is implemented in the final stage of the rendering pipeline, where a hypothetical ”Fragment Shader” alters the attributes of polygons to simulate the required details.

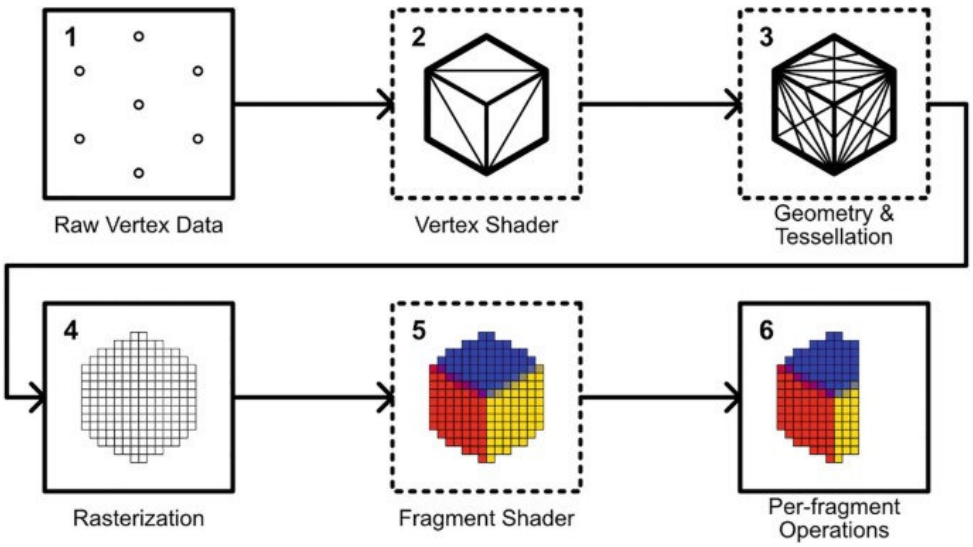


Figure 5.18: Rendering Pipeline
[54]

Among the main mapping methods are:

- **Texture Mapping**: Uses images to fill the surfaces of polygons
- **Environment Mapping** (or Reflection Mapping): exploits images of the surrounding environment to simulate highly reflective surfaces
- **Bump Mapping**: modifies normals during rendering to emulate surface three-dimensionality

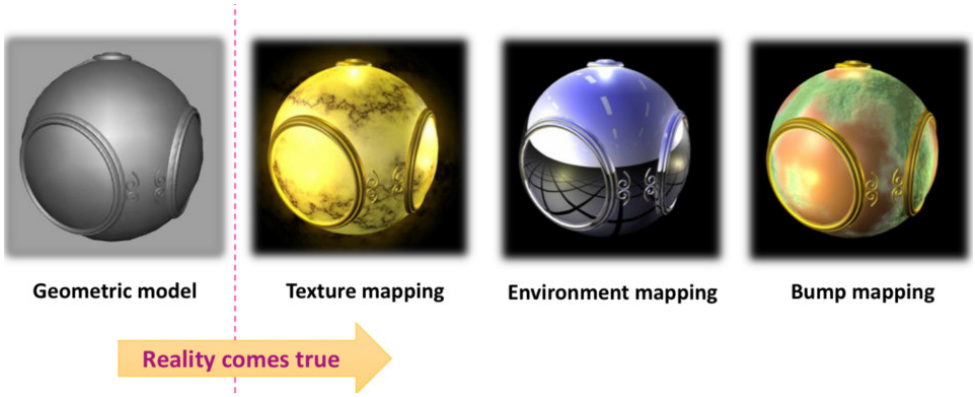


Figure 5.19: Types of ”Mapping”

This chapter will focus mainly on the first point. Thanks to this process, an image is associated with each geometric surface. The ”Textures” used are composed of a set of ”Texels” (Texture elements or Texture pixels), which determine the resolution of an image. The width and height of textures must be powers of two and must be loaded into VRAM before they can be used.

Thanks to Texture Mapping, it is possible to recreate realistic effects using simple geometric models, significantly reducing the computational cost compared to detailed simulation of every element. Moreover, changing the visual appearance of an object becomes much easier because it is sufficient to change the texture associated with it.

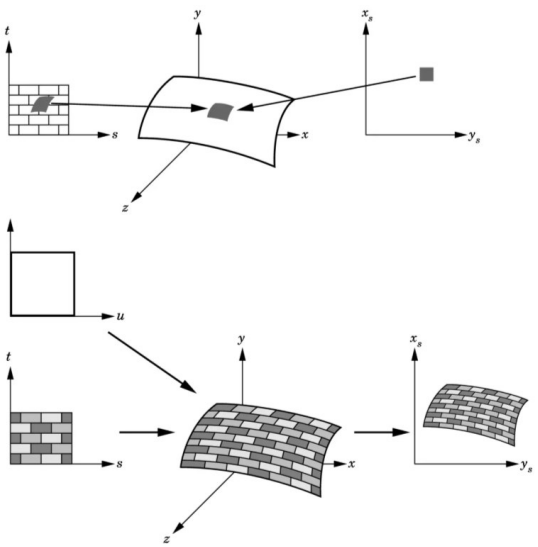


Figure 5.20: Mapping a 2D texture onto a 3D surface using ”texture coordinates”. [55]

Figure 5.21: Resident Evil 2 characters with texture variations for different outfits. [56]

The Mapping process involves associating the texels of the texture with the ”fragments” of the geometric model. For each pixel, it is necessary to identify the corresponding point on the object’s surface. Conversely, given a point on the object, the corresponding point on the texture must be determined.

With the concept of Direct Mapping, each vertex of the object must be associated with a specific portion of the texture. Texture coordinates indicating which portion to use are assigned to each vertex. Subsequently, fragment interpolation fills the other portions of the object. Usually, texture coordinates range between 0 and 1, where $(0, 0)$ represents the bottom-left corner and $(1, 1)$ the top-right corner.

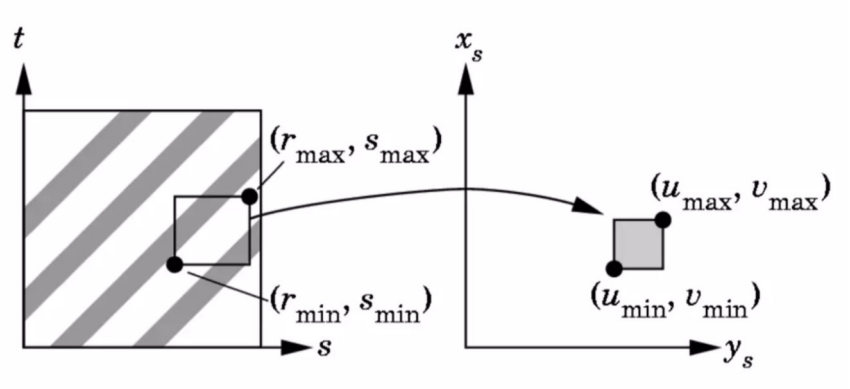


Figure 5.22: Region-based mapping from texture to screen coordinates with ”UV boundaries”.

[55]

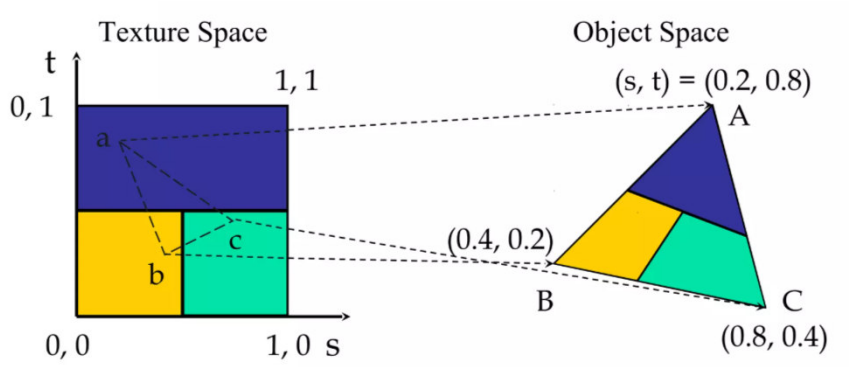


Figure 5.23: Mapping from texture space to object space, showing the correspondence between texels and vertices.

[55]

In the case of the PSX, textures are loaded into VRAM along with the display and drawing

buffers (making sure not to overwrite them).

The traditional method of mapping textures to the vertices of primitives is based on UV coordinates, which represent specific points of the original texture. The GPU then interpolates these values and draws them pixel by pixel on the screen. Besides UV coordinates, PlayStation uses two additional elements: T-PAGES and CLUTs.

5.9.2 Concept of T-PAGE

The term **T-PAGE** ("Texture Page") is based on the organization of VRAM into multiple pages. This structure resembles a grid where each cell represents a specific page. In the case of the **PSX**, VRAM is divided into a grid consisting of 16×2 cells, each measuring 64×256 pixels at 16-bit color depth.

The actual coordinates of a Texture Page are expressed as XY pairs, while the UV coordinates of a geometric primitive correspond to offsets relative to the edges of the Texture Page. The configuration of the latter acts as a reference point for textured primitives, which fetch texture values starting from the corner at position $(0, 0)$.

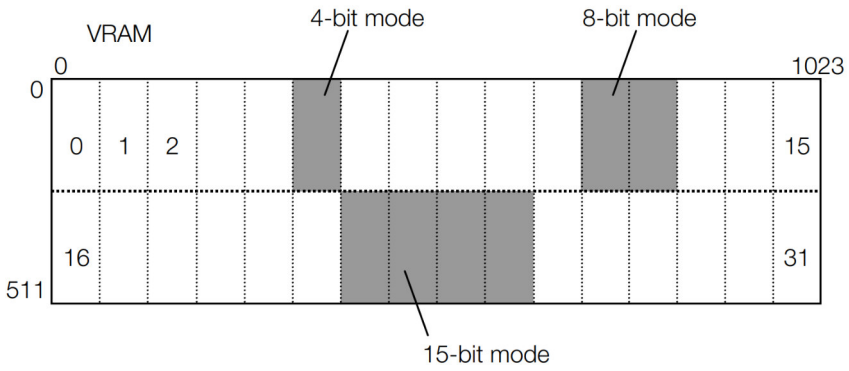


Figure 5.24: Structure of a Texture Page

An important aspect to consider concerns the size of 4- and 8-bit textures. Although the width of these images differs from that of the original image, the UVs do not require adjustments. UVs always represent the absolute value in pixels and respect the original image width (provided the color depth is correctly specified in the Texture Page).

However, a key limitation of this system is that UV coordinates can only take values between 0 and 255. For this reason, it is not possible to display a texture larger than 256×256 pixels, regardless of color depth. To manage larger textures, the image must be divided into multiple primitives that together represent the total area of the image.

5.9.3 Concept of CLUT

The term *CLUT*, or *Color Lookup Table*, refers to a table containing the colors used to represent image data in 4- and 8-bit modes. Each pixel of an image acts as an index into the *CLUT*, allowing each numeric value to be associated with the corresponding color in the table. It is organized in VRAM as a 256×1 image for 8-bit textures and 16×1 for 4-bit textures.

Textures used by the PSX support color depths of 4, 8, and 16 bits per pixel. Since the PSX video memory addresses pixels in 16-bit *WORD*s, 4- and 8-bit textures occupy one-quarter and one-half, respectively, of the effective width of the original image. For this reason, these two types of textures almost always require a *CLUT* as a reference.

Thanks to its structure, it is possible to store a significant number of colors while keeping texture size limited.

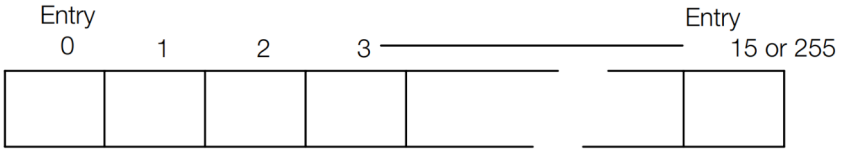


Figure 5.25: Structure of a CLUT

5.9.4 Insight on the TIM format

Sony PlayStation manages textures primarily through the TIM file format. It is possible to obtain this format using the software *"TIM Tool"* (included in Sony's *Developers Tools CD*). This software allows generating TIM files from images in BMP, JPG, PNG, and other formats.

The TIM format is designed to contain not only pixel data but also the XY coordinates of the image within VRAM, as well as information about the CLUT.

As shown in the screenshot in Figure 5.26, the rectangle located in the upper-left corner corresponds to the frame buffer. Inside this area, the green box corresponds to the drawing buffer while the yellow box represents the display buffer. On the right side of the application, the selected Texture Page is visible, containing both the texture and the CLUT.

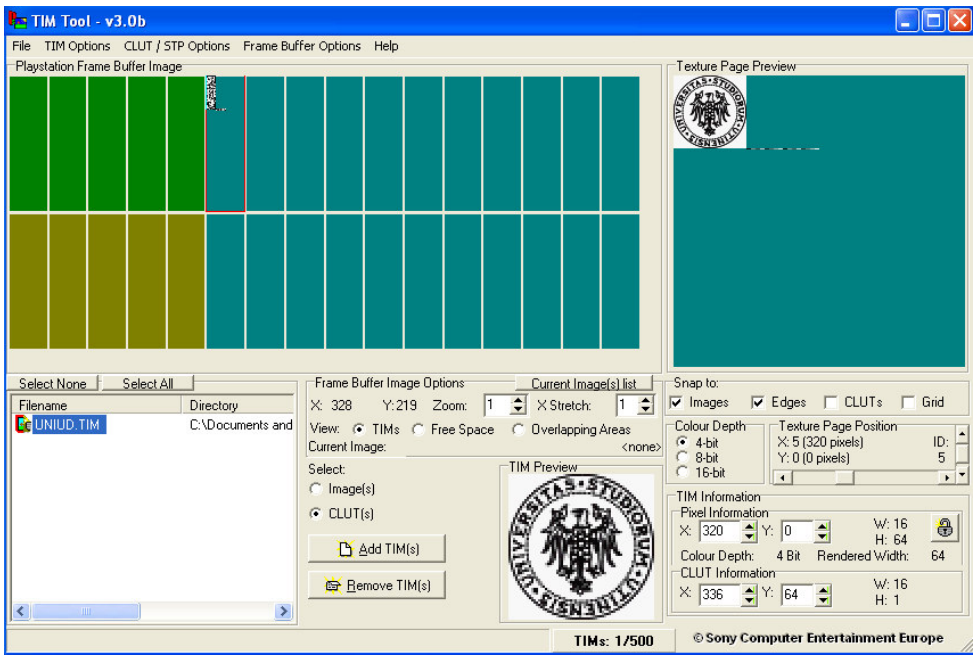


Figure 5.26: Main screen of the ”TIM Tool” software

5.9.5 PSX Graphic Artifacts

The visual artifacts associated with the PlayStation have over time become a sort of Visual Identity of the console. Paradoxically, some modern games artificially reproduce these imperfections to give them a nostalgic and retro look.

PlayStation Wobbling

This type of artifact, known as ”wobbling,” occurs due to the way PSX handles Texture Mapping. To understand this phenomenon, it is useful to introduce the following three examples of textures:

- **Flat Texture**: A simple 2D reference texture
- **Affine Texture Mapping**: Represents the actual result obtained by the PSX. When transformations such as rotations and translations are applied, the textures undergo noticeable distortion.
- **Perspective-Corrected Texture Mapping**: Represents the ideal result where textures are applied correctly respecting perspective.

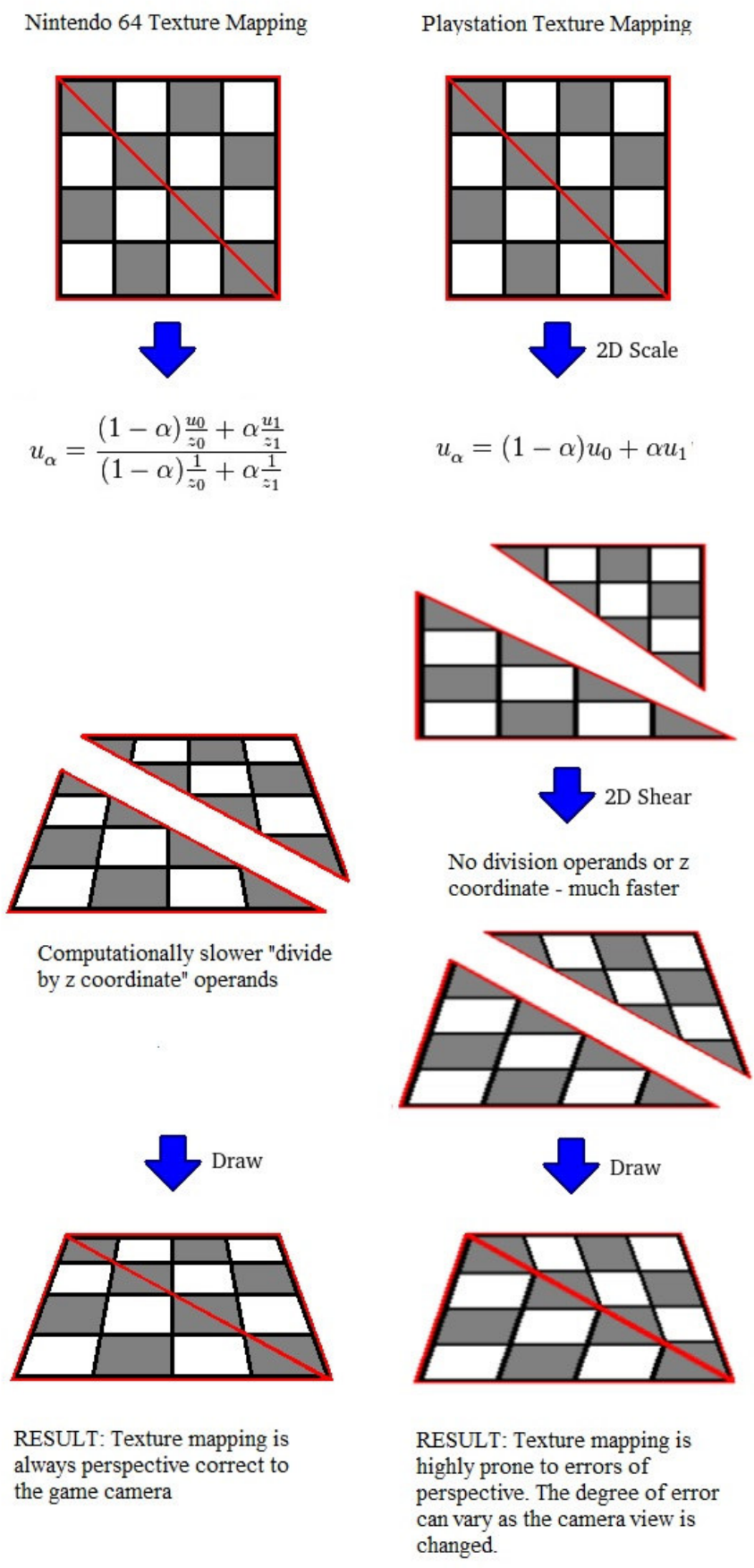


Figure 5.27: N64 vs PSX Texture Mapping

[57]

The main cause of this issue lies in the nature of the PSX GPU. Being a 2D drawing engine, it does not operate directly in a three-dimensional space and cannot handle depth (Z-axis). Therefore, the information that the GTE sends to the GPU is limited to XY coordinates, without any reference to the Z component. This leads to a series of texture distortions when applied to 3D objects.

A technique used to reduce this phenomenon is ”Tessellation,” i.e., subdividing primitives into smaller triangles. Thanks to this technique, the visual effect becomes less noticeable since the surface area of the distortion-affected regions is reduced.

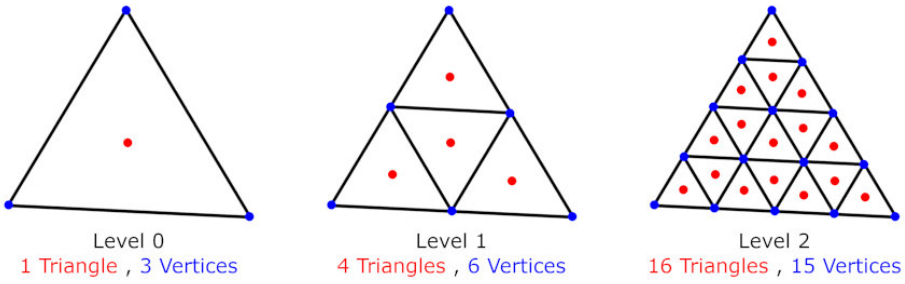


Figure 5.28: Example of Tessellation: Triangles are progressively subdivided into smaller triangles to increase definition.

[58]

Another problem related to the lack of a native Z-buffer is the possible polygon overlap (Z-fighting, discussed in chapter 5.5).

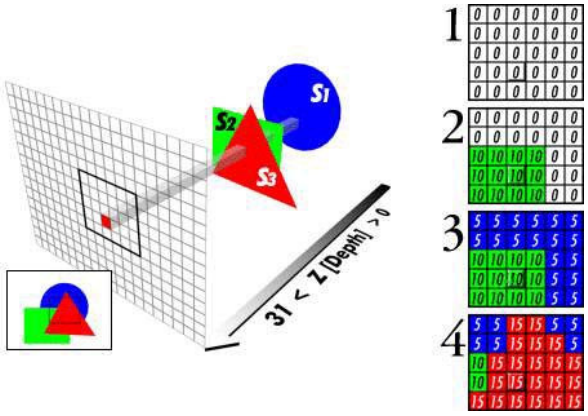


Figure 5.29: Example of Z-Fight: Polygons overlap due to lack of depth precision (Z-buffer).

[59]

Polygon Jittering

Another characteristic visual artifact of the PSX is ”Polygon Jittering.” This phenomenon does not directly involve textures but is related to the process of drawing polygons on the screen by the

GPU. The main cause of this problem is the absence of sub-pixel rasterization in the console's GPU. The XY coordinates used during rendering are treated as integer values without support for fractional values.

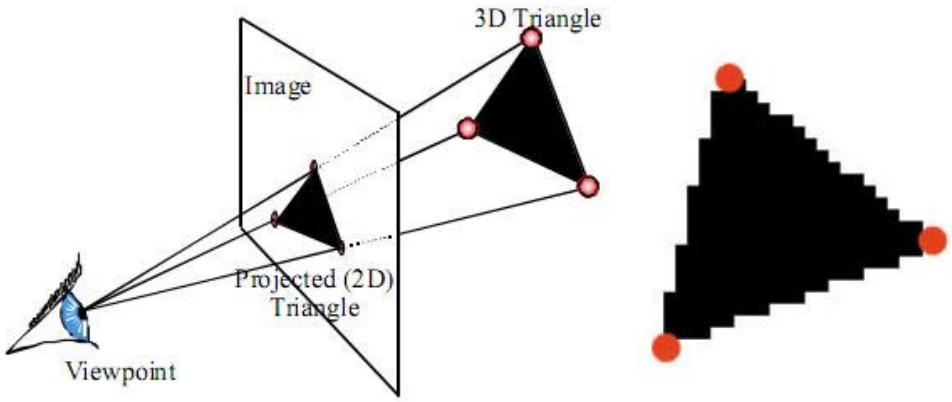


Figure 5.30: Projection of a 3D triangle onto 2D space, highlighting 'Polygon Jittering' caused by lack of sub-pixel accuracy

[59]

Consequently, screen coordinates are rounded up or down, snapping to the "pixel grid." This approach causes a visual artifact known as "snap to grid," where discrepancies in vertex positions during rendering become evident. The problem is further amplified by the lack of an FPU unit.

Lack of Mip Mapping

Sony PlayStation did not implement mip mapping techniques, a method where each texture is scaled and filtered at different resolutions based on the object's distance from the camera. This technique is mainly used to reduce aliasing effects on screen and to improve console performance, avoiding wasting VRAM space with "high-definition" textures for objects far from the camera.

A criticism often raised against Mip Mapping is that distant objects on the screen appear blurry. However, when viewed on a CRT, the PSX does not suffer from this problem and appears visually very sharp. The issue becomes more apparent on high-definition screens, where its graphics look very pixelated.

The Nintendo 64, unlike its competitor, implements mip mapping, accurate perspective calculations, and a true Z-buffer.

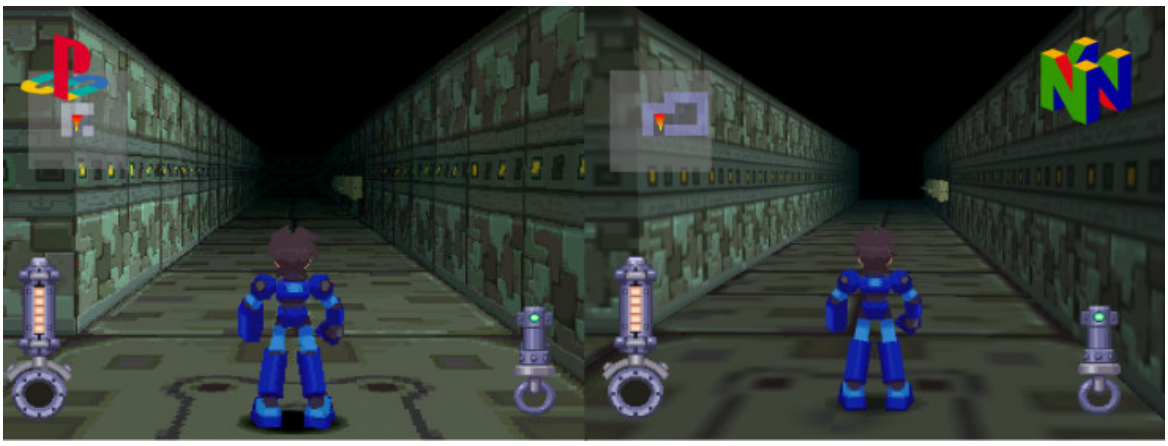


Figure 5.31: Mega Man Legends game comparison between PSX and N64.
[60]

5.10 Audio

As discussed in the chapter dedicated to the PlayStation Sound Processing Unit (SPU), it is possible to send audio samples to the SPU for playback as independent voices. The SPU can handle multiple voices simultaneously, and these must necessarily be stored in ADPCM format. Sources: [112] [130][131][132][133].

5.10.1 Types of ADPCM formats

The SPU features 24 hardware voices. Each of these voices can be used to play audio data, generate noise, or act as a modulator for another voice. Every voice has a programmable ADSR (Attack, Decay, Sustain, Release) filter and independent volume control for the left/right audio channels.

Sony PlayStation supports two main types of ADPCM format:

- **XA-ADPCM:**
  - Data is decompressed directly by the CD-ROM controller.
  - Samples are sent to the audio mixer without requiring the use of the SPU's DRAM.
  - Does not support sample looping.
  - Offers only two sample rate options (22,050 Hz and 44,100 Hz).
- **SPU-ADPCM:**
  - Supports sample looping and reverb effects.
  - Requires the SPU's DRAM to store samples.

  - Uses VAG and VAB file formats for playback of short “sound effects.”

Although the decompression algorithm is the same for both formats, in the case of XA-ADPCM decompression is handled directly by the CD-ROM controller without impacting SPU resources.

The ADPCM format

To better understand the ADPCM format, it is useful to first introduce the concept of PCM (Pulse Code Modulation). This method is based on uncompressed recording of the audio signal, sampling and digitally storing it without further processing. Common examples of PCM audio are WAV files.

PCM:

- The analog audio signal is sampled at regular intervals.
- Each sample is quantized to the nearest available value (chosen from a predefined range of discrete values).
- PCM files use 16 bits per sample.

As introduced at the beginning of the chapter, PSX uses a compressed version called ADPCM (Adaptive Differential Pulse Code Modulation). This method encodes audio samples by reducing their size while maintaining a balance between quality and compression.

ADPCM:

- Reduces audio samples to 4 bits per sample.
- Uses prediction to estimate the next amplitude based on the difference from previous values.
- Used by SPU-ADPCM and stored in the SPU’s DRAM.

5.10.2 Details on VAG and XA formats

Converting a WAV file to VAG

The PSX audio format does not allow direct playback of WAV files in the disc data track. To overcome this limitation, it is possible to convert audio files into XA or VAG formats. This process can be done using the programs ”VAGEdit” for converting WAV to VAG, and ”Movie Converter” for converting WAV to XA (both tools are provided in the PSYQ library).

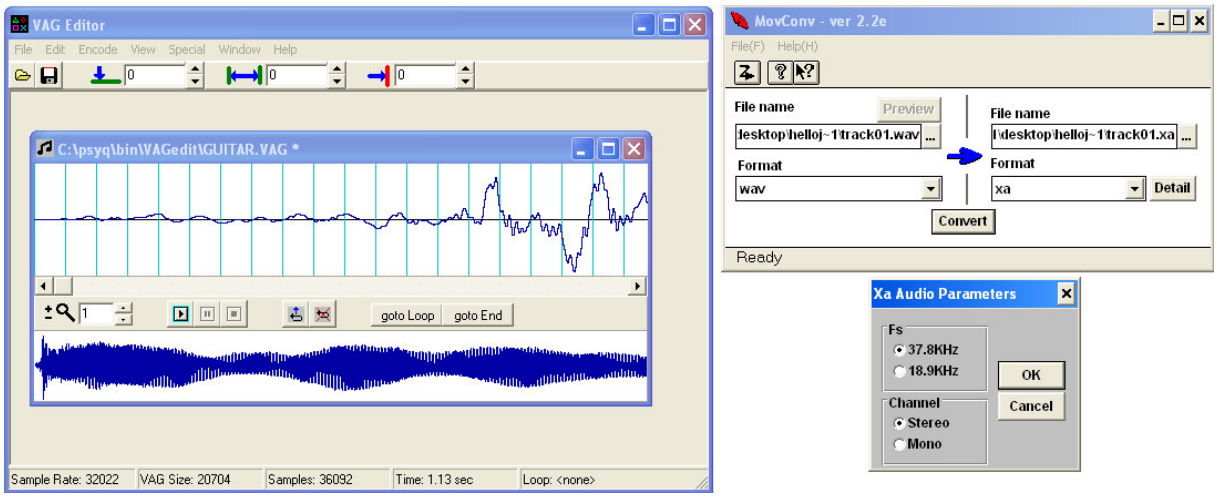


Figure 5.32: On the left, the ”VAGEdit” software. On the right, ”Movie Converter”.

VAG Header Information

The header of a VAG file contains essential information for the correct interpretation and playback of audio samples. Below is its typical structure:

- **4 bytes**: File identifier (e.g., ”VAGp”).
- **4 bytes**: Format version.
- **4 bytes**: Reserved field.
- **4 bytes**: File size (in bytes).
- **4 bytes**: Sampling frequency (e.g., 22,050 Hz).
- **12 bytes**: Reserved field.
- **16 bytes**: File name.
- **Audio data**: Audio samples compressed in ADPCM format.

To determine the sampling frequency of a VAG file, one can analyze the header using a hex editor. This information is crucial for correctly setting elements like pitch during sample playback.

Additionally, it is important to note that the endianness of the fields in the header is typically Big-Endian, although exceptions may exist depending on implementations.

XA Header Information

- **4 bytes**: File identifier.
- **4 bytes**: Total file size (in bytes).

- **2 bytes**: Sampling frequency (e.g., 18,900 Hz or 37,800 Hz).
- **1 byte**: Audio channel (mono or stereo).
- **1 byte**: Compression information (ADPCM algorithm).
- **Audio data**: Compressed audio samples.

5.10.3 Management of Audio Tracks on CD-ROMs

Sony PlayStation can utilize the capabilities of CD-ROMs to contain multiple tracks for playback of music and soundtracks. Tracks, which can be data or audio, allow for two distinct approaches to integrate audio in a project:

Conversion and insertion into the data track

Audio tracks in WAV format converted into XA or VAG formats can be loaded directly into the disc's data track. This approach is considered particularly useful for managing short-duration audio tracks.

Creation of new audio tracks

It is possible to add independent audio tracks on the CD-ROM, which coexist alongside the data track. This method allows for a clear separation of game data from the soundtrack, providing greater flexibility.

To create new tracks, it is necessary to intervene directly in the disc generation process. In this analysis, the program `MKPSXISO`, developed by user Lameguy64, was chosen to generate the required ISO file.

Specifically, it is possible to modify the BATCH script shown in chapter 5.8.1 by replacing the following command:

```
ECHO Building IMG file...
BUILDCD -l -i GAME.IMG CDLAYOUT.CTI
```

with the following amateur software:

```
ECHO Building IMG file...
MKPSXISO CDLAYOUT.XML
```

This program also requires a configuration file to define the CD structure, this time in XML format.

```
<iso_project image_name="GAME.ISO" cue_sheet="GAME.CUE" no_xa="0">
    <track type="data">
        <identifiers system="PLAYSTATION" application="PLAYSTATION" volume=
           "MYDISC"
                    volume_set="GAME" publisher="ME" data_preparer="
                       MKPSXISO"
                    copyright="COPYLEFT"/>
        <license file="LCNSFILE\LICENSEE.DAT"/>
        <directory_tree>
            <file name="SYSTEM.CNF" type="data" source="SYSTEM.TXT"/>
            <file name="MAIN.EXE" type="data" source="MAIN.EXE"/>
            <file name="01.PRM" type="data" source=".\ASSETS\COMMON\01.PRM"
               />
            <file name="01.CMP" type="data" source=".\ASSETS\COMMON\01.CMP"
               />
            <file name="POWERUP.VAG" type="data" source=".\ASSETS\SOUND\
               POWERUP.VAG"/>
            <file name="YOULOOSE.VAG" type="data" source=".\ASSETS\SOUND\
               YOULOOSE.VAG"/>
            <file name="COUNTGO.VAG" type="data" source=".\ASSETS\SOUND\
               COUNTGO.VAG"/>
            <dummy sectors="1024"/>
        </directory_tree>
    </track>
    <track type="audio" source=".\ASSETS\MUSIC\TRACK_01.WAV"/>
    <track type="audio" source=".\ASSETS\MUSIC\TRACK_02.WAV"/>
    <track type="audio" source=".\ASSETS\MUSIC\TRACK_03.WAV"/>
</iso_project>
```

Codice 5.14: Example of XML file for generating an ISO image with audio and data tracks.

The CUE file

After executing the script, besides the ISO file, a CUE file is generated. This text file describes the organization of tracks present on the disc, specifying essential information regarding the type and position of each track.

The CUE file thus becomes the main reference point to be used together with the ISO file, as it contains all the necessary information for correctly interpreting the CD's structure.

```
    FILE "Tekken 3 (Europe) (Track 1).bin" BINARY
      TRACK 01 MODE2/2352
        INDEX 01 00:00:00
    FILE "Tekken 3 (Europe) (Track 2).bin" BINARY
      TRACK 02 AUDIO
        INDEX 00 00:00:00
        INDEX 01 00:02:00
    FILE "Tekken 3 (Europe) (Track 3).bin" BINARY
      TRACK 03 AUDIO
        INDEX 00 00:00:00
        INDEX 01 00:02:00
```

Codice 5.15: Original CUE file from the game ”Tekken 3” (Namco). [134]

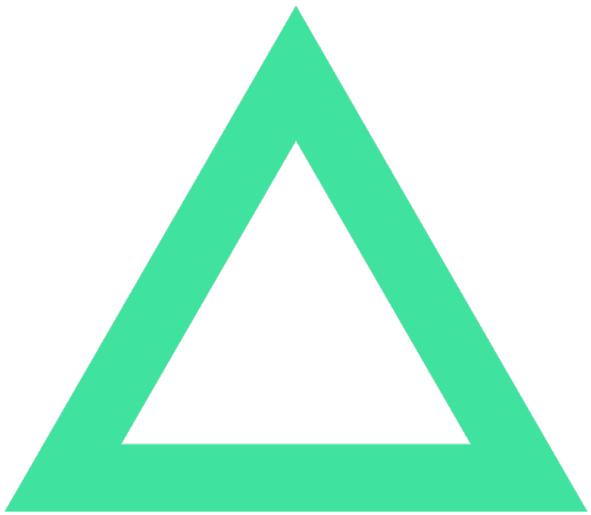
5.10.4 Example of audio implementation

```c
#include <libspu.h>
#include <libcd.h>
int main() {
    SpuCommonAttr attr; // Initialize the SPU system
    SpuSetTransferMode(SPU_TRANSFER_BY_DMA);
    attr.mask = SPU_COMMON_MVOLL | SPU_COMMON_MVOLR;
    attr.mvol.left = 0x3FFF; // Maximum volume left channel
    attr.mvol.right = 0x3FFF; // Maximum volume right channel
    SpuSetCommonAttr(&attr);
    if (CdInit() == 0) {    // Initialize the CD system
        printf("Error initializing CD-ROM.\n");
        return -1;
    }
    // Specify the track to play
    int tracknum = 2; // Audio track number
    int t[] = {tracknum, 0};
    // Play the specified track
    if (CdPlay(2, t, 0) == 0) {
        printf("Error playing track %d.\n", tracknum);
        return -1;
    }
    printf("Playing track %d...\n", tracknum);
    while (1) {}
    return 0;
}
```

Codice 5.16: Simple example of audio implementation in C

Part III

Creating Demos on PlayStation: A Technical and Creative Showcase



Chapter 6

Demo Disc One (Showcase of Various Demos)

The first tech demo developed in this analysis consists of a set of subprograms aimed at showcasing a specific feature or technological upgrade of the console. The main interface of this software presents eight functions, selectable from a menu where the user can navigate through the options using the controller's directional arrows. Once the desired function is highlighted, it can be launched by pressing the X button, while pressing SELECT returns to the main menu. By pressing the START button, the project credits can also be viewed. Each program emphasizes a different aspect, from primitive management for creating 3D elements to texture rendering, up to demonstrating special effects achievable by leveraging the console's resources. These features are presented gradually to provide an overview of the main programming and computer graphics concepts illustrated in the previous chapters. The demo was developed in the C language by integrating the libraries provided by the PSYQ SDK and was tested both on an emulator and on original hardware. The complete project is available on GitHub at the following address: [135].

6.1 Cube Transformations

The "Cube Transformation" function illustrates the transition to 3D graphics introduced in this console generation. The PSX was indeed among the first to extensively integrate 3D technology into the video game market, laying the foundation for a new level of immersion for gamers.

In the demo, the cube is represented by triangles with Gouraud shading. The user can rotate the model along the X, Y, and Z axes using the Triangle, Square, and Circle buttons respectively. Pressing the Cross button resets the cube to its original position.

It is also possible to modify the cube's scale in real time: the Up, Right, and Left directional

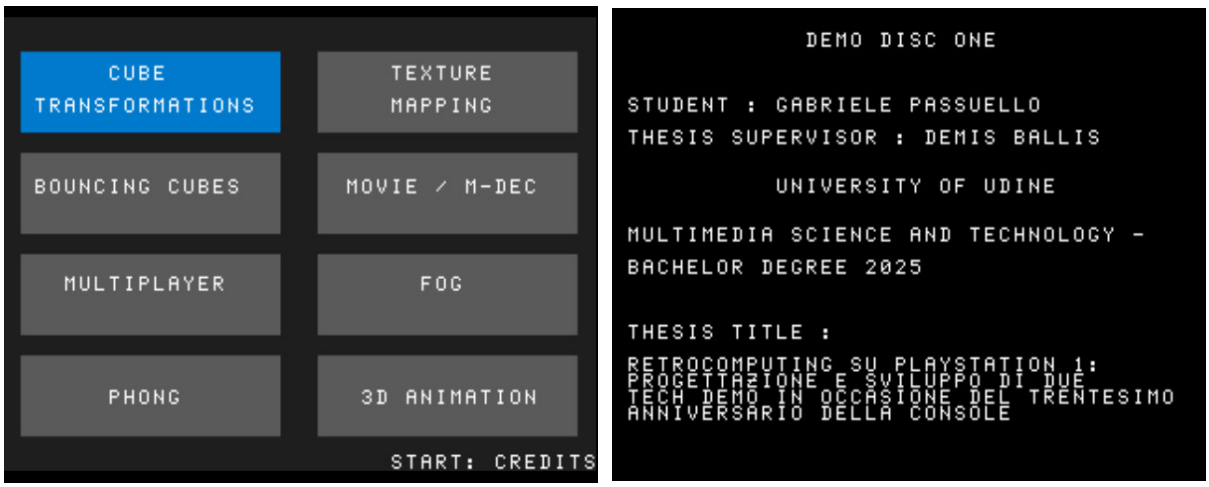


Figure 6.1: On the left, the menu. On the right, the project credits.

buttons change the scale of a single axis, while pressing the Down button restores the initial values. The cube can also be uniformly scaled up or down on all three axes using the shoulder buttons R1 and L2.

The numerical values for rotation and scale are constantly displayed on screen to allow real-time monitoring of the applied transformations.

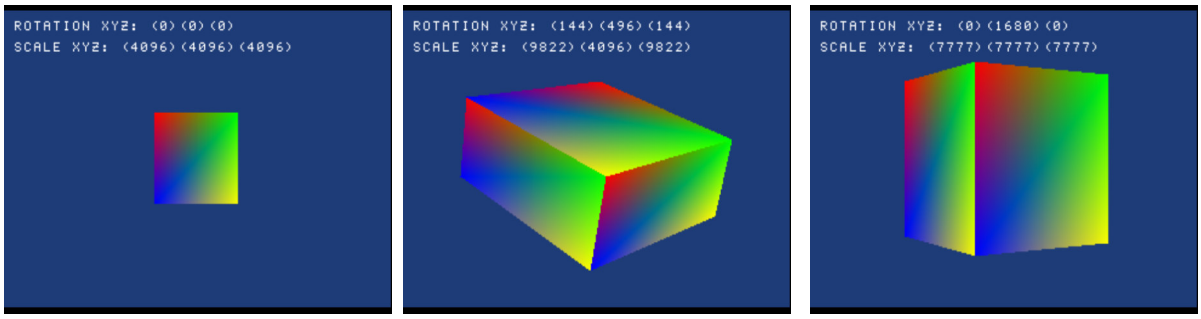


Figure 6.2: The ”Cube Transformations” function in execution.

6.2 Bouncing Cubes

The ”Bouncing Cubes” function initially displays a single cube bouncing off the edges of the screen. This demonstrates the console’s ability to simultaneously manage multiple moving 3D objects.

The user can interact with the program by pressing the UP button to increase the number of cubes, while pressing the DOWN button decreases the number. This allows testing the platform’s capability to maintain adequate performance even as graphical complexity grows. It is also possible to adjust the cubes’ scale using the Right (increase) and Left (decrease) buttons.

Each cube is composed of six faces, each rendered with two triangles, for a total of twelve polygons per cube on screen. The colors of each vertex are randomly assigned to make the

objects more visible on screen. Finally, the number of cubes and the applied scale are constantly updated on screen.

Figure 6.3: The ”Bouncing Cubes” function in execution.

6.3 Multiplayer

The ”Multiplayer” function simulates a four-section split screen with the aim of illustrating how local multiplayer could be managed on a single console. In each of the four screen areas, a simple colored square is drawn along with an identifying text label.

The program’s structure is based on dividing both the drawing buffer and the screen buffer into four distinct viewports, within which the corresponding content is rendered.

In a more advanced version of the program, each player connected to a different controller could independently control their own object, since each viewport corresponds to a separate 3D world, thus implementing a true local multiplayer system.

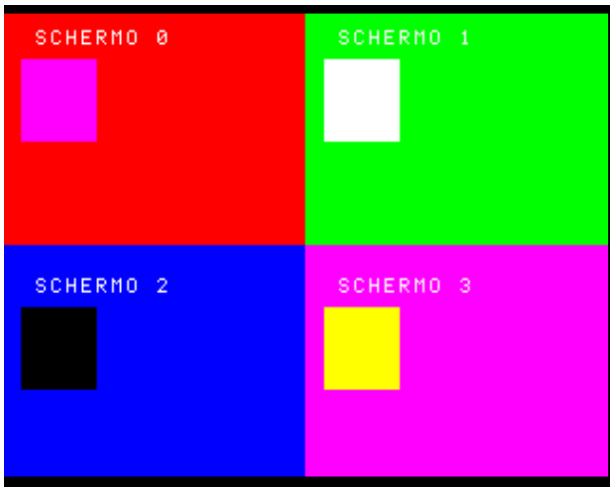


Figure 6.4: The ”Multiplayer” function in execution.

6.4 Texture Mapping

The ”Texture Mapping” function differs from the previous ones by loading the 3D model of a cube from a binary file stored on the CD, which defines its vertices, faces, and normals. The object is then rendered on screen and continuously rotated around the Y-axis, highlighting the console’s ability to apply different textures in real time.

The two textures, converted into .TIM format and loaded from the CD-ROM, can be selected during execution by pressing the Square and Triangle buttons (the first represents the University of Udine logo, while the second depicts a brick wall).

From a logical standpoint, the program first initializes the camera by loading both the 3D model and its related textures. At each iteration of the main loop, the cube’s position is updated, the perspective transformation is updated via the GTE, and primitives are added to the ordering table according to their depth. Subsequently, the PSX handles the on-screen rendering. The user can also use the directional arrows to move the camera.

This function aims to demonstrate how the console can assign textures to 3D objects, increasing realism and enhancing the level of detail in a game environment.

Figure 6.5: The ”Texture Mapping” function in execution.

6.5 Fog

The ”Fog” function demonstrates how distant objects can progressively fade into fog, both for aesthetic reasons and for hardware performance optimization. In the following program, three cubes are managed and rendered, initially positioned at fixed distances (Z) from the camera. The user can move the left cube with the Up/Down arrow keys, the central cube with the Left/Right arrow keys, and the right cube with the Triangle/Circle buttons. Pressing one of these keys moves the cube forward or backward relative to the viewpoint, while the Cross button resets the position of all objects.

From a graphical perspective, the program calculates a blend percentage between the cube’s original color and the ”fog” color (set to a light tone) based on the distance z from the viewpoint.

Below a certain threshold (the ”fogNear” value), objects retain almost all of their original color. Once this distance is exceeded, the radiance (or brightness) of the original color is gradually reduced, blending with the fog color until it completely disappears beyond the ”fogFar” value.

Theoretically, one way to describe homogeneous fog refers to the transport equation, which calculates the radiance $L(x)$ as a function of the medium's absorption and the distance traveled by light:

$$L(x) = e^{-\kappa_a |x-x_0|} L(x_0) + \left(1 - e^{-\kappa_a |x-x_0|}\right) L_e$$

where κ_a is the absorption coefficient of the medium, $L(x_0)$ is the initial radiance (the ”original” color), and L_e represents the background (or ambient) radiance. Practically, fog is often approximated by blending the object's color C_{in} with the fog color C_{fog} according to an attenuation factor f:

$$C = f\, C_{\text{in}} + (1 - f)\, C_{\text{fog}},$$

where f can be calculated as

$$f = e^{-(\text{density} \cdot z)},$$

with density controlling how quickly the color blends with the fog and z the object's distance from the observer. Moving beyond fogNear, f decreases, allowing the fog color to dominate; once past fogFar, the object is completely ”engulfed.” This mechanism simulates a continuous ”veil” effect between the object and the observer, improving visual immersion while helping to optimize performance by hiding distant details.

This technique was widely used in many titles of the era (for example in the game *Silent Hill*, Konami) to enhance immersion while optimizing performance since the console does not need to render details hidden by the fog. Sources: [136].

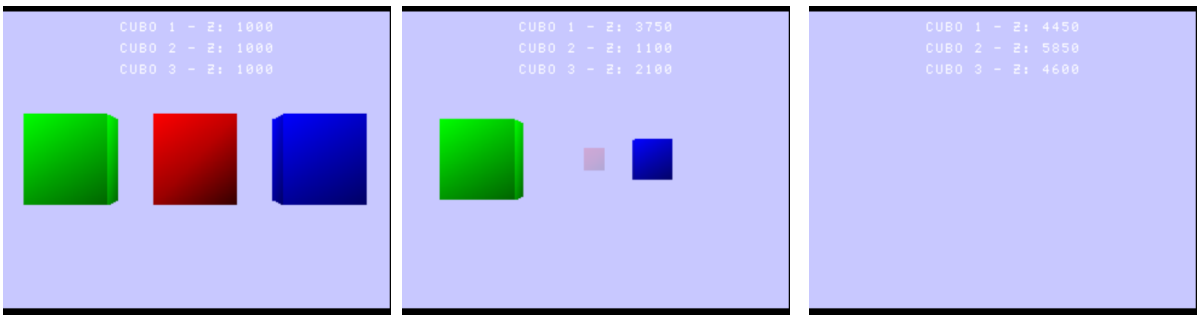


Figure 6.6: The ”Fog” function in execution.

6.6 Phong

The ”Phong” function represents the implementation of ”Phong Shading” on the PSX. This example demonstrates how to achieve advanced lighting effects despite the console lacking

hardware support for it. This algorithm is renowned for its ability to generate detailed lighting by computing specular reflection on a per-pixel basis.

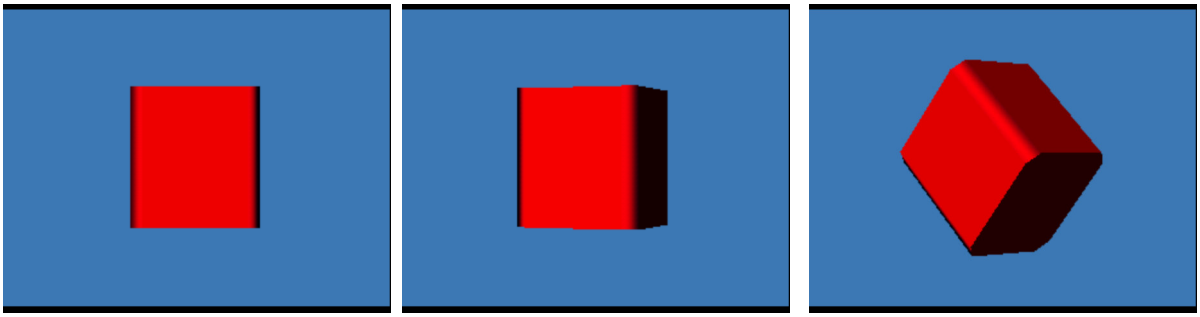


Figure 6.7: The ”Phong” function in execution.

The code included in this thesis is an adaptation of the demo known as ”Phong,” present in the PlayStation Developers Tool Kit sample list provided by Sony. This example was originally written by the developer Oka. The modifications made aim to contextualize the original example and optimize it for the purposes of this project.

Excerpt from the original sample:

```
/* $PSLibId: Run-time Library Release 4.3$ */
/*
 *    Phong Shading program
 *
 *    This program includes the function phong_tri, which performs Phong
 *    shading on one triangle. "Triangle color", "normal vectors of
    vertices",
 *    and "screen coordinates of vertices" are provided to phong_tri, which
 *    then performs Phong shading and renders the triangle.
 *    Since phong_tri renders directly to video memory, Z-sorting with other
 *    polygons requires rendering and re-texturing in a non-display area.
 *
 *         1995,3,29 by Oka
 *         Copyright  (C)  1995 by Sony Computer Entertainment
 *              All rights Reserved */
```

Codice 6.1: Excerpt from Oka’s code

The software in this analysis uses a software approach to simulate ”Phong Shading.” In the PSX context, this requires a significant computational load compared to ”Gouraud Shading.” The function `phong_tri` is responsible for computing the interpolated normal for each pixel of the triangle and applying the lighting model.

Unlike Flat and Gouraud Shading:

- **Flat Shading**: Uses a single normal per primitive (triangle) and applies a uniform color. The result has faceted surfaces.
- **Gouraud Shading**: Interpolates the colors of the triangle's vertices. This technique produces smoother transitions but does not effectively capture specular highlights. Supported by PSX.
- **Phong Shading**: Interpolates the vertex normals across each pixel and calculates lighting at the pixel level. It offers visually more realistic results but requires a high computational cost. Not natively supported by PSX.

To compute the lighting, each pixel uses the interpolated normal to calculate lighting by combining ambient, diffuse, and specular components, following the Phong reflection model. Sources: [137].

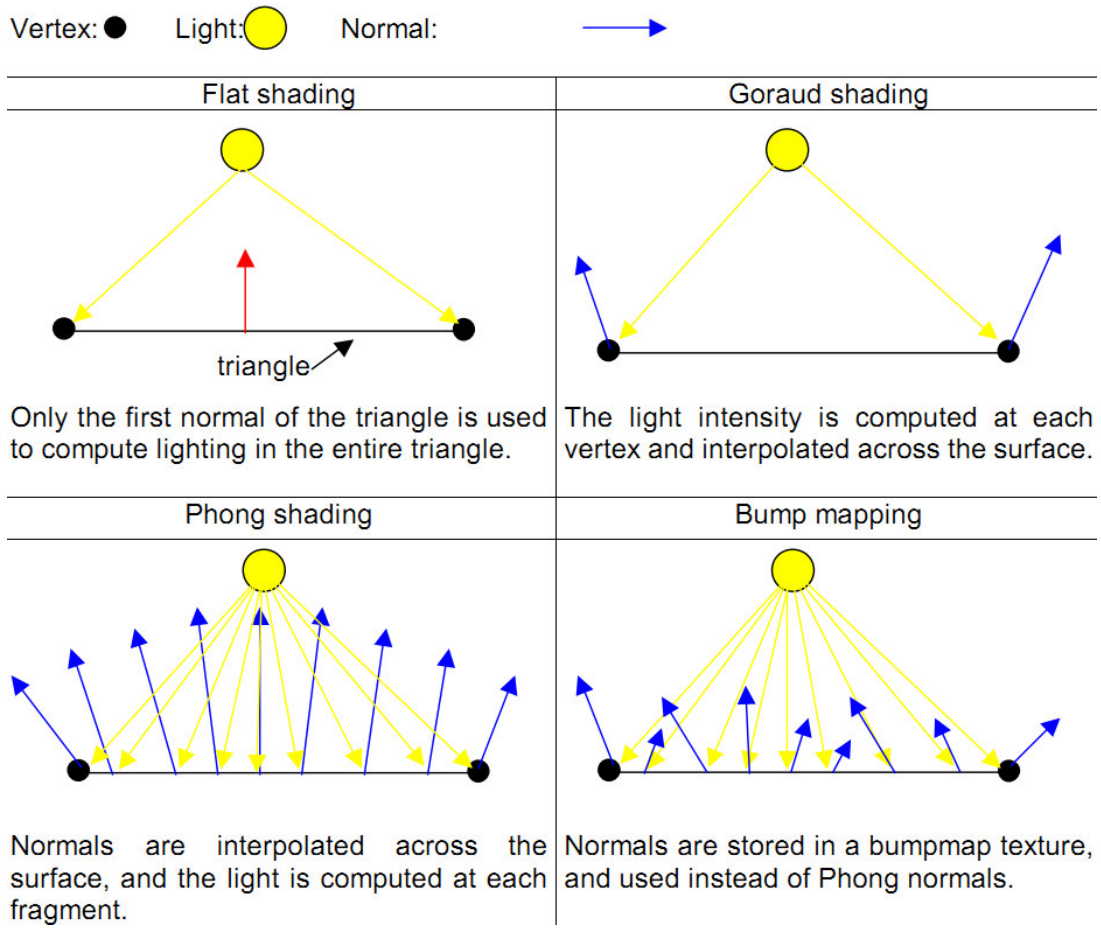


Figure 6.8: Comparison of shading models: Flat, Gouraud, Phong, and Bump Mapping.

6.7 Movie / M-DEC

The "Movie / M-DEC" function demonstrates the video playback capabilities of the PSX, using the .STR format to play a short commercial of about thirty seconds from the game *Spyro the*

Dragon.

The implementation of this function is based on the **STR player library by user Lameguy64**, which adapts the original example provided by Sony's PsyQ, developed by authors Yutaka, Suzu, Masa, and Ume.

The improved library includes the removal of legacy UTF-16 components, enhancements in code formatting and variable naming, and a simplification of memory allocation and buffering techniques.

The main playback routine, `PlayStr`, initializes the video stream with:

- Resolution configuration (`xres`, `yres`).
- Framebuffer positioning (`xpos`, `ypos`).
- STR file setup via `STRFILE`.

Key components of the playback process include:

- **Ring Buffer**: A cyclic memory buffer used for efficient video data management.
- **MDEC Initialization**: Resets the decoder and manages frame data streaming.
- **Video Synchronization**: Ensures frames are decoded and rendered sequentially, avoiding tearing or visual artifacts.

```
int PlayStr(int xres, int yres, int xpos, int ypos, STRFILE *str) {
    strNumFrames = str->NumFrames;
    strScreenWidth = xres;
    strScreenHeight = yres;
    strFrameX = xpos;
    strFrameY = ypos;
    strPlayDone = 0;
    strDoPlayback(str);
    return strPlayDone ? 1 : 0;
}
```

Codice 6.2: Simplified snippet of the playback logic.

Playback of .STR files uses the MDEC to decode and display video frames directly into the framebuffer, leveraging double buffering to achieve smooth transitions.

To convert a video file into .STR format, the following steps must be followed:

1. Convert the original video file to .AVI format using *VirtualDub* to adjust resolution, frame rate (15 FPS), and compression.
2. Convert the resulting .AVI file into the .STR format using Sony's *Movie Converter Tool*.

Thanks to these tools, developers were able to integrate FMV (Full Motion Video) sequences into their games, adding cinematic elements without the need for additional hardware. Sources: [138] [139] [140] [141].

Figure 6.9: The ”Movie / M-DEC” function running.

6.8 3D Animation

The *”3D Animation”* function showcases a simple 3D animation of a boxer throwing punches. The program allows adjusting the animation speed via the controller to offer direct control over the frame rate. The directional buttons *Left/Right* control the animation speed, while the *Cross* button resets the speed to its default value.

The boxer models, exported in `TMD` format and subsequently converted into binaries loaded from the CD-ROM, were sourced from the samples of the *Graphic CD Development Toolkit*, provided as part of the PSX development tools. This toolkit includes examples of 3D models and animations illustrating rendering on the PlayStation.

Additionally, a modified version of user Lameguy64’s software was used for loading and managing the `.TMD` files.

The function logic can be described as follows:

1. **Loading 3D models.**
    - The 3D models of the boxer are stored in files `tmd_00`, `tmd_01`, etc., and are loaded into memory at the start of the function.
2. **Animation management.**
    - The animation is structured as a loop that updates the animation frame. The current frame number and its duration are also displayed on screen.
3. **Frame rendering.**
    - Frames are drawn on the screen using PSX graphic primitives to represent the boxer's movements.

The purpose of this function is to demonstrate the PlayStation's ability to handle real-time 3D animations. Sources: [142] [143] [144].

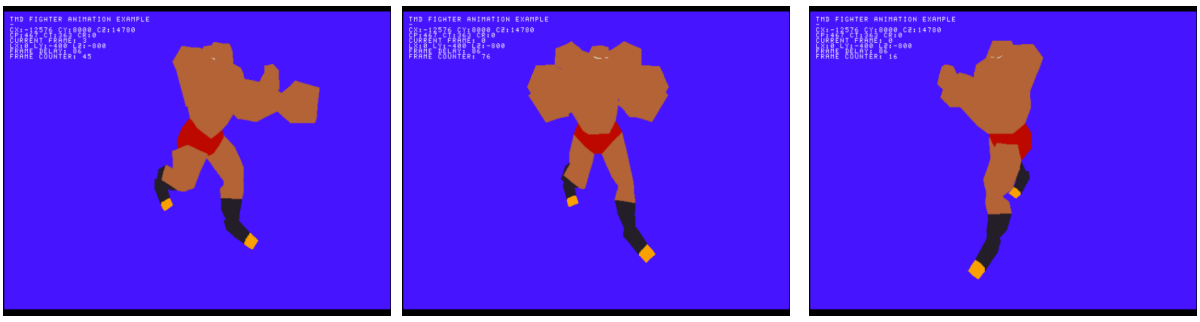


Figure 6.10: The "3D Animation" function running.

Chapter 7

Demo Disc Two (WipEout)

Figure 7.1: Game from the WipEout series in action, developed by Psygnosis. From the first installment in 1995 to the latest in 2021, the series saw the release of around 10 main titles.

7.1 Reverse Engineering WipEout

The game *WipEout* (1995) marked a turning point for the PlayStation era, standing out due to a combination of advanced graphics, an electronic soundtrack, and innovative gameplay. Thanks to the console's 3D capabilities, the title featured detailed environments and lighting effects that enhanced the sense of speed and immersion. These elements showcased the potential of real-time rendering on the PS1, establishing the game as a model for fully exploiting the available hardware. The soundtrack, curated by electronic music artists such as "The Chemical Brothers" and "Leftfield," perfectly integrated with the gameplay, setting a benchmark in the fusion between video game and musical culture.

The game quickly became both a commercial success and a strong technical showcase of the PSX hardware. For these reasons, the title was chosen as the basis for the development of the second Tech Demo in this analysis, as it allows exploration of the technologies used at the time and demonstrates their effectiveness in a modern context.

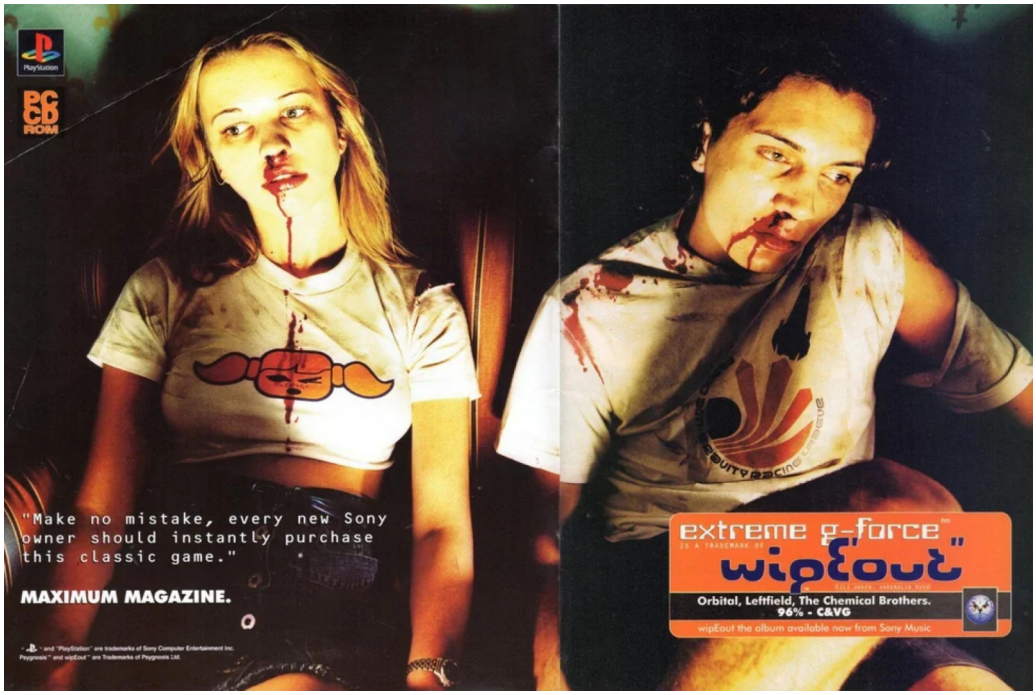


Figure 7.2: Controversial in-game advertising insert.

The demo aims to recreate a simplified version of the game *WipEout*, focusing on three essential aspects:

- **Loading the track and spaceship:** loading the 3D assets into memory and rendering them.
- **Basic movement logic:** implementation of basic physics for controlling the spaceship.
- **Soundtrack:** adding a soundtrack to enrich the audiovisual experience.

To facilitate development and ensure historical accuracy, original binary files (taken from an original disc version) will be used, recovered thanks to reverse engineering conducted by developer Dominic Szablewski. Szablewski documented the decoding and organization process of the game assets on his blog, providing details about their format and functions.

Another significant event for analyzing the game was a leak of the source code of a PC version of the game in 2022. Although the code was partially unreadable, it still allowed a better understanding of the overall game structure, thus easing the reverse engineering process.

The complete project is available on GitHub at the following address: [135].

Sources: [145] [146] [147].

7.2 Structure of a PRM File (Object)

The game developers used the `.PRM` format to save three-dimensional objects in a structured way. This type of binary file is designed to store information such as vertices, normals, primitives, and rendering-related attributes. Additionally, it allows fast loading and efficient access to data during runtime.

In the context of *WipEout*, data related to all the spaceships present were saved in the file `ALLSH.PRM` (All Ships). The file structure is illustrated as follows:

- **Name**: Name of the object (spaceship), represented as a string.
- **numverts**: Total number of vertices composing the model.
- **numnormals**: Number of normals associated with the vertices.
- **numprimitives**: Number of primitives used to draw the model.
- **flags**: Rendering-related information, such as transparency or special effects.
- **origin**: Position of the object in global space (`vx`, `vy`, `vz`).
- **vertices**: Array of vertices, each vertex described by coordinates (`vx`, `vy`, `vz`).
- **normals**: Array of normals, one for each vertex or primitive.
- **primitives**: List of primitives, each with additional data such as color, texture type, and UV coordinates for texture mapping.

The function `LoadObjectPRM` handles the loading of `.PRM` files. It uses a combination of helper functions such as `GetChar`, `GetShortBE`, and `GetLongBE` to decode the binary file data and populate the corresponding object data structures.

In detail, the loading process for a spaceship consists of the following steps:

1. **File opening**: The `.PRM` file is opened in binary read mode, verifying its existence and accessibility.
2. **Reading the Object Name**: The object name is read using the `GetChar` function, which reads one character at a time from the buffer and stores it.

```
for (i = 0; i < 32; i++) {
    object->name[i] = GetChar(bytes, &b);
    }
object->name[31] = '\0'; // Ensure null-termination
```

3. **Reading Basic Parameters**: Values `numverts`, `numnormals`, `numprimitives`, and `flags` are read. These values are stored as 32-bit integers and read using the `GetLongBE` function.

```
    object->numverts = GetLongBE(bytes, &b);
    object->numnormals = GetLongBE(bytes, &b);
    object->numprimitives = GetLongBE(bytes, &b);
    object->flags = GetLongBE(bytes, &b);
```

4. **Memory Allocation for Vertices and Normals**: Memory space is dynamically allocated to store the number of vertices and normals.

```
    object->vertices = (SVECTOR*) malloc(object->numverts *
        sizeof(SVECTOR));
    object->normals = (SVECTOR*) malloc(object->numnormals *
        sizeof(SVECTOR));
```

5. **Reading Vertex Coordinates**: Each vertex consists of three coordinates stored as 16-bit integers. These values are read with the `GetShortBE` function and stored in the `SVECTOR` structure.

```
    for (i = 0; i < object->numverts; i++) {
        object->vertices[i].vx = GetShortBE(bytes, &b);
        object->vertices[i].vy = GetShortBE(bytes, &b);
        object->vertices[i].vz = GetShortBE(bytes, &b);
    }
```

6. **Reading Primitives**: A primitive represents a face of an object and can be of different types (triangles or quadrilaterals). For each primitive, the type is identified and its specific data (vertex coordinates, color, texture, etc.) are read.

7. **Memory Management**: At the end of loading, the file is closed and the buffer is freed to avoid memory leaks.

The function sequence can thus be summarized as:

1. Sequential reading of data.
2. Dynamic memory allocation for object components.
3. Populating data structures with information extracted from the `.PRM` file.
4. Storing primitives and configuring rendering.

7.3 Ship Rendering Part 1 (Primitives and Wireframe)

The function `RenderObject` is used to render a ship on screen. The steps involved are as follows:

1. **Setting the *Transformation matrix***: Determine the position, orientation, and projection of objects in the three-dimensional space relative to the camera.
2. **Loop over primitives**: For each primitive, vertex coordinates are processed and clipping algorithms are applied.
3. **Geometry rendering**: Each primitive is drawn on the screen.

Example of rendering a `TypeF3` primitive:

```
1    POLY_F3* poly;
2    F3* prm = (F3*) object->primitives[i].primitive;
3    poly = (POLY_F3*) GetNextPrim();
4    gte_ldv0(&object->vertices[prm->coords[0]]);
5    gte_rtpt();
6    gte_stsxy3(&poly->x0, &poly->x1, &poly->x2);
7    SetPolyF3(poly);
8    addPrim(GetOTAt(GetCurrBuff(), otz), poly);
```

Codice 7.1: Example of rendering a TypeF3 primitive

The function utilizes the GTE to perform operations such as geometric transformations and rendering. Finally, the primitives are drawn on the screen using concepts like buffers and *Ordering Tables*.

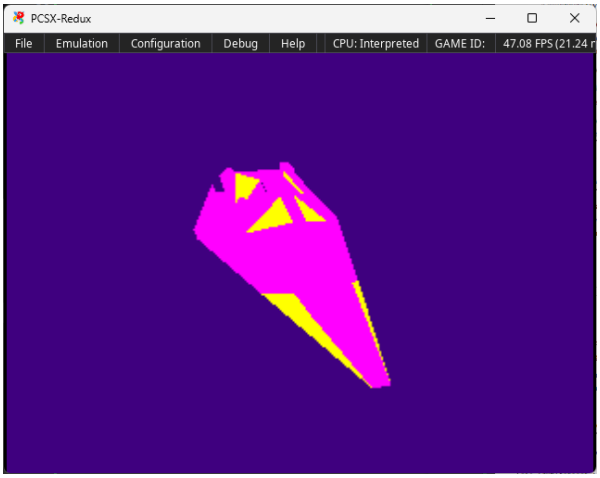


Figure 7.3: Rendered ship(Filled Polygons)

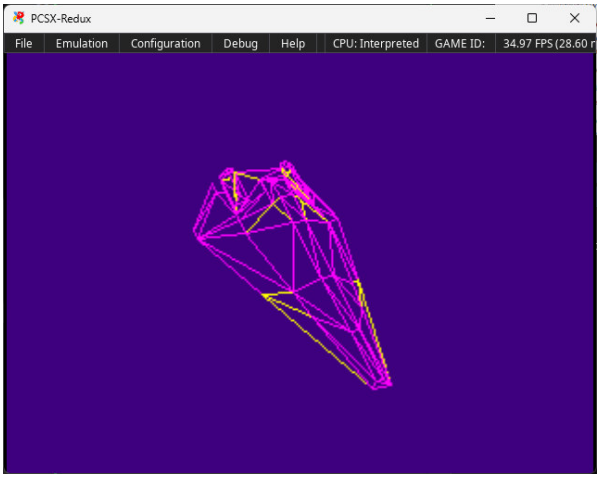


Figure 7.4: Rendered ship(Wireframe)

7.4 Structure of a CMP File (Texture)

Files with the extension .CMP (Compressed) were used by the WipEout developers to store the game's textures in a compressed format, optimizing the use of space on the CD-ROM. In this analysis, the ALLSH.CMP file contains a list of textures related to the ships in TIM format compressed using the LZSS (Lempel-Ziv-Storer-Szymanski) algorithm. The latter ensures effective lossless compression.

The .CMP file is structured as a sequence of long-type values, represented with Little-Endian ordering (unlike the PRM files, which use Big-Endian). The file is organized with the following values:

- **numtextures**: Total number of textures in the file.
- **timsize[0]...[n]**: Array containing the sizes in bytes of the decompressed TIM textures.
- **tim[0]...[n]**: Compressed data of each texture.

The main function for handling .CMP files is `LoadTextureCMP`. This function reads, decompresses, and loads the textures into VRAM following these steps:

1. **File reading**: The function `FileRead` reads the .CMP file and loads its contents into a memory buffer.
2. **Reading file structure**: The number of textures is read, and their offsets are calculated using the sizes in `timsize[]`.
3. **LZSS data decompression**: The function `ExpandLZSSData` decompresses the data.
4. **Loading into VRAM**: Each texture is loaded into VRAM via the function `UploadTextureToVRAM`.

7.5 Ship Rendering Part 2 (Textures)

The next step is the application of textures to the ships. The functions responsible for this operation are `UploadTextureToVRAM` and `RenderObjectWithTexture`. The first function handles:

1. Identifying the texture type.
2. Loading the CLUT (Color Lookup Table).
3. Loading the texture.

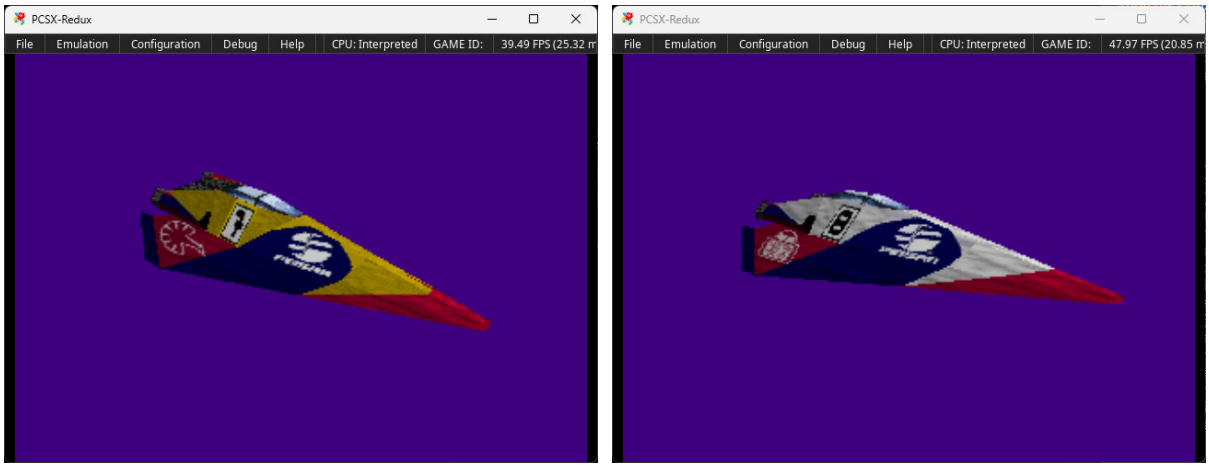


Figure 7.5: Rendering of two textured ships.

The second function is responsible for applying the texture during rendering:

```
void RenderObjectWithTexture(Object3D *object) {
    for (int i = 0; i < object->numprimitives; i++) {
        switch (object->primitives[i].type) {
            case PRIM_FT3: {
                POLY_FT3 *poly;
                FT3 *prm = (FT3*) object->primitives[i].primitive;
                // FT3: Triangle primitive with texture
                poly = (POLY_FT3*) GetNextPrim();
                // UV coordinates calculation: Load object vertices and
                   transform them
                gte_ldv0(&object->vertices[prm->coords[0]]);
                gte_rtpt();
                gte_stsxy3(&poly->x0, &poly->x1, &poly->x2);

                // Assign UV coordinates to the primitive
                setUVWH(poly,
                        prm->uv[0][0], prm->uv[0][1],
                        prm->uv[1][0], prm->uv[1][1],
```

```
                        prm->uv[2][0], prm->uv[2][1]);

                // Set TPage and CLUT
                setTPage(poly, object->tpage); // TPage: texture page in
                    VRAM
                setClut(poly, object->clut);  // CLUT: Color Lookup Table
                    for texture
                // Add the primitive to the Ordering Table
                SetPolyFT3(poly);
                addPrim(GetOTAt(GetCurrBuff(), otz), poly);
                break;
            }
            case PRIM_FT4: {
                POLY_FT4 *poly;
                FT4 *prm = (FT4*) object->primitives[i].primitive;
                // FT4: Quadrilateral primitive with texture
                poly = (POLY_FT4*) GetNextPrim();
                // UV coordinates calculation: Load object vertices and
                    transform them
                gte_ldv0(&object->vertices[prm->coords[0]]);
                gte_rtpt();
                gte_stsxy4(&poly->x0, &poly->x1, &poly->x2, &poly->x3);

                // Assign UV coordinates to the primitive
                setUVWH(poly,
                        prm->uv[0][0], prm->uv[0][1],
                        prm->uv[1][0], prm->uv[1][1],
                        prm->uv[2][0], prm->uv[2][1],
                        prm->uv[3][0], prm->uv[3][1]);

                // Set TPage and CLUT
                setTPage(poly, object->tpage); // TPage: texture page in
                    VRAM
                setClut(poly, object->clut);  // CLUT: Color Lookup Table
                    for texture
                // Add the primitive to the Ordering Table
                SetPolyFT4(poly);
                addPrim(GetOTAt(GetCurrBuff(), otz), poly);
                break;
            }
            default:
                break;
        }
    }
}
```

7.6 Structure of TRV, TRF, and TRS Files (Track Vertices, Faces, Sections)

The following two chapters analyze the process of loading and rendering the track within the software, following the methods used by the developers at the time. In the original game context, the track is stored on the CD-ROM through a series of files, each containing specific information:

- **Track.TRV**: defines the vertices that make up the track.
- **Track.TRF**: describes the faces of the track.
- **Track.TRS**: divides the track into sections.
- **Track.VEW**: specifies visibility lists for each section.

The track is organized following a well-defined hierarchy. It is composed of sections, each including faces, which in turn are defined by vertices. This structure is represented through a *linked-list* of sections that allows navigation both forwards and backwards. The first section is identified as the starting point of the track, while the last reconnects to the first, forming a loop.

The faces constituting the track are represented by primitives, specifically quadrilaterals. Additionally, each section has a `VECTOR` type variable called `center`. This defines the center of the sector, a crucial piece of information used to determine the sector's distance from the camera and evaluate whether the section should be rendered or not.

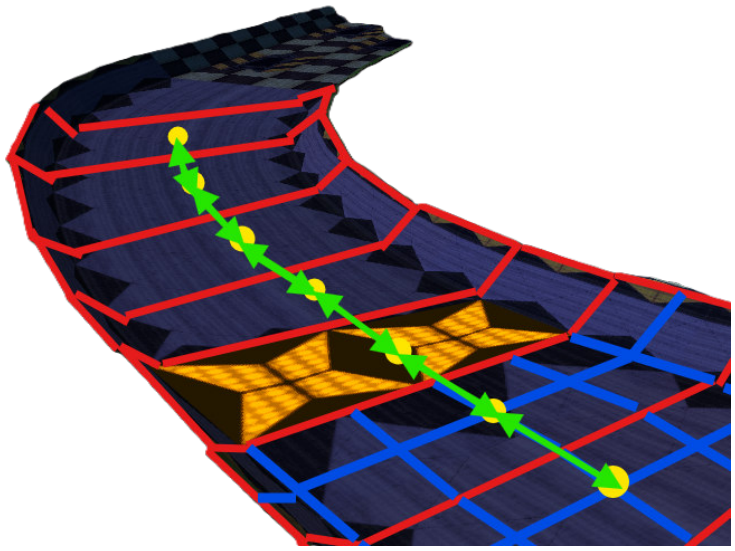


Figure 7.6: Example of track structure: Sections are shown in red, faces in blue, and the section center in yellow. The green bidirectional arrows represent the linked-list.

This structure is managed in the demo software through three main data structures (`struct`): `Face`, `Section`, and `Track`.

```c
typedef struct Face {
  short indices[4];  // Indices of the four vertices of the face.
  char flags;        // Flags describing the properties of the face.
  SVECTOR normal;    // Normal to the face for lighting calculation.
  CVECTOR color;     // Color of the face.
  char texture;      // ID of the texture associated with this face.
  short clut;        // Color Look-Up Table (CLUT).
  short tpage;       // Indicates the texture page in VRAM.
  short u0, v0;      // UV coordinates of the first vertex of the texture.
  short u1, v1;      // UV coordinates of the second vertex of the texture.
  short u2, v2;      // UV coordinates of the third vertex of the texture.
  short u3, v3;      // UV coordinates of the fourth vertex of the texture.
} Face;
```

```c
typedef struct Section {
  short id;          // Unique identifier of the section.
  short flags;       // Flags representing properties of the section.

  struct Section *prev; // Pointer to the previous section.
  struct Section *next; // Pointer to the next section.

  VECTOR center;     // Geometric center of the section.
  SVECTOR normal;    // Normal of the section, for calculating
     orientations or visual effects.

  short numfaces;    // Number of faces present in the section.
  short facestart;   // Index of the first face in the section.
} Section;
```

```c
typedef struct Track {
  long numVertices;   // Total number of vertices in the track.
  VECTOR *vertices;   // Pointer to the array of vertices.

  long numFaces;      // Total number of faces in the track.
  Face *faces;        // Pointer to the array of faces.

  long numSections;   // Total number of sections in the track.
  Section *sections;  // Pointer to the array of sections, organized as a
     circular linked list.
} Track;
```

Codice 7.2: Track-related structs used in the Tech Demo.

The `Track.VEW` file, although not included in the current analysis to simplify track implementation, provides a visibility mapping of each section relative to others. It is organized into five

views:

- **North view**: includes sections visible in front of the ship.
- **South view**: includes sections visible behind the ship.
- **East view**: includes sections visible to the right of the ship.
- **West view**: includes sections visible to the left of the ship.
- **Global view**: includes all sections visible around the ship (used for modes such as replay or demo).

7.7 Track Rendering

The track rendering is managed through a pipeline that includes loading the track data, handling its textures, and rendering the faces/sections of the track.

The first step consists of loading the geometric data of the track from the corresponding files seen in the previous chapter, starting from the vertices contained in the TRV file. These are read from the file and stored in an array within the `Track` struct.

```
void LoadTrackVertices(Track *track, char *filename) {
  u_long length, b = 0;
  u_char *bytes = (u_char*) FileRead(filename, &length);
  if (bytes == NULL) {
    printf("Error reading %s from the CD.\n", filename);
    return;
  }

  track->numVertices = length / BYTES_PER_VERTEX; // Total number of
     vertices in the track.
  track->vertices = (VECTOR*) malloc(track->numVertices * sizeof(VECTOR));
     // Allocate memory for the vertices.

  for (u_long i = 0; i < track->numVertices; i++) {
    track->vertices[i].vx = GetLongBE(bytes, &b); // X coordinate of the
       vertex.
    track->vertices[i].vy = GetLongBE(bytes, &b); // Y coordinate of the
       vertex.
    track->vertices[i].vz = GetLongBE(bytes, &b); // Z coordinate of the
       vertex.
  }

  free(bytes); // Free the temporary buffer.
}
```

Codice 7.3: Function `LoadTrackVertices`

Next, the faces are defined, which will compose the surface of the track. These are loaded from the TRF file.

```
void LoadTrackFaces(Track *track, char *filename, u_short texturestart) {
  u_long length, b = 0;
  u_char *bytes = (u_char*) FileRead(filename, &length);

  if (bytes == NULL) {
    printf("Error reading %s from the CD.\n", filename);
    return;
  }

  track->numfaces = length / BYTES_PER_FACE;
  track->faces = (Face*) malloc(track->numfaces * sizeof(Face));

  for (u_long i = 0; i < track->numfaces; i++) {
    Face *face = &track->faces[i];
    face->indices[0] = GetShortBE(bytes, &b);
    face->indices[1] = GetShortBE(bytes, &b);
    face->indices[2] = GetShortBE(bytes, &b);
    face->indices[3] = GetShortBE(bytes, &b);

    face->texture += texturestart;
  }

  free(bytes);
}
```

Codice 7.4: Function `LoadTrackFaces`

The track rendering is then divided into sections. Thanks to this approach, it is possible to optimize the rendering process by displaying only the sections visible to the camera.

```
void RenderTrackSection(Track *track, Section *section, Camera *camera,
    u_short numsubdivs) {
  MATRIX worldmat, viewmat;
  SVECTOR v0, v1, v2, v3;
  for (int i = 0; i < section->numfaces; i++) {
    Face *face = track->faces + section->facestart + i;
    v0.vx = track->vertices[face->indices[0]].vx - camera->position.vx;
    v1.vx = track->vertices[face->indices[1]].vx - camera->position.vx;
    RenderQuadRecursive(face, &v0, &v1, ...);
  }
```

```
}
```

Codice 7.5: Function RenderTrackSection

The function RenderTrackAhead is responsible for selecting only the sections visible around the ship. In this way, only the necessary sections are rendered.

```
void RenderTrackAhead(Track *track, Section *startsection, Camera *camera)
    {
  Section *currsection = startsection;

  for (u_short i = 0; i < 20; i++) {
    RenderTrackSection(track, currsection, camera, i < 6 ? 1 : 2);
    currsection = currsection->next;
  }
}
```

Codice 7.6: Function RenderTrackAhead

Each face is assigned its texture. The function LoadTextureCMP reads the texture data in CMP format and decompresses it before loading it into VRAM, as explained in chapter 7.5.

```
void LoadTextureCMP(char *filenamecmp, char *filenamettf) {
    u_long b, length;
    u_char *bytes;
    static void *timsbaseaddr;
    static long timoffsets[400];

    bytes = (u_char*) FileRead(filenamecmp, &length);

    if (bytes == NULL) {
        printf("Error reading %s from the CD.\n", filenamecmp);
        return;
    }
    b = 0;
    u_short numtextures = GetLongLE(bytes, &b);
    // Decompressione dei dati
    timsbaseaddr = malloc(totaltimsize);
    ExpandLZSSData(&bytes[b], timsbaseaddr);
    free(bytes);

    // Caricamento delle texture nella VRAM
    for (u_short i = 0; i < numtextures; i++) {
        Texture *texture = UploadTextureToVRAM(timoffsets[i]);
        if (texture != NULL) {
            texturestore[texturecount++] = texture;
        }
```

```
    }
}
```

Codice 7.7: Function `LoadTextureCMP`

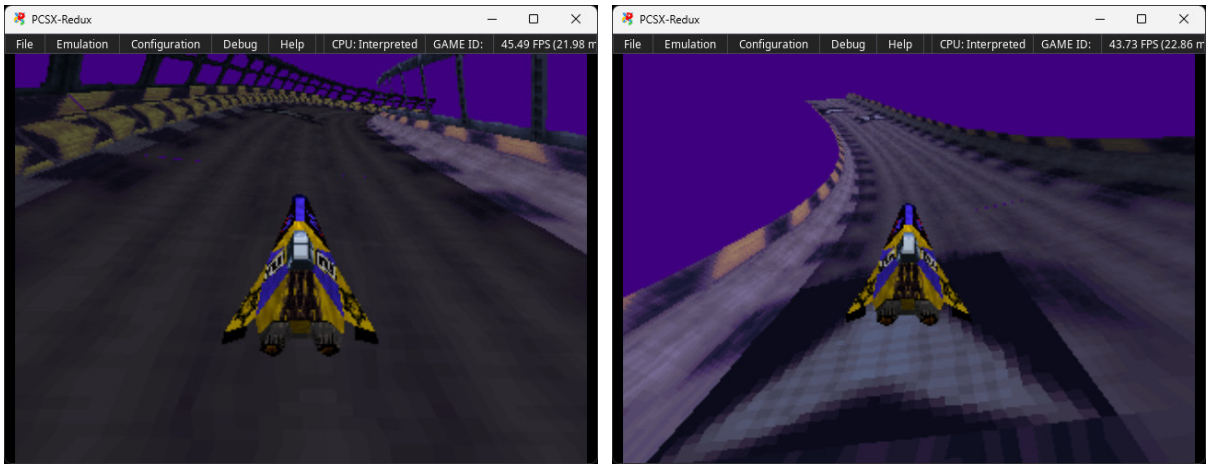


Figure 7.7: Track rendering.

7.8 Implementation of Physics, Gameplay, and Sound Effects

This chapter analyzes the implementation of the gameplay loop and how physics is applied in the game. In the original WipEout, the ships do not touch the track; rather, they hover above it. To replicate this effect, it was necessary to manage player movement, interaction with the track, and forces acting on the ship.

First, the function `ShipInit` configures the ship by positioning it at the starting point and associating it with the nearest section of the track.

```
void ShipInit(Ship *ship, Track *track, VECTOR *startPos) {
    Section *current;
    VECTOR delta;
    long distMagnitude, minDist;

    // Initialize the ship's position and dynamic parameters
    ship->object->position = *startPos;
    ship->vel = ship->acc = ship->thrust = ship->drag = (VECTOR) {0, 0, 0};
    ship->yaw = ship->pitch = ship->roll = 0;
    ship->thrustMax = 15000;
    [...]

    // Find the closest section of the track
    current = track->sections;
    minDist = 99999999;
    do {
```

```
        // Calculate the distance and update the closest section
        [...]
        current = current->next;
    } while (current != track->sections);
}
```

Codice 7.8: Function ShipInit

Next, the function ShipUpdate calculates the new position of the ship considering various forces such as gravity, friction, and thrust.

```
void ShipUpdate(Ship *ship) {
    VECTOR force = {0, 0, 0};
    VECTOR noseVelocity, baseToShip;
    long dotProduct, height;

    // Calculate the ship's orientation (right, up, forward) based on yaw,
       pitch, and roll
    [...]

    // Calculate the thrust force and forward-projected velocity
    [...]

    // Determine the height between the ship and the track
    baseToShip.vx = ship->object->position.vx -
       ship->section->baseVertex.vx;
    baseToShip.vy = ship->object->position.vy -
       ship->section->baseVertex.vy;
    baseToShip.vz = ship->object->position.vz -
       ship->section->baseVertex.vz;

    dotProduct = (ship->section->normal.vx * baseToShip.vx +
                  ship->section->normal.vy * baseToShip.vy +
                  ship->section->normal.vz * baseToShip.vz) >> 12;

    height = max(dotProduct, 50);

    // Calculate the total force (attraction, repulsion, thrust) and
       update acceleration, velocity, and position
    [...]

    // Update angular rotation and realign the ship
    [...]

    // Update nearest track section and orientation matrix
    UpdateShipNearestSection(ship);
    [...]
```

```
}
```

Codice 7.9: Function ShipUpdate

The ship updates the nearest section of the track through the function UpdateShipNearestSection.

```
void UpdateShipNearestSection(Ship *ship) {
    Section *current = ship->section;
    VECTOR delta;
    long distMagnitude, minDist = 99999999;

    // Search for the nearest section among the 4 adjacent ones
    for (u_short i = 0; i < 4; i++) {
        // Calculate distance from the current section
        delta.vx = current->center.vx - ship->object->position.vx;
        delta.vy = current->center.vy - ship->object->position.vy;
        delta.vz = current->center.vz - ship->object->position.vz;
        distMagnitude = SquareRoot12(delta.vx * delta.vx + delta.vy *
            delta.vy + delta.vz * delta.vz);

        // Update the nearest section
        if (distMagnitude < minDist) {
            minDist = distMagnitude;
            ship->section = current;
        }

        current = current->next; // Next section
    }
}
```

Codice 7.10: Function UpdateShipNearestSection

The player interacts with the ship via the joypad, controlling its behavior by pressing buttons.

```
void ShipUpdate(Ship *ship) {
    VECTOR force, noseVelocity, baseToShip;
    long dotProduct, height;
    short sinX, cosX, sinY, cosY, sinZ, cosZ;

    // Calculate sine and cosine for yaw, pitch, and roll [...]

    // Calculate the right, up, and forward vectors based on the ship's
        orientation [...]

    // Calculate the thrust force based on the forward direction [...]

    // Calculate velocity and forward-projected velocity [...]
```

```
13
14      // Determine the height between the ship and the track [...]
15
16      // Calculate the total force (attraction, repulsion, and thrust) [...]
17
18      // Update the ship's acceleration, velocity, and position [...]
19
20      // Update rotation (yaw, pitch, and roll) based on angular velocity
            [...]
21
22      // Update the nearest section and the orientation matrix [...]
23  }
```

Codice 7.11: Function `ShipUpdate`

Sound effects, such as the initial countdown and background music, are managed by the SPU. The function `PlayAudioTrack` starts a specific track, while `AudioPlay` handles the triggering of sound effects.

```
1  void AudioPlay(int voiceChannel) {
2      // Start audio playback for the specified channel
3      SpuSetKey(SpuOn, voiceChannel);
4  }
```

Codice 7.12: Function `AudioPlay`

```
1  void PlayAudioTrack(u_short trackNumber) {
2    u_int i;
3    u_char param[4];
4    u_char result[8];
5
6    // Transfer can be done via I/O or DMA
7    SpuSetTransferMode(SpuTransByDMA);
8
9    // Get CD TOC (Table of Contents) [...]
10
11   // Prevent out-of-bounds position [...]
12
13   // Set CD parameters: Report mode ON, CD-DA ON (see LibOvr.pdf, page 188)
14   param[0] = CdlModeRept | CdlModeDA;
15
16   // Set mode
17   CdControlB(CdlSetmode, param, 0);
18
19   // Wait for three VSyncs
20   VSync(3);
21
```

```
22    // Play the track from the TOC array
23    CdControlB(CdlPlay, (u_char*) &loc[trackNumber], 0);
24  }
```

Codice 7.13: Function `PlayAudioTrack`

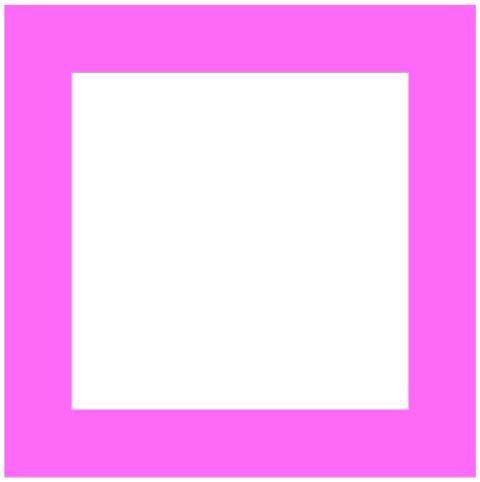
Finally, the camera follows the ship, dynamically adapting to its position and direction. To achieve this, the function `LookAt` is used, which calculates the camera transformation matrix based on position, direction, and the ”UP” vector.

```
1  void LookAt(Camera *camera, VECTOR *eye, VECTOR *target, VECTOR *up) {
2      VECTOR xRight, yUp, zForward, pos, temp;
3      VECTOR x, y, z;
4
5      // Calculate the zForward vector (direction from camera to target) and
           normalize it
6      zForward.vx = target->vx - eye->vx;
7      zForward.vy = target->vy - eye->vy;
8      zForward.vz = target->vz - eye->vz;
9      VectorNormal(&zForward, &z);
10
11     // Calculate the xRight vector (cross product between zForward and up)
           and normalize it
12     VectorCross(&z, up, &xRight);
13     VectorNormal(&xRight, &x);
14
15     // Calculate the yUp vector (cross product between zForward and
           xRight) and normalize it
16     VectorCross(&z, &x, &yUp);
17     VectorNormal(&yUp, &y);
18
19     // Assign vectors x, y, z to the lookat matrix [...]
20
21     // Invert the camera position (eye) to obtain the translation
22     pos.vx = -eye->vx;
23     pos.vy = -eye->vy;
24     pos.vz = -eye->vz;
25
26     // Save the rotation part in the rotation matrix rotmat [...]
27
28     // Combine rotation and translation in the lookat matrix
29     ApplyMatrixLV(&camera->lookat, &pos, &temp);
30     TransMatrix(&camera->lookat, &temp);
31 }
```

Codice 7.14: Function `LookAt`

Part IV

Resources and Final Considerations



Chapter 8

Conclusions

Software development for the PlayStation remains, despite its thirty-year history, a subject of great interest both for fans of Sony's console and for retrocomputing enthusiasts. Thanks to numerous online communities and dedicated forums, it is possible to access original documentation, tools, and hobbyist software for developing applications for the original hardware. The work carried out in this thesis focused not only on retracing the techniques and technologies of the era but also on addressing the challenges encountered by programmers through the use of original tools or their equivalents. This approach made it possible to highlight both the limitations of the console and the innovations introduced by the PlayStation to the videogame industry.

The experimental chapters, centered on the development of the two tech demos, allowed a practical evaluation of the console's capabilities, confronting the memory, computation, and hardware architecture constraints of the PSX. In particular, their development enabled the exploration of advances such as 3D rendering, advanced audio and video handling, and the use of the CD-ROM.

Ultimately, this work aims to act as a bridge to the past, emphasizing and valuing both the weaknesses and the strengths of a console that defined a generation.

Glossario

ABI *Application Binary Interface*
ADPCM *Adaptive Differential Pulse Code Modulation*
API *Application Programming Interface*
AV *Audio/Video*
BSS *Block Started by Symbol*
CISC *Complex Instruction Set Computer*
CLUT *Color Lookup Table*
CPI *Cycles Per Instruction*
CPU *Central Processing Unit*
DAC *Digital-to-Analog Converter*
DA *Digital Audio*
DCT *Discrete Cosine Transform*
DC *Direct Current*
DMA *Direct Memory Access*
DRAM *Dynamic Random Access Memory*
DSP *Digital Signal Processing*
DTL-H *Development Tool Hardware*
E3 *Electronic Entertainment Expo*
EDO *Extended Data Out*
FIFO *First In, First Out*
FMV *Full Motion Video*
FPU *Floating Point Unit*
GNU *GNU's Not Unix*
GPU *Graphics Processing Unit*
GTE *Geometry Transformation Engine*
I/O *Input/Output*
ISA *Industry Standard Architecture*
JPEG *Joint Photographic Experts Group*
LIFO *Last In, First Out*
LSB *Least Significant Byte*
MDEC *Motion Decoder / Macroblock Decoder*
MMU *Memory Management Unit*
MIPS *Microprocessor Without Interlocked Pipeline Stages*
MSB *Most Significant Byte*
MPEG *Moving Picture Experts Group*
NES *Nintendo Entertainment System*
NTSC *National Television System Committee*
OT *Ordering Table*
PAL *Phase Alternating Line*
PCM *Pulse Code Modulation*
PDA *Personal Digital Assistant*
RAM *Random Access Memory*
RGB *Red, Green, Blue*
RISC *Reduced Instruction Set Computer*
RF *Radio Frequency*
RFU *Radio Frequency Unit*
SCEA *Sony Computer Entertainment of America*
SCEE *Sony Computer Entertainment of Europe*
SCEI *Sony Computer Entertainment of Japan*
SCPH *Sony Computer PlayStation Hardware*
SDK *Software Development Kit*
SDevTC *Sony Developer Toolchain*
SNES *Super Nintendo Entertainment System*

SoC	*System On a Chip*	**TEXEL**	*Texture Element*
SPU	*Sound Processing Unit*	**TOC**	*Table of Contents*
STP	*Special Transparency Processing*	**ULP**	*Unit of Least Precision*
SRAM	*Static Random-Access Memory*	**VBlank**	*Vertical Blanking*
T-PAGE	*Texture Page*	**VRAM**	*Video Random Access Memory*
TCB	*Thread Control Blocks*	**XA**	*Extended Architecture*

Istruzioni MIPS menzionate

LI	*Load Immediate*	**ADDU**	*Add Unsigned*
LA	*Load Address*	**ADD**	*Addition*
LUI	*Load Unsigned Immediate*	**ADDI**	*Add Immediate*
LW	*Load Word*	**ADDIU**	*Add Immediate Unsigned*
LH	*Load Half*	**SUB**	*Subtract*
LB	*Load Byte*	**SUBU**	*Subtract Unsigned*
LHU	*Load Half Unsigned*	**MULT**	*Multiply*
LBU	*Load Byte Unsigned*	**MULTU**	*Multiply Unsigned*
SW	*Store Word*	**DIV**	*Divide*
SH	*Store Half*	**DIVU**	*Divide Unsigned*
SB	*Store Byte*	**MFHI**	*Move From HI*
J	*Jump*	**MFLO**	*Move From LO*
JAL	*Jump and Link*	**AND**	*Logical And*
JR	*Jump Register*	**ANDI**	*Logical And Immediate*
JALR	*Jump and Link Register*	**OR**	*Logical Or*
BEQ	*Branch if Equal*	**ORI**	*Logical Or Immediate*
BNE	*Branch if Not Equal*	**XOR**	*Exclusive Or*
BLEZ	*Branch if Less Than or Equal to Zero*	**XORI**	*Exclusive Or Immediate*
BGEZ	*Branch if Greater Than or Equal to Zero*	**SLL**	*Shift Left Logical*
		SRL	*Shift Right Logical*
BLTZ	*Branch if Less Than Zero*	**SRA**	*Shift Right Arithmetical*
BGTZ	*Branch if Greater Than Zero*	**SLLV**	*Shift Left Logical Variable*
BLT	*Branch if Less Than*	**SRLV**	*Shift Right Logical Variable*
BLE	*Branch if Less Than or Equal*	**SRAV**	*Shift Right Arithmetic Variable*
NOP	*No Operation*		

Visual Documentation

[1] *Edge (magazine)*. Page Version ID: 1256577091. Nov. 10, 2024. URL: https://en.wikipedia.org/w/index.php?title=Edge_(magazine)&oldid=1256577091 (visited on 11/17/2024).

[2] *30 anni di PlayStation: 'Nemmeno Sony credeva nel successo di PS1', rivela Kutaragi*. Everyeye Videogiochi. Section: Videogiochi. Sept. 29, 2024. URL: https://www.everyeye.it/notizie/30-anni-playstation-nemmeno-sony-credeva-successo-ps1-rivela-kutaragi-745445.html (visited on 11/13/2024).

[3] *CD-ROM - Wikipedia*. URL: https://en.wikipedia.org/wiki/CD-ROM (visited on 11/13/2024).

[4] *Sony Interactive Entertainment*. Page Version ID: 1257115515. Nov. 13, 2024. URL: https://en.wikipedia.org/w/index.php?title=Sony_Interactive_Entertainment&oldid=1257115515 (visited on 11/13/2024).

[5] *Psygnosis*. Page Version ID: 1255624711. Nov. 5, 2024. URL: https://en.wikipedia.org/w/index.php?title=Psygnosis&oldid=1255624711 (visited on 11/13/2024).

[6] *Destruction Derby Images - LaunchBox Games Database*. URL: https://gamesdb.launchbox-app.com/games/images/10354-destruction-derby (visited on 11/18/2024).

[7] *Wipeout Images - LaunchBox Games Database*. URL: https://gamesdb.launchbox-app.com/games/images/11090-wipeout (visited on 11/18/2024).

[8] *Novastorm Images - LaunchBox Games Database*. URL: https://gamesdb.launchbox-app.com/games/images/31091-novastorm (visited on 11/18/2024).

[9] *Spyro the Dragon Images - LaunchBox Games Database*. URL: https://gamesdb.launchbox-app.com/games/images/4730-spyro-the-dragon (visited on 11/18/2024).

[10] *Crash Bandicoot: Warped Images - LaunchBox Games Database*. URL: https://gamesdb.launchbox-app.com/games/images/493-crash-bandicoot-warped (visited on 11/18/2024).

[11] *Silent Hill Images - LaunchBox Games Database*. URL: https://gamesdb.launchbox-app.com/games/images/984-silent-hill (visited on 11/18/2024).

[12] *Metal Gear Solid Images - LaunchBox Games Database*. URL: https://gamesdb.launchbox-app.com/games/images/187-metal-gear-solid (visited on 11/18/2024).

[13] *Final Fantasy VII Images - LaunchBox Games Database*. URL: https://gamesdb.launchbox-app.com/games/images/503-final-fantasy-vii (visited on 11/18/2024).

[14] *Tekken 3 Images - LaunchBox Games Database*. URL: https://gamesdb.launchbox-app.com/games/images/2414-tekken-3 (visited on 11/18/2024).

[15] Bruarn. *Models of PlayStation*. May 28, 2022. URL: https://commons.wikimedia.org/w/index.php?curid=118426033 (visited on 11/14/2024).

[16] *Image of PlayStation SCPH-1000 Box*. URL: https://i.ebayimg.com/images/g/InIAAOSwINZml-lr/s-l1600.webp (visited on 11/14/2024).

[17] *Vano disco del modello SCPH-1000*. URL: https://imgur.com/8tIZvXN (visited on 11/14/2024).

[18] *Sony Playstation SCPH-100x GPU Specs*. TechPowerUp. Nov. 14, 2024. URL: https://www.techpowerup.com/gpu-specs/sony-playstation-scph-100x-gpu.b8176 (visited on 11/14/2024).

[19] *PlayStation Serie SCPH-1000 — PlayStation Generation*. URL: https://www.playstationgeneration.it/2011/04/playstation-serie-scph-1000.html (visited on 11/13/2024).

[20] *Fighting Box*. URL: https://cdn.suruga-ya.com/database/pics_webp/game/140010075.jpg.webp (visited on 11/14/2024).

[21] *Sony Playstation SCPH-100x GPU Specs*. TechPowerUp. Nov. 14, 2024. URL: https://www.techpowerup.com/gpu-specs/sony-playstation-scph-100x-gpu.b8176 (visited on 11/14/2024).

[22] *SONY Playstation SCPH-5500*. URL: https://bangladesh.desertcart.com/products/14914474-sony-playstation-1-complete-system-console-ps-1-psx (visited on 11/14/2024).

[23] Obsolete Sony [@ObsoleteSony]. *Originally, the PlayStation was meant to support Video CD, but this feature was limited to the SCPH-5903 model available only in Hong Kong and Taiwan. Sony released this console due to the popularity of VCD in these regions, but it didn't sell well.* Twitter Post. Aug. 21, 2024. URL: https://x.com/ObsoleteSony/status/1826212657965289826 (visited on 11/14/2024).

[24] *Sony Playstation SCPH-700x GPU Specs*. TechPowerUp. Nov. 14, 2024. URL: https://www.techpowerup.com/gpu-specs/sony-playstation-scph-700x-gpu.b8180 (visited on 11/14/2024).

[25] Keep Fronting [keepfronting]. *Ad for the PS1 Dual Analogue Controller*. Tumblr Post. Tumblr. URL: https://keepfronting.tumblr.com/post/189547023443/vgjunk-ad-for-the-ps1-dual-analogue-controller (visited on 11/14/2024).

[26] *PlayStation 10 Million Model - Midnight Blue (SCPH-7000W) — PlayStation Generation*. URL: https://www.playstationgeneration.it/2010/08/playstation-10-million-model.html (visited on 11/14/2024).

[27] *Sony Playstation SCPH-900x GPU Specs*. TechPowerUp. Nov. 14, 2024. URL: https://www.techpowerup.com/gpu-specs/sony-playstation-scph-900x-gpu.b8182 (visited on 11/14/2024).

[28] *PSone (SCPH-100) — PlayStation Generation*. URL: https://www.playstationgeneration.it/2010/08/psone.html (visited on 11/14/2024).

[29] *Debugging Station - DTL-H1202*. Instagram Post. Jan. 27, 2024. URL: https://www.instagram.com/p/C2l2hYrxFpK/ (visited on 11/14/2024).

[30] iCrewPlay Marco Consiglio. *PlayStation Net Yaroze: la console fatta per creare i giochi*. Section: Notizie in Vetrina. July 31, 2022. URL: https://www.icrewplay.com/videoludica-la-playstation-net-yaroze/ (visited on 11/14/2024).

[31] *Sony PlayStation Multitap — SCPH-1070 — Sony PS1 — Gumtree Australia*. URL: https://www.gumtree.com.au/s-ad/mitcham/console-accessories/sony-playstation-multitap-scph-1070-sony-ps1/1310483197 (visited on 11/15/2024).

[32] *Curioso y Misterioso: PS1 Multitap*. Video in Spanish. URL: https://www.youtube.com/watch?v=C1z4r3BG1Ew (visited on 11/15/2024).

[33] Rodrigo Copetti. *PlayStation Architecture - A Practical Analysis*. 2019. URL: https://www.copetti.org/writings/consoles/playstation/.

[34] *Endianness*. Page Version ID: 1233149731. July 7, 2024. URL: https://en.wikipedia.org/w/index.php?title=Endianness&oldid=1233149731 (visited on 11/16/2024).

[35] *PlayStation*. Page Version ID: 141378433. Oct. 2, 2024. URL: https://it.wikipedia.org/w/index.php?title=PlayStation&oldid=141378433 (visited on 11/17/2024).

[36] *How to use Signed and Unsigned in VHDL*. URL: https://creativo.blog.ir/1399/01/08/vhdl (visited on 11/20/2024).

[37] *RGB Color Images*. ResearchGate. URL: https://www.researchgate.net/figure/RGB-Color-Images-The-Figure8-shows-how-to-find-the-RGB-value-decomposition-of-a-32-bit_fig6_325568402 (visited on 11/20/2024).

[38] *Single-bit Left/Right Rotating, Logical Shift, and Arithmetic Shift Operations on 8-bit Data*. ResearchGate. URL: https://www.researchgate.net/figure/Single-

bit-left-right-rotating-logical-shift-and-arithmetic-shift-operations-on-8-bit_fig6_282306909 (visited on 11/20/2024).

[39] Mux. *Delay Slots*. low tech. July 12, 2011. URL: https://sigalrm.blogspot.com/2011/07/delay-slots.html (visited on 11/21/2024).

[40] *Atomicity, Interlaced Mode, Buffered Ordering Tables - PlayStation Development Network*. URL: https://www.psxdev.net/forum/viewtopic.php?t=561 (visited on 11/22/2024).

[41] *FF7/Kernel/Memory Management - Final Fantasy Inside*. URL: https://wiki.ffrtt.ru/index.php/FF7/Kernel/Memory_management (visited on 11/22/2024).

[42] *PSX/TIM format - Final Fantasy Inside*. URL: https://wiki.ffrtt.ru/index.php/PSX/TIM_format#Image_data (visited on 11/22/2024).

[43] *DIY Calculator :: The Origin of the Computer Console/Display/Screen/Monitor*. URL: https://www.clivemaxfield.com/diycalculator/popup-h-console.shtml (visited on 11/22/2024).

[44] *How PlayStation Graphics & Visual Artefacts Work*. URL: https://pikuma.com/blog/how-to-make-ps1-graphics (visited on 11/23/2024).

[45] *CS307: Introduction to Computer Graphics*. URL: https://cs.wellesley.edu/~cs307/lectures/Transparency-S22.html (visited on 12/19/2024).

[46] Ícaro L. L. da Cunha and Luiz M. G. Gonçalves. *An Adaptive and Hybrid Approach to Revisiting the Visibility Pipeline*. Publisher: Centro Latinoamericano de Estudios en Informática. Apr. 2016. URL: http://www.scielo.edu.uy/scielo.php?script=sci_abstract&pid=S0717-50002016000100008&lng=es&nrm=iso&tlng=en (visited on 12/20/2024).

[47] Hendrik Lensch. *Computer Graphics - Camera Transformations , pagina 19*. URL: https://resources.mpi-inf.mpg.de/departments/d4/teaching/ws200708/cg/slides/CG15-Camera.pdf (visited on 12/12/2024).

[48] *Line Clipping — Set 1 (Cohen–Sutherland Algorithm)*. GeeksforGeeks. Section: Algorithms. URL: https://www.geeksforgeeks.org/line-clipping-set-1-cohen-sutherland-algorithm/ (visited on 12/21/2024).

[49] *Liang-Barsky Line Clipping*. URL: https://www.cs.helsinki.fi/group/goa/viewing/leikkaus/intro.html (visited on 12/21/2024).

[50] Fabio Biscaro. *Numeri binari in notifica IEEE 754 a 32 bit*. Dec. 26, 2012. URL: https://www.youtube.com/watch?app=desktop&v=N6U_xUdUWC8&t=230s (visited on 12/21/2024).

[51] *Adding Fixed point arithmetic to your design - theDataBus.in*. July 23, 2020. URL: https://thedatabus.in/fixed-point (visited on 12/21/2024).

[52] *FixedPoint - vanhunteradams*. URL: https://vanhunteradams.com/FixedPoint/FixedPoint.html (visited on 12/21/2024).

[53] *SECURITY OFFENSE AND DEFENSE STRATEGIES: VIDEO-GAME CONSOLES ARCHITECTURE UNDER MICROSCOPE*. SlideShare. July 11, 2016. URL: https://www.slideshare.net/slideshow/security-offense-and-defense-strategies-videogame-consoles-architecture-under-microscope/63910924 (visited on 12/22/2024).

[54] *Pipeline di Rendering*. URL: https://media.springernature.com/lw685/springer-static/image/chp%3A10.1007%2F978-1-4842-8652-4_1/MediaObjects/523243_1_En_1_Fig1_HTML.jpg (visited on 12/24/2024).

[55] *Texture mapping*. SlideShare. Jan. 1, 2014. URL: https://www.slideshare.net/slideshow/texture-mapping/29622076 (visited on 12/24/2024).

[56] *Resident Evil 1.5 - PS1 - Characters by GR-85 on DeviantArt*. June 4, 2021. URL: https://www.deviantart.com/gr-85/art/Resident-Evil-1-5-PS1-Characters-881591595 (visited on 12/24/2024).

[57] Imgur. *Imgur: N64 vs PSX Texture Mapping*. Imgur. URL: https://imgur.com/aGc8WTE (visited on 12/24/2024).

[58] *Attribute Tessellation — Simplygon 10.2.400.0*. URL: https://documentation.simplygon.com/SimplygonSDK_10.2.400.0/overview/concepts/attributetessellation.html (visited on 12/24/2024).

[59] *Why do Playstation 1 polys jitter when the camera pans?* NeoGAF. Aug. 20, 2014. URL: https://www.neogaf.com/threads/why-do-playstation-1-polys-jitter-when-the-camera-pans.879050/ (visited on 12/24/2024).

[60] *Quinforce Gaming: Mega Man Image*. URL: https://quinforcegaming.wordpress.com/wp-content/uploads/2016/12/psx_vs_n64___megaman_legends___by_elias1986.png?w=736 (visited on 12/24/2024).

General Bibliography

[61] *Sony PlayStation: The Price Heard Around the World - "299" - (E3 1995 Keynote) - YouTube*. URL: https://www.youtube.com/watch?v=ExaAYIKsDBI (visited on 11/13/2024).

[62] *PlayStation History*. URL: https://playstationmuseum.com/history.html (visited on 11/13/2024).

[63] *Official Complete History of PlayStation [HD] - YouTube*. Tags: #history, #ps1, #xbox, #ps4, #pc, #nintendo, #documentary. URL: https://www.youtube.com/watch?v=CEgnzJkuZq4 (visited on 11/13/2024).

[64] *Lista modelli PlayStation® — PlayStation Generation*. URL: https://www.playstationgeneration.it/2011/04/lista-modelli-playstation.html (visited on 11/13/2024).

[65] *PlayStation Serie SCPH-3000 e SCPH-3500 — PlayStation Generation*. URL: https://www.playstationgeneration.it/2011/04/playstation-serie-scph-3000-e-scph-3500.html (visited on 11/14/2024).

[66] *PlayStation Serie SCPH-5000 e SCPH-5500 — PlayStation Generation*. URL: https://www.playstationgeneration.it/2011/04/playstation-serie-scph-5xxx.html (visited on 11/14/2024).

[67] *PlayStation Serie SCPH-7000 / SCPH-7500 — PlayStation Generation*. URL: https://www.playstationgeneration.it/2011/04/playstation-serie-scph-7xxx.html (visited on 11/14/2024).

[68] *PlayStation Serie SCPH-9000 — PlayStation Generation*. URL: https://www.playstationgeneration.it/2011/04/playstation-serie-scph-9xxx.html (visited on 11/14/2024).

[69] *PSone (SCPH-100) — PlayStation Generation*. URL: https://www.playstationgeneration.it/2010/08/psone.html (visited on 11/14/2024).

[70] *PlayStation models*. In: *Wikipedia*. Page Version ID: 1254347140. Oct. 30, 2024. URL: https://en.wikipedia.org/w/index.php?title=PlayStation_models&oldid=1254347140 (visited on 11/14/2024).

[71] *Net Yaroze*. In: *Wikipedia*. Page Version ID: 1239658662. Aug. 10, 2024. URL: https://en.wikipedia.org/w/index.php?title=Net_Yaroze&oldid=1239658662 (visited on 11/14/2024).

[72] Sony Computer Entertainment Inc. *PlayStation Hardware*. First edition. Publication date: August 1998. Confidential Information of Sony Computer Entertainment, not for general distribution. All rights reserved by Sony Computer Entertainment Inc. Unauthorized reproduction prohibited. Sony Computer Entertainment America, 919 E. Hillsdale Blvd., 2nd floor, Foster City, CA 94404 & Sony Computer Entertainment Europe, Waverley House, 7-12 Noel Street, London W1V 4HH, England: Sony Computer Entertainment Inc., 1998.

[73] Rodrigo Copetti. *PlayStation Architecture - A Practical Analysis*. 2019. URL: https://www.copetti.org/writings/consoles/playstation/.

[74] *PSXSPX Specifications*. Detailed specifications and technical information about the PlayStation. URL: https://problemkaputt.de/psx-spx.htm#memorymap (visited on 11/16/2024).

[75] *Sega Mega Drive*. In: *Wikipedia*. Page Version ID: 141076671. Sept. 12, 2024. URL: https://it.wikipedia.org/w/index.php?title=Sega_Mega_Drive&oldid=141076671 (visited on 11/15/2024).

[76] *Nintendo Entertainment System*. In: *Wikipedia*. Page Version ID: 141762635. Oct. 22, 2024. URL: https://it.wikipedia.org/w/index.php?title=Nintendo_Entertainment_System&oldid=141762635 (visited on 11/15/2024).

[77] *Classic RISC pipeline*. In: *Wikipedia*. Page Version ID: 1257250164. Nov. 14, 2024. URL: https://en.wikipedia.org/w/index.php?title=Classic_RISC_pipeline&oldid=1257250164 (visited on 11/15/2024).

[78] *Endianness*. In: *Wikipedia*. Page Version ID: 1233149731. July 7, 2024. URL: https://en.wikipedia.org/w/index.php?title=Endianness&oldid=1233149731 (visited on 11/16/2024).

[79] *Motorola 68000*. In: *Wikipedia*. Page Version ID: 1256024444. Nov. 7, 2024. URL: https://en.wikipedia.org/w/index.php?title=Motorola_68000&oldid=1256024444 (visited on 11/16/2024).

[80] *My Hardest Bug Ever*. URL: https://www.gamedeveloper.com/programming/my-hardest-bug-ever (visited on 11/15/2024).

[81] dbousamra. *PlayStation Emulation Guide*. Gist. Includes a PlayStation Emulation Guide by Lionel Flandrin. URL: https://gist.github.com/dbousamra/f662f381d33fcf5c4a5475c4a656fa19 (visited on 11/26/2024).

[82] Kingcom. *armips: An Assembler for Various ARM and MIPS Platforms*. Accessed: 2024-11-20. Licensed under MIT. Original release date: 2013-10-05. 2013. URL: https://github.com/Kingcom/armips.

[83] Grumpy Coders. *PCSX-Redux: A PlayStation 1 Development and Reverse Engineering Project*. An assembler for various ARM and MIPS platforms. Builds available at http://buildbot.orphis.net/armips/. Licensed under GPL-2.0. Original release date: 2018-12-12. 2018. URL: https://github.com/grumpycoders/pcsx-redux.

[84] *PSX-EXE Format*. URL: https://www.retroreversing.com/ (visited on 12/10/2024).

[85] ARM9. *bin2exe.py - ARM9/psxdev*. Accessed: 2024-11-20. A Python script for converting binary files into PlayStation executable format. 2024. URL: https://github.com/ARM9/psxdev/blob/master/libpsx/tools/bin2exe.py.

[86] Massimo Marchi. *Appunti di MIPS 32*. Capitolo 3: Le pseudo istruzioni. 2014. URL: https://marchi.ricerca.di.unimi.it/Teaching/Architetture14b/Es1/Assembly.pdf.

[87] Università degli Studi di Napoli Federico II. *Programmazione Strutturata: Procedure, Subroutines, Functions*. Materiale didattico disponibile online. 2024. URL: https://www.docenti.unina.it/webdocenti-be/allegati/materiale-didattico/232556.

[88] Automatica e Gestionale Università di Roma - Dipartimento di Ingegneria Informatica. *Rappresentazione dei numeri e complementi didattici*. Accessed: 2024-11-25. URL: http://www.diag.uniroma1.it/~marte/homepage/didattica/complementi.didattici/PRIMA%20PARTE%20-%20complementi%20didattici/3-rappresentazione.pdf.

[89] *Arithmetic shift*. In: *Wikipedia*. Page Version ID: 1221514546. Apr. 30, 2024. URL: https://en.wikipedia.org/w/index.php?title=Arithmetic_shift&oldid=1221514546 (visited on 11/26/2024).

[90] Alessandro Sperduti. *Pipeline e Architetture RISC*. Accessed: November 25, 2024. n.d. URL: https://www.math.unipd.it/~sperduti/ARCHITETTURE-1/pipeline-2.pdf.

[91] Università di Verona. *Architetture dei Sistemi di Elaborazione: Delay Slot*. Accessed: November 25, 2024. n.d. URL: https://www.di.univr.it/documenti/OccorrenzaIns/matdid/matdid557805.pdf.

[92] Wikipedia contributors. *Delay slot*. Accessed: November 25, 2024. n.d. URL: https://en.wikipedia.org/wiki/Delay_slot.

[93] *Nintendo 64 Architecture — A Practical Analysis*. The Copetti site. Section: writings. Sept. 12, 2019. URL: https://www.copetti.org/writings/consoles/nintendo-64/ (visited on 11/26/2024).

[94] Stanford University. *RISC vs. CISC*. Accessed: November 25, 2024. n.d. URL: https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/.

[95] Spiceworks. *RISC vs. CISC: Differences and Advantages*. Accessed: November 25, 2024. n.d. URL: https://www.spiceworks.com/tech/tech-general/articles/risc-vs-cisc/#_001.

[96] *Nerdly Pleasures: The Rise of Interlacing in Video Game Consoles*. Nerdly Pleasures. Oct. 12, 2017. URL: https://nerdlypleasures.blogspot.com/2017/10/the-rise-of-interlacing-in-video-game.html (visited on 11/26/2024).

[97] *An Overview of PlayStation Aesthetics — Jaybee*. Patreon. URL: https://www.patreon.com/posts/overview-of-49375996 (visited on 11/22/2024).

[98] Edward Angel. *Interactive Computer Graphics: A Top-Down Approach with WebGL*. 8th. Pearson, 2022. ISBN: 9780137869761. URL: https://www.pearson.com.

[99] *MIPS architecture*. In: *Wikipedia*. Page Version ID: 1256527461. Nov. 10, 2024. URL: https://en.wikipedia.org/w/index.php?title=MIPS_architecture&oldid=1256527461 (visited on 11/26/2024).

[100] *Stack vs Heap Memory Allocation*. GeeksforGeeks. Section: Difference Between. Dec. 26, 2018. URL: https://www.geeksforgeeks.org/stack-vs-heap-memory-allocation/ (visited on 11/26/2024).

[101] MIPS Technologies, Inc. *MIPS32 Architecture For Programmers, Volume II: The MIPS32 Instruction Set*. 2.00. MD00086. Document Number: MD00086, Copyright © 2001-2003 MIPS Technologies Inc. MIPS Technologies, Inc. Mountain View, CA, USA, June 2003.

[102] *DuckStation: Fast PS1 Emulator*. URL: https://duckstation.org/ (visited on 11/25/2024).

[103] *Downloads — PSXDEV*. Accessed: 2024-12-10. A comprehensive collection of PlayStation 1 development tools, SDKs, and documentation. URL: https://www.psxdev.net/downloads.html (visited on 12/10/2024).

[104] *psx.arthus.net*. Accessed: 2024-12-10. A comprehensive archive of PlayStation development resources, including tools, documentation, and tutorials. URL: https://psx.arthus.net/ (visited on 12/10/2024).

[105] *ps1.consoledev.net*. URL: https://ps1.consoledev.net/ (visited on 12/10/2024).

[106] *no$psx*. URL: https://problemkaputt.de/psx.htm (visited on 12/10/2024).

[107] *Code::Blocks*. URL: https://www.codeblocks.org/ (visited on 12/10/2024).

[108] *Official Playstation 1 Software Development Kit (PSYQ)*. URL: https://www.retroreversing.com/ (visited on 12/10/2024).

[109] *Painter's algorithm*. In: *Wikipedia*. Page Version ID: 1248759290. Oct. 1, 2024. URL: https://en.wikipedia.org/w/index.php?title=Painter%27s_algorithm&oldid=1248759290 (visited on 12/19/2024).

[110] *PlayStation Ordering Table Tutorial*. URL: https://psx.arthus.net/sdk/Psy-Q/DOCS/TECHNOTE/ordtbl.pdf (visited on 12/12/2024).

[111] *Net Yarooze User Guide*. URL: https://psx.arthus.net/sdk/NetYaroze/Net%20Yaroze%20Official%20-%20Startup%20Guide.pdf (visited on 12/12/2024).

[112] *Run-Time Library Overview 4.4*. URL: https://psx.arthus.net/sdk/Psy-Q/DOCS/Devrefs/Libovr.pdf (visited on 12/12/2024).

[113] *Tech Note SCEE: Developers Guide - 2.9*. URL: https://psx.arthus.net/sdk/Psy-Q/DOCS/TECHNOTE/scee_dev.pdf (visited on 12/21/2024).

[114] *Dev Refs Inline Programming Reference*. URL: https://import.cdn.thinkific.com/167815/aT4TYXwuQleC3CYjyzZl_Sony-PlayStation-GTEInlineReference.pdf (visited on 12/21/2024).

[115] *Training (Oct 96) GTE*. URL: https://import.cdn.thinkific.com/167815/sQCykqKxQyqNKGaBDwum_Sony-Slides-GTE.pdf (visited on 12/21/2024).

[116] *Geometry Transformation Engine (GTE) - PlayStation Specifications - psx-spx*. URL: https://psx-spx.consoledev.net/geometrytransformationenginegte/#gte-registers (visited on 12/21/2024).

[117] *Fixed Point Arithmetic and Tricks*. URL: http://x86asm.net/articles/fixed-point-arithmetic-and-tricks/index.html (visited on 12/21/2024).

[118] Verity Townsend. *Final Fantasy's original creator reveals why FFVII was released on PlayStation and not Nintendo 64*. AUTOMATON WEST. Feb. 15, 2024. URL: https://automaton-media.com/en/news/20240215-27218/ (visited on 12/22/2024).

[119] *The Ultimate Guide To PSX CD-Rs*. Alex. URL: https://alex-free.github.io/psx-cdr/ (visited on 12/22/2024).

[120] *PSXSPX CDROM Drive*. URL: https://problemkaputt.de/psxspx-cdrom-drive.htm (visited on 12/22/2024).

[121] *[DOWNLOAD] The Revenge of STRIPISO - PlayStation Development Network*. URL: https://www.psxdev.net/forum/viewtopic.php?f=60&t=997 (visited on 12/22/2024).

[122] *[DOWNLOAD] PSXLICENSE (V1.0) - PlayStation Development Network*. URL: https://www.psxdev.net/forum/viewtopic.php?f=69&t=704 (visited on 12/22/2024).

[123] *PSXSPX CDROM File Playstation EXE and SYSTEM.CNF*. URL: https://problemkaputt.de/psxspx-cdrom-file-playstation-exe-and-system-cnf.htm (visited on 12/22/2024).

[124] Video Games End Stuff. *Behind The Scenes - How PlayStation Discs are made*. Oct. 27, 2020. URL: https://www.youtube.com/watch?v=5rKMeesV1jI (visited on 12/22/2024).

[125] *Dev Refs - File Formats*. URL: https://psx.arthus.net/sdk/Psy-Q/DOCS/Devrefs/Filefrmt.pdf (visited on 12/22/2024).

[126] Sydney Butler. *Why Did the PlayStation 1 Have Wobbly Graphics?* How-To Geek. Section: Video Games. June 15, 2024. URL: https://www.howtogeek.com/why-did-the-playstation-1-have-wobbly-graphics/ (visited on 12/24/2024).

[127] *Graphics Processing Unit (GPU) - PlayStation Specifications - psx-spx*. URL: https://psx-spx.consoledev.net/graphicsprocessingunitgpu/ (visited on 12/24/2024).

[128] *Why PlayStation 1 Graphics Warped and Wobbled so much — MVG - YouTube*. URL: https://www.youtube.com/watch?v=x8TO-nrUtSI (visited on 12/24/2024).

[129] Albert Fornells Herrera. *11. Texture Mapping. Computer Graphics*. Material didattico per grafica computerizzata. ENTI, University of Barcelona, 2016. URL: https://bitbucket.org/afornellsENTI/computergraphics.

[130] *PSXSPX CDROM File Formats*. URL: https://problemkaputt.de/psxspx-cdrom-file-formats.htm (visited on 12/26/2024).

[131] *CDROM Drive - PlayStation Specifications - psx-spx*. URL: https://psx-spx.consoledev.net/cdromdrive/#cdrom-xa-subheader-file-channel-interleave (visited on 12/26/2024).

[132] John "Lameguy" Wilbert Villamor. *Lameguy64/mkpsxiso*. original-date: 2016-08-06T12:53:05Z. Dec. 12, 2024. URL: https://github.com/Lameguy64/mkpsxiso (visited on 12/26/2024).

[133] *PlayStation Development Network - PSX CD-ROM XA (eXtended Architecture)*. URL: https://www.psxdev.net/help/psx_extended_architecture.html (visited on 12/26/2024).

[134] *psx-cue-sbi-collection/redump.org/Tekken 3 (Europe).cue at master · opsxcq/psx-cue-sbi-collection · GitHub*. URL: https://github.com/opsxcq/psx-cue-sbi-collection/blob/master/redump.org/Tekken%203%20(Europe).cue (visited on 12/26/2024).

[135] Gabriele Passuello. *Thesis PSX - Passuello Gabriele*. Repository pubblico su GitHub per la tesi di Gabriele Passuello su Playstation. 2025. URL: https://github.com/gabrilink/Thesis-PSX-Passuello-Gabriele-PUBLIC (visited on 01/03/2025).

[136] Anonymous. *Real-Time Fog using Post-processing in OpenGL*. Accessed: 2024-12-10. George Mason University, n.d. URL: https://cs.gmu.edu/~jchen/cs662/fog.pdf.

[137] OpenGL Notes. *Shading Models*. Accessed: 2024-12-29. n.d. URL: https://opengl-notes.readthedocs.io/en/latest/topics/lighting/shading.html.

[138] Sony Computer Entertainment. *Data Conversion Utilities – PlayStation Developer Reference Series*. Accessed: 2024-12-29. 1998. URL: https://archive.org/details/SCE-DataConversionUtilities-Nov1998/page/n88/mode/1up.

[139] Sony Computer Entertainment. *MDEC Note - Technical Reference for Motion Decoding on PlayStation*. Accessed: 2024-12-29. n.d. URL: https://psx.arthus.net/sdk/Psy-Q/DOCS/TECHNOTE/mdecnote.pdf.

[140] Lameguy64. *STR Player Library for PlayStation Development*. Accessed: 2024-12-29. n.d. URL: https://www.psxdev.net/forum/viewtopic.php?t=507.

[141] Emu-Land Community. *Discussion on Video Encoding for PlayStation*. Accessed: 2024-12-29. n.d. URL: https://www.emu-land.net/forum/index.php?topic=24926.30.

[142] Lameguy64. *Animation Tutorial*. Accessed: 2024-12-29. 2014. URL: https://www.psxdev.net/forum/viewtopic.php?f=51&t=658.

[143] Sony Computer Entertainment. *Psy-Q Developer Reference: 3D Graphics*. Accessed: 2024-12-29. 1998. URL: https://psx.arthus.net/sdk/Psy-Q/DOCS/Devrefs/3dgraph.pdf.

[144] Lameguy64. *TMD Loader and Viewer*. Accessed: 2024-12-29. 2017. URL: https://www.psxdev.net/forum/viewtopic.php?f=64&t=626.

[145] Dominic Szablewski. *Reverse Engineering Wipeout (PSX)*. Accessed on January 2, 2025. 2015. URL: https://phoboslab.org/log/2015/04/reverse-engineering-wipeout-psx (visited on 01/02/2025).

[146] Forest of Illusion. *Wipeout Source Code Leak for PC Port*. Accessed on January 2, 2025. 2022. URL: https://x.com/forestillusion/status/1508048268176990209 (visited on 01/02/2025).

[147] Mixmag. *Wipeout: The First Rave-Inspired Video Game*. Accessed on January 2, 2025. 2021. URL: https://mixmag.net/feature/wipeout-first-rave-inspired-video-game-dance-music-soundtrack-designers-republic-psygnosis (visited on 01/02/2025).