

ReAL Sound: Outline of a Reusable Audification
Library to Improve Game Accessibility for the
Visually Impaired

Zachary Arani
Kyushu University
Graduate School of Design

February 14, 2025

Abstract

In a world of ever-increasing software complexity, there has been a growing demand for interoperable, reusable technologies usable in many problem domains. This is especially true in the world of game development, where tools and architectures change from title to title. At the same time, the complexity and specificity of end-users has also grown immensely, bringing an increased demand for advanced accessibility features. To address these trends, we propose ReAL Sound: the **Re**-usable **A**udification **L**ibrary, which abstracts the creation of accessibility features for visually impaired persons in games via computer vision and spatial audio techniques. Through our proposal, we demonstrate the efficacy of ReAL Sound as a general-purpose tool for implementing accessibility features in games.

Notes on the Final Draft

This final draft sees the overhaul of the 4th chapter—which now features an in-depth look at our implementation. We hope this detailed exploration provides more clarity towards ReAL Sound’s motivation, structure, and ultimate aims. We also made enhancements to our proposal chapter with new visual aids. The introduction section now also addresses topics in inclusive and meta design. Finally, the conclusion sports a more developed future work section—specifically detailing later research plans.

Zachary Arani
February 14, 2025

Contents

1	Introduction	4
1.1	Overview	4
1.2	Recent Trends in Software Development	5
1.2.1	Interoperability, Abstraction, Modularity	5
1.2.2	User Experience Trends	6
1.3	Recent Trends in Audio Technologies	8
1.4	Recent Trends in Game Development	9
1.5	<i>ReAL Sound</i> : Towards a More Universal Accessibility Design . .	11
1.6	Conclusion	11
2	Literature Review	12
2.1	Computer Vision	12
2.1.1	Image Processing and Analysis	12
2.1.2	AI & Machine Learning	16
2.2	Spatial Audio	20
2.3	Games	23
2.3.1	Games and Accessibility	23
2.3.2	Games and Audification	24
2.3.3	Games and Computer Vision	25
2.3.4	Related Work	26
2.3.5	Conclusion	26
3	ReAL Sound	27
3.1	Proposal	27
3.2	Concepts	31
3.2.1	Anatomy of a Game	31
3.2.2	State Models	34
3.2.3	How Players Understand Games	37
3.2.4	Visual Analysis of a Game	41
3.3	Structural Outline	44
3.3.1	Overview	44
3.3.2	Vision Layer	46
3.3.3	Decision Layer	47
3.3.4	Audification Layer	49

3.3.5	Summary	50
3.4	Implementation Process	52
3.4.1	Overview	52
3.4.2	Planning	52
3.4.3	Training	56
3.4.4	Design	57
3.4.5	Execution	64
3.4.6	Conclusion	65
3.5	The End User Experience	66
3.6	Conclusion	66
4	Sample Implementation of ReAL Sound	68
4.1	Background	68
4.1.1	Choosing a Target Game	68
4.1.2	<i>Pong</i> : A Brief Overview	69
4.1.3	Translating <i>Pong</i>	70
4.1.4	Languages, Tools, and Frameworks	74
4.2	Structural Overview	77
4.3	Planning & The Vision Layer	79
4.4	Design & The Decision Layer	88
4.4.1	Code Samples	91
4.4.2	Special Entity Logic	95
4.5	The Audification Layer	100
4.5.1	Code Samples	101
4.6	Execution: Building an Application	105
4.7	Conclusion	106
5	Conclusions	107
5.1	Limitations	108
5.2	Future Work	108
5.3	Thanks	110

Chapter 1

Introduction

1.1 Overview

In this chapter we introduce the historical and environmental trends that contextualize and motivate our research. We begin with a discussion of recent advances in the general software industry—primarily its drive for more universal design¹ principles. We also take time to discuss new trends in user experience design, where we touch on the modern accessibility movement.

Building on this theme, we then take a turn to discuss the recent boom of AI technologies and how it relates to the course of software development as a whole. We also consider how new AI-powered technologies and design techniques have impacted the field of software as a whole.

Taking a turn, we make a quick detour to discuss recent advances in audio technology. We consider how these trends relate to the previous sections through the framework of universal and accessible design.

We then produce a variation on this theme with a discussion of the modern state of game development. We consider the problem of universal design within gaming and explore contemporary problems in game accessibility.

Finally having provided the relevant context to motivate our research, we argue for the need of a universal framework to develop accessibility tools—particularly in the realm of game development. We choose to narrow our scope—focusing on the problem of generalizing accessibility features for the visually impaired using modern computer vision, AI, and spatial audio technology.

We then propose our own design to address these problems: ***ReAL Sound***: the **Re**-usable **A**udification **L**ibrary—a universal real-time framework for transforming a game’s visual information into spatial audio using computer vision and AI technology—making games more accessible to the visually impaired.

¹To be clear, we use the term ‘universal design’ in this thesis when referring to ‘platform-agnostic, generalized, and abstracted design principles in computer technology.’ We are not referring to ‘universal design’ in the sense of ‘inclusive design principles.’

1.2 Recent Trends in Software Development

1.2.1 Interoperability, Abstraction, Modularity

The Diversification of Computer Technologies

As computer technology has advanced, so has its usage contexts. The static mainframes of 1960s laboratories quickly evolved into the homeward-bound personal computers of the 1980s, and eventually into the 'smart' devices that now line everything from our pockets to our televisions to even our refrigerators. This societal shift from monolithic computing styles to a diverse multi-modal paradigm has also created new challenges for designers.

The old conceptualization of immutable hardware configurations (dichotomies such as 'PC vs Mac,' 'Intel vs PowerPC,' etc.) has been replaced with a broad array of operating systems (**iOS**, **Android**, **Windows**, **MacOS**, **watchOS**, **WearOS**, **Linux**², **ChromiumOS**, **Solaris**, etc.) and hardware architectures (**x86**, **ARM**, **RISC-V**, **FPGA**, etc.) which are often broken down even further into distinct sub-categories³.

The Rise of Universal Software Design

This trend highlights a key problem area in contemporary tech: the need for modular, cross-platform, and *universal* software and design. It would not be an overstatement to say that billions-upon-billions of dollars have been spent over the last decade in an industry-wide attempt to transition core technologies to agnostic designs. Microsoft has undertaken a multi-decade campaign to phase-out classic Windows 32-bit applications [42], while Google opted to create an entire operating system—**ChromiumOS**—which essentially functions a massive, OS-level web-browser to sidestep the problem entirely [122].

In a similar fashion, nascent startups have frequently leaned towards platform-agnostic frameworks and technology-stacks. The **Electron** software framework allows the creation of cross-platform applications using standard web-development tools such as **JavaScript**, **HTML**, and **CSS** in a fashion similar to ChromiumOS. It would not be unfair to say that Electron's web-focused paradigms ushered in the modern era of application design [5]. It is also not hard to see why, as cross-platform applications which once demanded dozens of low-level technical programmers and leagues of platform specialists could now instead be developed by a handful of web-developers. This has had a large impact on the current state of technology, as JavaScript has functionally replaced C++ as the *lingua franca* of young developers, while modern application—from **Slack** and **Skype** to **Visual Studio Code** and **Atom**.

As software has become platform-agnostic, so too has its development tools. Now-ubiquitous technologies like **Docker** avoid the need for platform-specific

²It would be more accurate to say Linux *derivatives*—such as **Ubuntu**, **RedHat**, **Debian**, **Arch**, etc.

³For an example that has had immense impact on modern computing, see the **x86 32-bit/64-bit** schism.

considerations by simply packaging the platform as a *part* of the software itself—a paradigm now termed *Platform as a Service* (**PaaS**) [68]. This enables developers to avoid the infamous “*works on my machine*” dilemma [161]—where software issues appear only on platform configurations not used in development. This trend of *containerization* and *virtualization* has also had massive implications for software development and its future—expediting the diffusion of cross-platform technologies used in everything from standard computers to complex, IoT-enabled embedded devices. As proof of its success, we should note that Docker and its derivatives are now in-use by virtually every major software vendor: Microsoft, IBM, Google, Cisco, Meta, etc.

1.2.2 User Experience Trends

While these hardware trends have kept developers on their toes, they have offered end users a rich array of computing options. An evolution of user experience ⁴ expectations has naturally followed as different styles of computing (smartphones, smartwatches, smart TVs, tablets, smart home devices, etc.) achieved mass-adoption. Although the totality of UX trends are beyond the scope of this thesis, we must highlight the clear demands and expectations users have for cross-platform design.

Towards a Universal User Experience

But despite the trend towards universal design, numerous real-world applications have yet to successfully realize this ideal—necessitating platform-specific software. The experience of browsing a popular social media platform such as **Instagram**, for instance, is inherently different on a computer web browser versus a native mobile application. More infamously, the MacOS version of Microsoft’s **Office Suite**—which has been developed independently of the more well known Windows version since 1989—has no doubt confused many users making an operating system switch.

These platform-specific incongruities have often the target of user anger [114], as most users expect the features and quality of their experience to be preserved when they change usage contexts. This ever-present sense of user ire has become another key motivator behind the generalization of computer software. Instagram, to continue with the previous example, lacked a method of accessing the platform on windows devices until overwhelming user demand necessitated its creation [155]. In a similar (and more humorous) vein, the iPad’s omission of a standard calculator app became such a fixation of user frustration as to be headline news [156] when Apple finally debuted a native app in 2024. These emerging behaviors clearly demonstrate the need for universal design—not only for developers of software, but also their end-users.

Inclusive Development These trends extend even into software development, where there has been a growing call for accessible tools [84]. Although the image

⁴UX, a shorthand for User EXperience is used interchangeably hereafter.

of a 'software user' usually evokes a sense of novice-to-moderate tech-saviness, software development also requires tools (software) that developers (users) employ in their work. Advocates for **inclusive development** emphasize the importance of new and diverse software *metaphors*, *objects*, and *interfaces* that empower more users in creating accessible software [133]. Similar sentiments are echoed in the popularization of **meta-design**—creating design *for* designers [40]. In this sense, we can see that the next generation of software should de-emphasize the role of users as simple 'consumers' of static software—and should instead empower them to easily create, design, and express themselves via their media environments.

The AI Boom

Overlapping the trend towards universal design has been the sudden explosion of AI technologies in recent years [66]. Of course, there are many reasons to be skeptical of AI's sudden explosion in popularity, as AI research has been notorious for its seasonal relevance in the past. The term *AI Winter*—which describes the long droughts of public attention and spending towards AI between milestone research achievements—has been in the technical lexicon for decades now [54]. Despite this, it is hard to deny the widespread changes this most recent *AI Summer* has brought with it. And so, we shall introduce some relevant topics from the boom—namely through advances in machine learning.

Machine Learning Realized

Although machine learning has its roots in the 'cognitive revolution' of the 1960s [100] and produced notable results in early years such as the **ELIZA** chatbot of the 1960s, [157] the field essentially remained dormant until the turn of the 2010s—when modern technological advances and extensive collaboration efforts made feasible the AI theories proposed in earlier decades. The **ImageNet** database played a large part in kickstarting the modern AI boom by providing large sums of well-collected data necessary in creating modern AI [50]. Other research highlights often came from the labs of big-tech companies such as Netflix—who previously awarded *The Netflix Prize* to incentivize AI research [9]—and Google, who's seminal 2017 paper *Attention is All You Need* [151] revolutionized the field, ushering in the presently-dominant GPT era [80].

The ubiquitous popularity of ImageNet highlights a key feature of the AI boom—a focus on vision and visual media. While AI is used in a variety of problem domains—from recommendation engines to text prediction—vision and other sight-related tasks have had a tenacious popularity. The problems of object detection and object classification in particular have endured as areas of consistent growth. Many tasks considered crucial by modern science—such as facial recognition, tumor detection, and autonomous driving—are deeply entangled with object detection and classification.

Computer Vision

Although computer vision describes any process of modeling human sight using any computational means [61], it is often tied to the fields of AI and machine learning. Consequently, computer vision has also received much of the same boost in popularity and accessibility as machine learning has. The **OpenCV** library in particular has significantly streamlined computer vision implementations and has become a staple of modern computer vision development [29], as we will see in later chapters.

The continued improvement and diffusion of high-performance graphics cards (**GPUs**) has also had a hand in popularizing computer vision. GPUs are not required for computer vision or AI tasks, but their efficiency in performing *single instruction, multiple data* (**SIMD**) tasks makes them incredibly appealing for AI research—whose processing targets are often SIMD by nature. Processors custom-built for AI workloads—such as neural processing units (**NPU**s)—are generally more performant, but the GPUs ease-of-access for the home consumer (and average software developer) has cemented its status in AI history. This is mainly because the GPU had already reached wide-spread consumer adoption in the years prior to the AI boom—largely owing its success to an association with the ever-popular PC gaming market.

1.3 Recent Trends in Audio Technologies

Somewhat parallel (yet mainly orthogonal) to the previous trends has been new developments in audio technologies—primarily in the realm of 3D and spatial audio.

The Death and Rebirth of Spatial Audio

Spatial audio—that is, audio generated to give the impression of sound naturally emanating from around the listener—has its modern origins in the 1960s and 70s.

The popularization of the stereo sound format spawned many technologies which attempted (and failed) to overtake it. Quadraphonic sound—which utilized four speakers instead of two—significantly improved the reproduction of localized sound, but ultimately failed due to a myriad of technical shortcomings and disinterest from artists [36].

Spatial audio found a slightly stronger footing in the 1970s with Ambisonics—which abstracted sound into a speaker-independent format that was decoded at runtime to suit the listener’s speaker system. However it was not until the advent of VR technology in recent years which transitioned Ambisonics from a niche enthusiast standard to mass-market product. Ambisonics remains the format of choice for many players in the tech industry—from Google to Meta [77][96].

Coincident with the rebirth of spatial audio through VR technologies was the introduction of Dolby’s **Atmos** surround sound technology. Atmos extended

the traditional surround sound setup with new height channels—allowing three dimensional sound—and the conception of *Audio Objects*, which engineers design and ‘place’ around the listener’s position [72]. As Dolby is a titanic figure in the audio market, it would only take a few years before Atmos reached mass-adoption in the market—leading to Atmos-branded support in everything from soundbars to game consoles to smartphones [28].

These two trends—the mass-market adoption of VR technology, as well as the proliferation of the Dolby Atmos standard—have ushered in a new era of spatial and 3D audio. Numerous software libraries have incorporated extensive support as a consequence. We will examine the developmental nature of spatial audio in later chapters.

1.4 Recent Trends in Game Development

Gaming has often been at the forefront of technological progress—from real-time texture mapping techniques in *Virtua Fighter 2* [76], to novel solutions to the fast inverse square root problem found in *Quake III Arena* [79], to contemporary advances in VR and AR technologies. It should come as no surprise that the above trends also ring true for the world of gaming. In fact, one could easily argue that many of the problems seeking the solutions above are even more pronounced in games.

The Quest for Reusability

Take, for instance, the unique hardware and software requirements often found in games. Prior to the introduction of the smartphone, most games were consumed via bespoke hardware that was rendered obsolete only a few years after release. Although systems like the **Playstation 3** featured backwards compatibility⁵, the console itself bore no resemblance—in terms of development workflows or architectural design—to its progenitors [71]. In fact, most systems (including the Playstation consoles as well as the Nintendo **Wii**) accomplished backwards compatibility by packaging the entire previous generation’s processors as a sub-component of the new system⁶ [24] [25].

Game software, meanwhile, faces a very similar dilemma. For most of gaming history, game development frameworks (generally termed **game engines**) were bespoke and remained a corporate secret [52]. Engines were often abandoned at the end of a console generation, if not at the end of a single title’s development. Even developers who iteratively improved an engine over the course of multiple generations have been known to discard entire tools and frameworks at a moment’s notice—rebuilding entire codebases from scratch.

⁵It would be more accurate to say the system *originally* featured backwards compatibility, as the feature was infamously removed in subsequent revisions.

⁶Although this is a known fact in the case of the PlayStation 3, we should note the Wii’s architectural similarities to the GameCube have never been formally confirmed to be exactly identical by Nintendo, IBM, or ATI/AMD.

A growing trend of reusable game development software began in the 90s with the advent of 3D games. The inherent complexity of 3D game development made consistent engines an attractive prospect, which was further bolstered by the massive success of first person shooter titles *Quake* and *Unreal*. Both games were developed with high-quality engines which soon proliferated the market—becoming pillars of the modern game development world⁷.

This trend intensified in the 2000s via the rise in middleware and shared library tools which streamlined other aspects of development. Audio software like **fmod**, physics sub-engines such as **Havok**, and CRI’s **Sofdec** video decoder serve as noteworthy examples [41]. The introduction of easy-to-use public-facing game engines such as **Unity** as well as **GameMaker** in the 2010s have only furthered the push towards universal design [22]. As of the time of writing, it would not be an understatement to say that a presumed majority of the game industry is making use of either the Unity or Unreal engine for development [127]. Even in Japan—where bespoke and company-internal tools have endured in popularity—we have seen similar shifts, as companies like Square Enix have opted to replace their own internal engines with Unreal [136].

A More Accessible Future for Games

The diffusion of engines like Unity coincided roughly with the introduction of entry-level smartphones, which spawned a new market of so-called *casual games* as well as a newfound cottage industry for independent game development. The casual game segment introduced gaming to large demographics who previously found games inaccessible, while indie games often had a niche appeal that attracted unorthodox fan bases. Both market segments are generally considered to have had a democratizing effect on games—opening up the medium for audiences who had previously felt ‘unseen’ by the industry at large. As both markets grew to multi-billion dollar valuations, a greater push for games accessibility gained momentum.

This shift towards accessibility in games has percolated all the way up to blockbuster *triple-A* titles such as *The Last of Us: Part II* and *Assassin’s Creed: Valhalla*, both of which feature extensive accessibility features to accommodate for a wide variety of disability and user preferences. The largest game industry award show—*The Game Awards*—has even designated an entire award to celebrate yearly accomplishments in accessibility. Other significant milestones, such as Microsoft and Sony’s release of accessibility-focused devices has cemented the industry-wide recognition of accessibility in games. Now, entire teams and companies operate entirely to solve problems of accessibility within gaming.

⁷While the Unreal engine’s impact is still evident to modern readers, we should note that the Quake engine spawned numerous derivations—which have powered everything from *Call of Duty* games to the *Half-Life* and even recent hits like *ApeX: Legends*

1.5 *ReAL Sound*: Towards a More Universal Accessibility Design

If we unite the trends of previous sections, a harmonic theme begins to manifest: the need for universal, accessible design in not only game software, but also game development. This has remained a complex problem even into the current era, but the swift maturation of AI, computer vision, machine learning, and spatial audio technologies has provided us a clear set of tools to address this shortcoming in the medium of video games.

And so, we aim to contribute to the fields of games accessibility and inclusive design in this thesis with our own novel contribution—*ReAL Sound*: the **Re**-usable **A**udification **L**ibrary. *ReAL Sound* is a framework for streamlining and generalizing the process of developing game accessibility features for the visually impaired. To accomplish this, we make use of cutting-edge computer vision and spatial audio techniques. Through this thesis, we endeavor to explain *ReAL Sound*’s theoretical basis, application value, and implementation process. We then consider a implementation of *ReAL Sound* and examine its real-world utility.

1.6 Conclusion

In this chapter, we examined recent trends around numerous fields of software. We began with a look at the general software trends in universal design before moving to a mediation of the modern AI boom. Following this, we examined the primarily orthogonal trends in contemporary spatial audio technology before transitioning to a discussion of the current state of game development, highlighting how it echoes themes from earlier sections. Finally, we synthesized these trends into a clear problem statement: the need for improved universal design in games accessibility software. To address this perceived shortcoming, we proposed *ReAL Sound*: the **Re**-usable **A**udification **L**ibrary library, which combines recent advances in computer vision and spatial audio to streamline and generalize the development of accessibility features for the visually impaired.

The Following Chapters

In the following chapter, we conduct a brief review of relevant theory and research in connected fields. We then propose *ReAL Sound* in the following chapter. Afterwards, we demonstrate *ReAL Sound* through a sample implementation in chapter four before concluding our thesis in chapter five.

Chapter 2

Literature Review

Introduction To accomplish the goals set by our proposal, we must build on research and theory stemming from a wide range of disciplines. And so, to contextualize ReAL Sound, we first review the relevant theory and literature from the fields of computer vision and AI, acoustics and audio processing, and finally games and accessibility.

2.1 Computer Vision

Computer vision has matured in recent years from a niche experimental field into a veritable juggernaut of computer science. The multiplicative nature of vision naturally spawned numerous sub-fields of study—from the **image processing and analysis** of two dimensional image data, to **machine vision** used in robot guidance systems, to **imaging** techniques that have become a cornerstone of modern medicine, to **pattern recognition** techniques which drive facial detection algorithms [34].

We narrow our scope here to reviewing the relevant disciplines of image processing and pattern recognition.

2.1.1 Image Processing and Analysis

Image processing is a storied area of computer science research—with notable work dating back as far as the 1950s [45]. The primary goal of image processing is to translate visual data—specifically that of a two dimensional image—into digital data [86].

Defining a Digital Image

First, a real-world **panchromatic image** is conceived as a two dimensional function of light intensity: $f(x, y)$. As function return value indicates the brightness of the image at the point (x, y) [117]. The goal then becomes to find a mapping between this function and another type of $f(x, y)$ —which describes a

digital image, whose values have been transformed from the continuous nature of reality to the discrete nature of digital computers.

This process is achieved in a number of ways. In hardware, specialized sensors such as a **CCD** (Charge-Coupled Device) or **CMOS** (Complementary Metal Oxide Semiconductor) perform this translation by reacting to changes in real-world lighting conditions [10]. These sensors are found in everything from advanced medical technologies to the smartphone in your pocket.

In the realm of software, a digital image is generally defined as a combination of **pixels**, equating to the aforementioned $f(x, y)$ value above. Each pixel is typically made up of numerous **sub-pixels**, which usually equate to discrete numeric representations of the primary colors—**Red, Green, Blue**—giving name to the **RGB** representation of an image. The combination of sub-pixels defines the ultimate output of a pixel—a color with a given luminosity value [38]. The representation of a digital image as a mathematical function of two variables has allowed for many useful techniques to streamline image representation and compression—such as the **Discrete Cosine Transforms** used in ubiquitous file standards like **JPEG** [131].

Understanding a Digital Image

But even though digital images are no doubt necessary for our purposes, it hardly brings us any closer to achieving ReAL Sound. We will also need methods of analyzing digital images. This problem is often referred to as **image segmentation**. As Yu et al. described in a sweeping review of the discipline, image segmentation "divides images into regions with different features and extracts regions of interest (ROIs). [162]" These regions strive to be 'meaningful,' but present the problem of rigidly defining meaning.

Simple Methods There exist many techniques to segment an image. Basic forms such as **thresholding** simply compare a pixel against one or a few trivial criteria. For example, imagine a threshold algorithm which produced a striking, high-contrast black-and-white image. If we consider pixel values to be in the range of $[0, 255]$, then our algorithm $t(x, y)$ may look something like this:

$$t(x, y) = \begin{cases} 0 & f(x, y) \leq 128 \\ 255 & f(x, y) > 128 \end{cases}$$

More complex techniques, such as **k-means clustering** work to 'cluster' the image data into k discrete clusters [78]. This is accomplished via an iterative process of partitioning data points into k groups (termed **clusters**) organized around a center point (termed a **centroid**). The validity of clustering is considered by averaging the value of clustered points, which updates the centroid's position. The process is then run again—repeating until a satisfactory segmentation has been achieved.

Methods like k-means allow for the segmentation of images into 'meaningful'

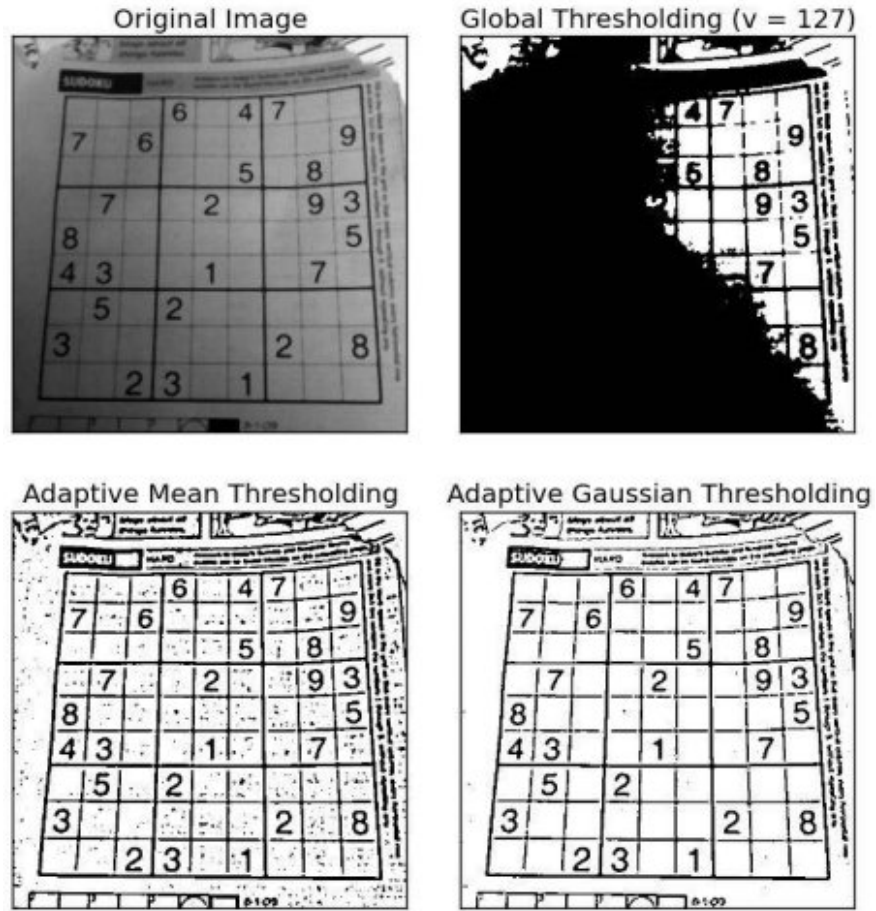


Figure 2.1: Demonstration of image thresholding techniques [111].

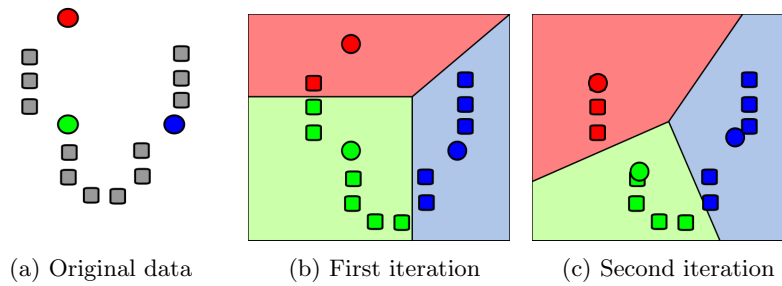


Figure 2.2: A demonstration of K-means clustering with three centroids [158].

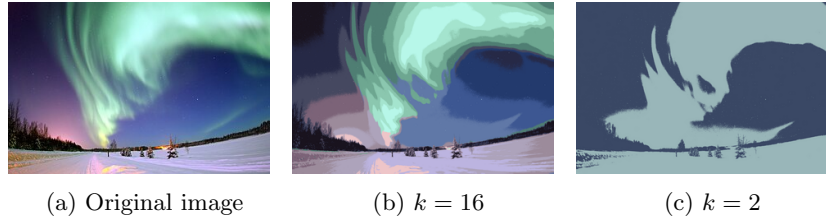


Figure 2.3: Segmentation of a digital image. [144].



Figure 2.4: Demonstration of edge detection techniques [63]

partitions. These clusters may correspond to useful semantics—such as the separation of colors in an image—which may be modified or analyzed independently.

Advanced Methods Moving beyond simple clusters, techniques have also been developed to analyze the ‘edges’ of objects within an image. Using methods of **edge detection**, it becomes possible to assess the physical shapes and locations of objects within image data [166]. This is done by considering the ‘rate of change’ of data within an image. For example, consider a black-box framed as the center of an (otherwise white) image. The pixel value at the edge of the box would be something like:

$$f(x_e, y_e) = 0$$

But the pixels surrounding it—the white which lies outside the box—instead have the values:

$$f(x_e + 1, y_e + 1) = 255$$

This sudden shift in value is detectable using methods rooted in calculus such as **image derivatives**. [56]. Although the specific mathematics driving edge detection are outside the scope of this paper, we must take a moment to talk about a relative field of study—**corner detection**.

As one may guess from its naming, corner detection runs parallel to edge detection—working to assess the corners of objects within an image instead of

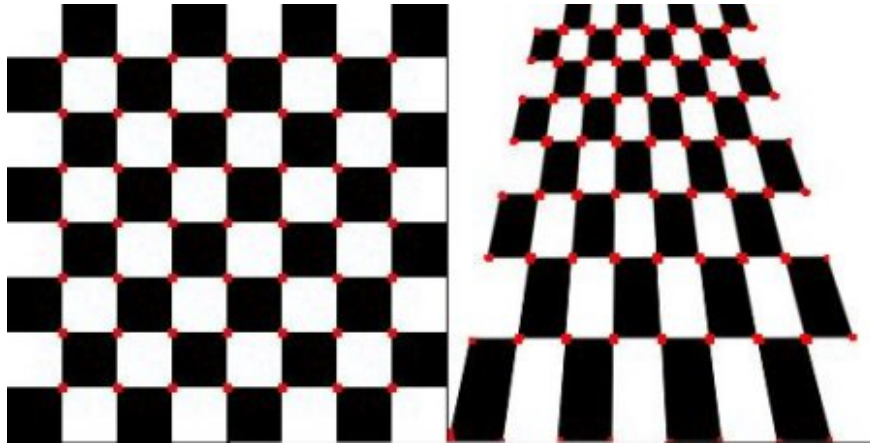


Figure 2.5: Demonstration of corner detection techniques [109]

entire edges. Similar (in abstract) to edge detection, corner detection calculates edges using image derivatives—with the additional requirement of large shifts in both the x and y directions of an edge. Or, to use the formal phrasing, corners are points with a low '*self-similarity*'. [90]' Methods of corner detection were popularized by Harris and Stephens in 1988, as we will see in later chapters [55].

The Diffusion of Computer Vision

Using these methods (and many others left unexplored in the interest of brevity), the act of analyzing a digital image—particularly for its regions of interest—becomes possible. Contemporary open source libraries such as **OpenCV** [12] have accelerated the spread of computer vision techniques by providing an extensive and high quality collection many classic computer vision algorithms. We will continue to review computer vision topics in later chapters. For now, we turn to AI and machine learning.

2.1.2 AI & Machine Learning

It is clear that providing even a brief overview of AI literature is beyond our scope. Consequently, we will be focusing on topics of relevance in the fields of machine learning.

Machine Learning: A Brief Review

Machine learning (**ML**) is, in its essence, the marriage of statistical modeling techniques and computer technology [85]. ML centers around the development of **models** which serve to predict an output based on a particular input. More specifically, the goal of a model—once completed—is to predict output *without* the helping hand of its creator.

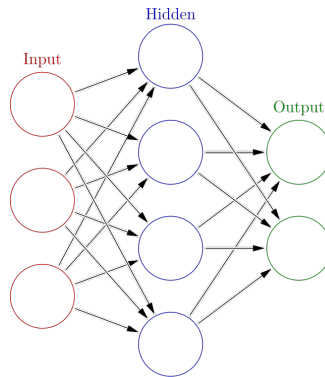


Figure 2.6: A basic neural network model [51].

This requires the model to be '*trained*' with a series of known inputs and outputs. When first constructed, the model is essentially a collection of random functions—turning any given input into functionally random output. As training data inputs are fed into the model, each (again, initially *random*) output is compared against the known 'correct' outputs found in the training data. The inner-workings of the model's functions are then slightly adjusted based on the model's output prediction error [64].

Generally—given enough statistically significant training data—a model may eventually become 'trained.' This trained model is capable of taking any given input and producing a good approximation of output. This is because the internal functions of the model—which once spewed random noise—were slowly 'shaped' over time (via the training data) into functions that actually fit the nature of the model's question [85].

Neural Networks Many machine learning paradigms extend this simple concept with additional theories and stipulations. Most notable is the model of a **neural network**¹ and its extensions **deep learning** and **reinforcement learning**. Neural networks frame the issue of machine learning in terms of the biological neural networks found in animal brains. Each input **neuron** makes connections to other neurons, eventually reaching a series of output neurons. The connections between neurons are the model functions described above. Neurons are traditionally grouped into several **layers**: the **input** and **output** layers, as well as any number of **hidden** layers which lie between the two. The name stems from its vague similarities to brain structure—each neuron forming convoluted connections to all of the neurons in surrounding layers [35].

Deep Learning A neural network is said to feature deep learning when using many neuron layers, as well as a wealth of training data. This is because

¹Also referred to as an **Artificial Neural Network (ANN)** ²

the ‘deep’ nature of deep learning models’ neuron structures are capable of learning things ‘deeply.’ Or, in other words, capable of learning ‘representations of data with multiple levels of abstraction [73].’ The model’s ability to predict several layers of semantic nuance has made them a good fit for multi-modal and multi-contextual tasks such as image and text classification [101].

Reinforcement Learning Reinforcement Learning, meanwhile, bucks the trend of training data altogether. Instead of providing static, prepared training data, reinforcement learning models provide **reward** metrics. When a model makes an output prediction, its accuracy is evaluated and a proportional reward signal is sent to the model. The model then adjusts its internal functions based on the reward signal—constantly attempting to maximize received rewards [65]. The **NEAT** (**N**euro**E**volution of **A**ugmenting **T**opologies) method [143] is one notable implementation.

MarI/O We will now review a simple example to reinforce our learning. Injecting some levity and topicality into our discussion, we consider a machine learning model trained to play the ever-popular *Super Mario World*. In 2015 *SethBling*³ demonstrated a machine learning model that successfully played *Super Mario World* [59] in a video viewed by over 11 million people, garnering significant press coverage [141] [46]. Hendrickson’s model—named **MarI/O**⁴—is based on the NEAT method discussed above.

What constitutes ‘input,’ ‘output,’ and ‘reward’ naturally differs significantly depending on the model. In the case of MarI/O, the model receives a simplified representation of the game-world (designed by Hendrickson) as input and produces a in-game command as output. The model’s input consists of key data such as the location of Mario, enemies, and items, while the output is a simple button press on a controller such as **UP** or **A**. The reward function used for reinforcement is based on how far Mario progresses within the given level. This is an incredibly simple metric to measure—as most *Super Mario World* stages move purely left-to-right. In other words, the further Mario ‘moves to the right’ within a level, the more rewarded the model is.

Hendrickson allowed the model to train for 24. To improve training efficiency, he forced a level reset if Mario failed to move within a given time window. At first, the model produced only poor results—often remaining motionless. Eventually, the model randomly produced a **RIGHT** output, which moved Mario forward. This decision received a large reward, altering the model’s internal functions in ways that biased it to continue outputting **RIGHT** commands. Although a successful start, the model failed whenever Mario reached obstructing objects. Eventually, a random combination of **RIGHT** directional inputs and **A** button presses allowed Mario to progress further in the level—giving the model additional rewards and further altering its internal functions.

³A noted online personality in the realm of games and computer science. Real name Seth Hendrickson.

⁴Source code available at [60].

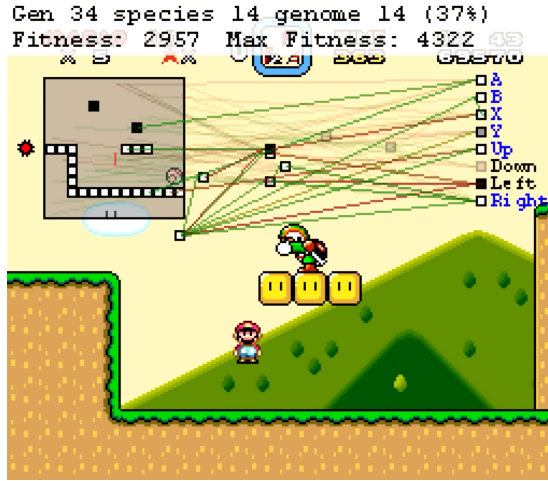


Figure 2.7: **MARI/O**. A model representation of the neural network is shown on the upper portion of the screen. The left side displays the simplified representation of game data provided to the model as input, while the right side shows the model’s controller output decisions. Information about the state of reinforcement learning is shown in the top of the display [60].

We can now see how MARI/O achieved mastery of its task—by producing random outputs until the outputs are deemed successful, which alters the model’s behavior. With enough iterations (and using reasonably valid assumptions), it becomes possible to train a litany of models to produce good output estimations over an even wider array of problem domains.

Object Recognition and Classification

Most crucial for our research are the advances made in **object recognition** and **object classification**. One may imagine how the research discussed above empowers these tasks. Recognition models are usually trained on massive data sets featuring thousands of images. These images are **annotated** with **features** that the model is trained to recognize [154]. For example, a face detection model would likely train on images featuring human faces. Each face is manually labeled and perhaps includes other key characteristics (age, gender, race, etc.) used in training and tuning the model. A small sub-set of training data is usually unused in training, instead being set aside to ‘test’ the model’s post-training validity. The trained model is considered valid if it produces output matching known ‘correct’ output in the testing data [165].

We will explore these topics further in the following chapters. For now we turn our attention to other fields of relevant research.

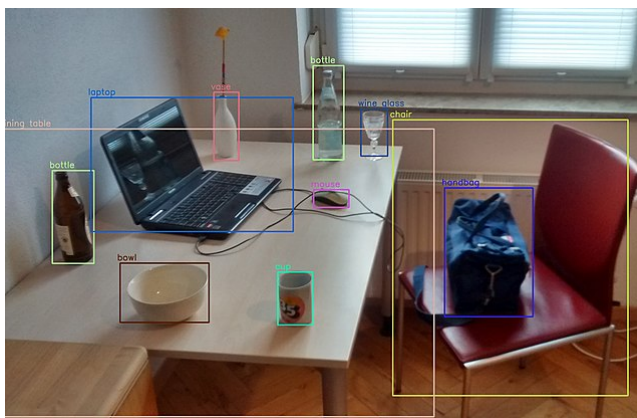


Figure 2.8: Demonstration of object detection and classification using **YOLO** [103].

2.2 Spatial Audio

In this section we explore theory and research related to **auditory displays**—from **audification** and **sonification** to **spatial audio** and **3D Audio**.

Auditory Displays, Audification, and Sonification

Our work deals with **auditory displays**—systems which employ “the use of sound to communicate information about the state of a computing device to a user [93].” More specifically, we utilize **sonification** techniques. Sonification was defined by Scaletti as “a mapping of numerically represented relations in some domain under study to relations in an acoustic domain for the purposes of interpreting, understanding, or communicating relations in the domain under study [69].” In simpler terms, sonification is the translation of information from other domains into audio⁵. Sonification is a complex research field that combines “psychological research in perception and cognition” with the development of computer “tools” and software “design [70].” Many core research topics overlap with physiology, psychology, acoustics, cognition, signal processing, and computer science.

We will now review a few concepts from the field relevant to our research.

⁵We should note that the definitions of audification and sonification are somewhat vague in the literature. As seen in [152], audification is sometimes considered a subset of sonification—one that deals with the *direct* one-to-one translation of non-audio data into audio data. However, as these terms maintain some flexibility, we have opted to use ‘audification’ in our research.

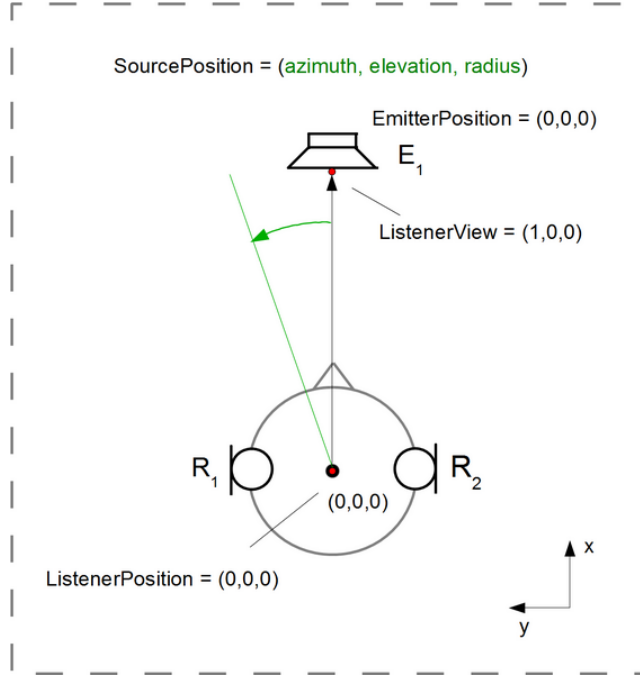


Figure 2.9: Visual demonstration of the Head-Related Transfer Function [116]

Sound Localization

Most key to our research is the topic of **sound localization**. Humans, along with numerous animals, are capable of localizing the origin of sounds in three dimensions [97]. **Interaural time-difference cues (ITDs)**—the difference between a sound signal as it reaches each ear indecently—was devised as a model for understanding sound localization. Lord Rayleigh proposed theories of sound localization in horizontal planes termed **duplex theory**, which ITDs model. In essence, it is observable that the difference in *travel-time* (i.e., path length) between a sound's origin point and each of the listener's ears produces a distinct localization of the sound in the human mind [97]. ITDs were later modeled by Woodworth mathematically:

$$ITD = \frac{a}{c}[\theta_{rad} + \sin(\theta_{rad})]$$

Where a is the radius of the relevant sphere (i.e., the human head), c is the speed of sound, and θ being the angle of the sound relative to the listener [160].

The Head-Related Transfer Function

As one may imagine, the specific characteristics of localization depend on the physiology of the listener. Not all heads and ears are made alike, after all. The

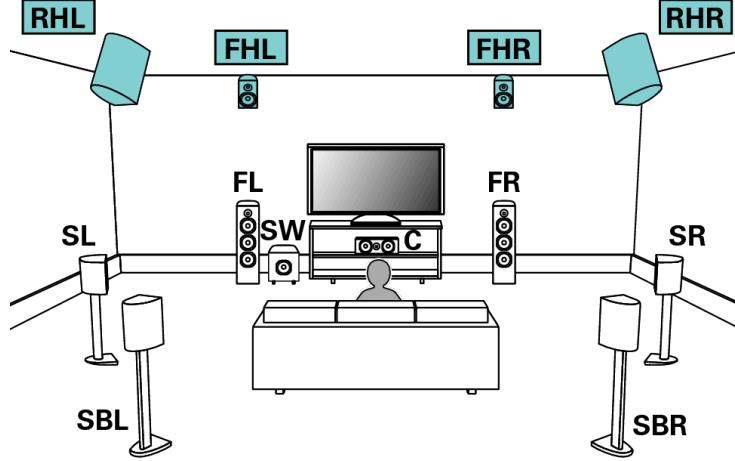


Figure 2.10: Sample 3D sound environment [32]. Note the height channels labeled in blue.

Head-related transfer function (HRTF) models this behavior [75]. The specific derivations of an HRTF are beyond the scope of this thesis, but we must note a few key observations. A high-quality HRTF is required for good spatial audio reproduction. However, a challenge lies in the fact that HRTFs are unique to the individual—and must be calculated per-person if precise results are desired. Consequently, spatial audio technologies supply ‘average-case’ HRTFs, which are sufficiently effective in general situations. HRTFs also presume an ‘open-field’ (that is, a room without any acoustic characteristics), meaning that additional acoustic information must be provided after-the-fact [102].

Spatial Audio and the 3D Audio Effect

Recent advances in spatial audio technology have married these two details—precise, research derived HRTFs, and **room modeling** (the simulation of acoustics) [140]—to great success. As we described above, humans localize sound in *three* dimensions—meaning **hight channels** are necessary to accurately produce a **3D audio** effect. We are, of course, referring to the **channels** used in modeling **surround sound** technology—the more common **center** and **surround** channels are also needed to produce 3D audio [134].

Contemporary research from laboratories such as Dolby have produced technologies such as **Dolby Atmos**, which incorporates hight channels, room (i.e., acoustic) metadata, and a conceptualization of **audio objects** to ease the generation of 3D audio [118]. Non-surround sound technology also makes use of Atmos and similar 3D audio branding. To achieve a similar effect, HRTFs are used to estimate and exaggerate spatiality, combined with room metadata and

audio objects [33].

Many advances in modern spatial audio originate from games-adjacent research, which we will cover in the next section.

2.3 Games

We at last arrive at the core field of our research—games. We shall begin with a review of relevant accessibility research before considering similar work in the realm of audification and computer vision before finally concluding this chapter.

2.3.1 Games and Accessibility

As described in the previous chapter, there has been an increasing trend towards accessibility in the games industry. One industry survey—commissioned by the causal game company *PopCap*—found that over 20% of their userbase identified as disabled [53]. Similar examinations have shown games to be a popular pastime for disabled persons, despite high barriers of entry—especially in action-focused genres [164]. The trend eventually received headline attention in 2015 when the United States of America’s Federal Communications Commission imposed legal requirements for improved accessibility on the gaming industry [128].

Naturally, there has also been a growing body of research on games and accessibility. Previous work has laid the groundwork motivations for game accessibility research [98] [49] and critically examined barriers to widespread adoption [119]. Others have sought design of bespoke accessible games [83], while many more have proposed bespoke after-market modifications of existing games [6]. Others still have performed comprehensive surveys of accessibility features in popular titles as well as user impressions of these features—identifying middleware as a ‘critical link’ in the ‘chain of accessibility’ [119].

Defining Games Accessibility

Previous work has endeavored to formally define and categorize games accessibility concepts. [164] partitioned target demographics into those with **vision**, **hearing**, **motor**, and **cognitive impairments**—based on WHO guidelines in [112] as well as [17]. The authors then proceed analyze the accessibility and the gaming experience through the lens of gameplay loop made of three phases—**receiving stimuli**, **determining response**, and **providing input**. They then consider challenges and solutions for improving this loop for each disability category. Speaking on the visually impaired, the authors conclude that even though these players are “physically able to perform” gameplay tasks, they are “unable to perceive primary stimuli” and that “without this feedback, it is impossible to determine what in-game responses and what physical input to provide [164].”

Echoing general consensus the authors in [164] later conclude that good accessibility features for the visually impaired should replace visuals with audio—

providing audio cues and sonification to inform players on otherwise absent primary stimuli.

2.3.2 Games and Audification

Despite this, the authors in [121] observe cases where visually impaired users claim mastery over games not designed specifically for their impairments—such as in *Mortal Kombat*. This is (in some part) due to the neurological theory of **brain plasticity**—a phenomenon wherein un-used parts of the brain (e.g., visual processing centers) are reused for other tasks [106], thereby affording visually impaired users additional ‘brain capacity’ for processing audio information. They explain that *Mortal Kombat* in particular is accessible (albeit unintentionally) to visually impaired users on account of its wide array of distinct and heavily signaled audio cues for different character moves and attacks. In general, repeated research has shown that improved auditory feedback ‘helps visually impaired individuals move faster in a game [2].’

Audio Games

This is reflected in the wealth of existing **audio games**, which either supplement or entirely replace the visual component of games with audio-oriented techniques. Or, to use the phrasing found on **audiogames.net**—the largest repository for audio games online:

“An audio game is a game that consists (only) of sound. Its gamemechanics [sic] are usually based on the possibilities [sic] of sound as well. Usually (but not always) audio games have only auditory (so no visual!) output. Audio games are NOT specifically games for the blind! It is true that most audio games around at the moment are developed by and for the blind community. But we think audio games have the potential [sic] to be a genre on its own due to the immense undiscovered possibilities of sound. We believe that audio games have the potential [sic] to be a complete gaming genre on its own.” [8]

The authors of [44] analyzed the trend of audio games, proposing a semiotic model of sound in games that considered gaming through the **casual**, **semantic**, and **reduced** listening modes. They also make inroads at merging theory with practice by demonstrating the **TiM (Tactile Interactive Multimedia)** project, a development of the Stockholm International Toy Research Center. The TiM project evaluated the efficacy and possibilities of audio games through three self-made titles: *Mudsplat*, *X-tune*, and *Tim’s Journey*. Through their work, they demonstrate ‘functional’ design solutions that prove even visually impaired children are capable of handling complex control interfaces. Moreover, the authors proceed to argue that ‘sound-based games’ do not have to be developed only for players with visual impairments.

The authors of [148] ventured to gain an ‘in-depth understanding of the audio game experience.’ They also surveyed tools for creating audio games—finding a dearth of information. They go as far as stating that they could find “(almost) no documentation or reflection of the processes that [other] researchers engaged in when designing their games”—highlighting a need for more audio game development tools.

Despite this, some have managed bespoke development of audio games. The first noteworthy example is Japanese game director Kenji Eno’s *Real Sound: Kaze No Regret*⁶, which is (to this author’s knowledge) the only audio game ever brought to mass-market. According to Eno, the game was inspired by letters he had received from blind fans following the success of his 1995 horror game *D*—which featured a slow paced, atmospheric, and puzzle-oriented design attracted blind players by chance [145]. The game was a modest success, being re-released on the Dreamcast in 1999.

While *Real Sound* may be the only mass-market audio game, there have also been many other notable entireties in the medium. The enduring popularity and open-source nature of id software’s *Doom* and *Quake* series have naturally extended themselves to the audio game community as well. *Shades of Doom* [47] and *AudioQuake* [6] are two such examples, which modify each game’s source code to include audification features and other accessibility support. AudioQuake has since expanded into the **AGRIP** project, including support for custom level designs via its ‘level description language [7].’

Others have sought ground-up re-creations of iconic titles—such as *Mach 1 Car Racing*—a remake of Namco’s arcade classic *Pole Position* [44]. There has also been a successful niche of audio rhythm games—such as *Blind Hero* [163], *Finger Dance* [99], and even *Metris* [57], a musical version of *Tetris*. The recent indie game *Blind Drive* [115], was developed from the ground up as a commercial release and brought audio games to entirely new audience outside of research circles.

2.3.3 Games and Computer Vision

Recent years have also seen a growing interest in the intersection of games and computer vision.

One common application has actually been the usage of computer games to train computer vision models—as games are able to produce easily understandable and realistic data on-the-fly [135] [129]. Others have considered ways to integrate computer vision into development processes—[123] merged openCV with the Unreal engine while [113] proposed methods of automating game testing through CV methods.

There has been some overlap between computer vision, games, and accessibility (and serious games) research. The authors of [21] proposed an educational game making use of computer vision, while [3] demonstrated an online game used in training a computer vision model that improves website accessibility.

⁶Originally released exclusively in Japan for the Sega Saturn in 1997. The game has since been re-released in audio drama format.

2.3.4 Related Work

Lastly, we note a few research efforts bearing resemblance to our own work. [37] performed similar research by using computer vision techniques and a bespoke algorithm to achieve computer-automated gaming. [149] also produced a similar result to our own proposal—integrating audification tools (sans computer vision techniques) into a custom point-and-click adventure game engine. The very recent work of [89], meanwhile, examines the implementation of audification techniques in the newly released hit fighting game *Street Fighter 6*, which received new accessibility features around the time of writing.

2.3.5 Conclusion

In this chapter, we reviewed theories and research relevant to our proposal. We began with an overview of computer vision research—namely in the areas of image processing and analysis—before moving onwards to AI and machine learning research. There, presented a brief summary of machine learning topics and considered an example using video games. We also made a brief stop to review the fields of object recognition and classification.

Afterwards, we switched disciplines and reviewed spatial audio theory—starting with concepts of sound localization before transitioning to discussions on spatial audio and 3D audio effects.

Finally, we turned our focus to games—beginning with a look at games accessibility research. We then mediated on audification research as it related to games—particularly through the lens of audio games. Then, we took a look at the intersection between computer vision and games research before finally making brief considerations of work similar to our ReAL Sound proposal.

Although this chapter forms an adequate summary of relevant topics, we will continue to introduce and review theory as needed in the following chapters as well. We now move to a detailed proposal and exploration of the ReAL Sound system.

Chapter 3

ReAL Sound

Introduction In this chapter, I propose and outline the *ReAL Sound* system: a **Re**-usable **A**udification **L**ibrary that abstracts the design of accessibility features for the visually impaired via the use of computer vision, spatial audio, and audification techniques.

3.1 Proposal

Introduction In this section, I propose ReAL Sound and justify its novelty and utility based on the literature review performed in the previous chapter.

What is ReAL Sound?

ReAL Sound (**Re**-usable **A**udification **L**ibrary) is a software framework concept that generalizes the creation of visual accessibility features for games. Using computer vision techniques and modern spatial audio technology, ReAL Sound aims to abstract the process of moment-to-moment analysis of a game’s state as well as the generation of 3D spatial audio objects. Through this abstraction, implementors of ReAL Sound (as distinguished from *users* of ReAL Sound—the visually impaired persons benefitting from the features) can sidestep many of the language, platform, and architecture specific headaches involved in the creation of game accessibility features, especially in the case of after-market games.

In essence, ReAL Sound seeks to convert a game’s visual information (the things a player sees on-screen) into audio information (the things a player hears). This is done so that visually impaired persons can better understand a game without having to ‘look’ at it directly. Moreover, ReAL Sound aims to abstract this process, making a simple implementation possible for a plethora of games. ReAL Sound achieves this generalization by abstracting feature development process into a few core steps for any given target game:

Planning The creation of simple ‘rules’ and ‘definitions’ which accurately describe the game.

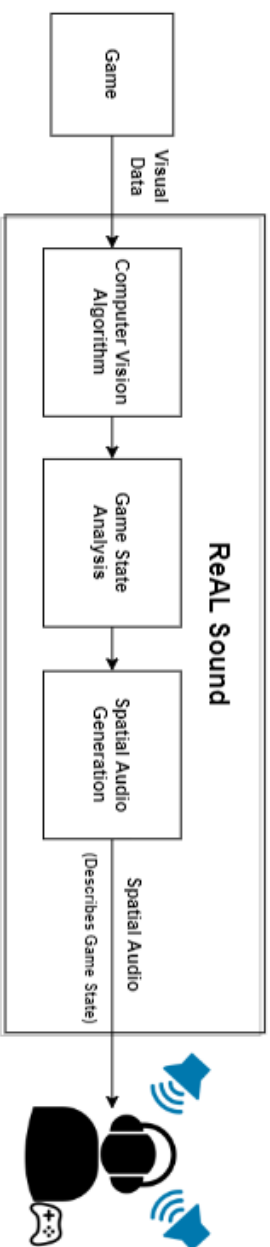


Figure 3.1: A simple graphical overview of ReAL Sound

Training The act of preparing computer vision techniques based on the **Planning** stage to analyze the game’s current state data.

Design The creation of simple ‘rules’ and ‘definitions’ for translating the data analyzed in the **Training** phase into spatial audio objects.

Execution The real-time marriage of the previous stages. Uses a computer vision model **trained** on the **planned** rules which translates the game’s real-time visual data into spatial audio objects as **designed** by the implementor.

With ReAL Sound, a implementor with only a moderate amount of technical knowledge could feasibly add accessibility features to any target game—all with zero knowledge of the game’s underlying source code or architecture. A visually impaired user of ReAL Sound, meanwhile, gains access to accessibility features which translate the game’s visual information into easily parsable audio data—allowing them to intuitively understand a game without seeing the game’s visual output.

Why is ReAL Sound?

As discussed in the previous chapter, there is a well-established and ever-growing need for software interoperability and platform agnostic support, as well as a growing demand for improved software accessibility. Implementors of game accessibility features already face numerous game-specific challenges in the design process—as each game’s bespoke nature demands equally unique accessibility design. The added barriers of specific architecture, engine, and language implementation have especially stymied the development of accessibility features in the gaming industry—where many development tools target a single architecture or operating system, are company specific, and are often used on a per-game basis.

ReAL Sound significantly streamlines these problems through abstraction—significantly simplifying this troubled implementation process. As a consequence, greater flexibility and more time are handed to feature designers—who are now better quipped to craft higher quality features at a faster clip.

Moreover, the generalization process requires zero knowledge of the game’s actual codebase, meaning games can be modified in the after-market by dedicated enthusiasts. As a consequence, the fan-driven movement of retrofitting of older titles with modern accessibility features becomes easier—allowing visually impaired persons access to a wealth of classic titles while costing modern developers zero resources. This is becoming especially crucial in modern times, as some estimates show that over 85% of games are functionally abandoned—with no publisher or developer entity maintaining ownership [74]. This fact effectively renders the vast majority of published games, ‘*abandonware*’ with little hope of official improvements by original developers [27].

How is ReAL Sound?

ReAL Sound is made possible through modern machine learning, computer vision, and spatial audio technologies. Using the latest computer vision techniques through libraries like OpenCV [12], even novice developers can implement object detection algorithms—which analyze visual input and return semantic information about the its contents—with ease.

The Modern AI Boom On top of this, the recent ‘boom’ in AI technologies [66] [94] has brought an equally intense focus to the development, improvement, and democratization of AI tools and systems [14] [58]. Ecosystems like *Hugging-Face* [159] and projects like Google’s *TensorFlow* [88] have dramatically changed the landscape of AI technology development—trivializing many tasks considered inaccessible to the common developer just a few years ago.

Agnostic Vision Requirements Considering the ever-changing state of modern AI and computer vision technology, ReAL Sound does not specifically call for any one particular solution for its **Training** or **Execution** stages. Instead, ReAL Sound only requires the *successful* real-time identification of in-game objects *as defined by the implementor*. The means by which this goal is accomplished is left up to the implementor, and may be achieved by any available means.

Consequently, ReAL Sound does not technically demand the usage of AI *at all*. Many well-proven pre-AI techniques—such as *template matching* [13] and *corner detection* [87]—have proven to be successful in many simple use-cases. I will demonstrate this in a later chapter, creating one implementation of ReAL Sound using the original corner detection algorithm—the *Harris corner detector* [55], originally developed in 1986.

Spatial Audio Running alongside the AI boom has been a comparably smaller (yet not insignificant) boom of 3D and spatial audio technologies. As discussed in the previous chapter, many advances have been made in sonification, audification, and spatial audio techniques in recent years. This trend has given end-users an abundance of choice for low-cost, high quality audio playback devices with native spatial audio support—from in-ear monitors like the the Apple *AirPods* [62] to even computer display monitors from major manufacturers like Dell [15]. Future patents promise even greater advancements through advanced techniques like automatic HRTF adjustments specific to the user’s unique physiology [150].

Many new software development libraries been developed to address this boom in consumer demand. Fan driven efforts like the *Spatial_Audio_Framework* [92] as well as company-produced projects such as Valve Software’s Steam Audio [139] have seen popular adoption. Legacy libraries with widespread adoption, such as the quarter-century old *Qt* application development framework [11], have also introduced support for modern spatial audio technologies in recent versions [23]. The implementations of ReAL Sound presented later in later sections

utilize Qt as a backend framework—although numerous other libraries feature equivalent functionality.

Conclusion In this section, I proposed ReAL Sound and provided justifications for its utility and novelty by considering contemporary technology innovations and emergent user trends.

3.2 Concepts

Introduction In this section, I explain in detail the theoretical concepts which underpin ReAL Sound. I later use these concepts to rigidly define the system’s core structure. To begin, I use formal math notation to construct a semi-formal definition of a ‘video game.’ I then explore this definition through the lens of automata theory. Following this, I use the construct to abstract methods of interpreting a game via visual analysis.

3.2.1 Anatomy of a Game

Introduction

Here, I construct a semi-rigid abstraction of video games using formal notation. Later, I will use these constructs to more clearly explore related concepts.

Game

A Game, termed M (for *meta*), can be conceived as the combination of three sets: a collection of meta attributes A , a series of in-game states G , and a group of entities I :

$$M = \{A, G, I\}$$

Meta Attributes The game’s meta-attributes M_A can be imagined as data that is preserved over an entire game session. This data can persist between state transitions and usually describes overarching information such as current playtime or the currently active game level.

Game State

Each state, G_S (referred to interchangeably as *game state* hereafter) is comprised of four components—internal state-attributes A , internal state-logic L , as well as conditions C for inter-state transitions T .

$$G_S = \{A, L, C, T\}$$

State Attributes A state’s attributes A act similarly to the game’s overarching meta attributes M_A , but on a per-state context. These attributes may store information such as the current time spent within the state or other data describing the state in specific. State data is lost upon transitioning to a different state.

Internal Logic The state’s internal logic L defines all the behaviors and activities that are carried out *within* the state. For example, a racing game may contain a **start** state—which contains logic for playing a special sound effect when the race is started, or routines to visually display the text ‘*START!*’ on-screen.

In abstract, L can be imagined as a series of conditional requirements (‘rules’) L_C that yield specific game actions (‘responses’) L_A :

$$L : L_C \longrightarrow L_A$$

Conditions A state also contains conditional rules defining when to exit the state and transition to different state. C defines these rules as a set of boolean statements—which evaluate to either **true** or **false**. Take, for example, a sports game which transitions from a **match** state to a **finish** state after ten points are scored in the match. Imagine that the current match score is stored as an integer value through the state attribute S :

$$S : [0, 10], S \in G_A$$

Then there is a condition within C , lets call it C_S , which might look like this:

$$C_S = S \geq 10$$

When C_S evaluates to true (i.e., when the match score has reached ten points), the state will transition transition to the **finish** state.

Transitions The transition function T defines the mapping of “where” to or “how” to transition to a different state after a condition in C has evaluated to **true**. In essence, T maps a conditional statement C_S to the next game state G_N to transition to:

$$T : C_S \longrightarrow G_N$$

This function can be generalized by instead taking a specific state G_S and a generic condition C as input:

$$T : G_S \times C \longrightarrow G_N$$

Entities

Entities I are the objects which constitute a game’s internal structure—the ‘actors’ of the game. An entity can be anything contained within a game, such as the player, enemies, items, level construction, etc. Usually, an entity’s behavior can be described using verbs—in either an active (*Mario jumps on the platform*) or passive (*The platform was jumped on by Mario*) sense.

Each entity has attributes A as well a collection of states E :

$$I = \{A, E\}$$

Entity Attributes Similar to the previous definitions, entity attributes describe entity-specific attributes. For example, an entity’s location point P in a 2D game’s world space might be modeled as:

$$P = \{(x, y) : x, y \in \mathbf{R}^2\}$$

Similar information, like a enemy’s current health, an item’s purchase price, or a bomb’s damage radius are also considered Entity attributes.

Entity States An entity state, then, describes an entity’s different states of being. A player entity might be able to **jump**, while a spell entity might be **cast** or an enemy entity might be **killed**, to name a few examples.

Some examples may seem unintuitive, but still work within this framework. A game’s menu system, for instance can also be considered an entity. In this case, the act of clicking on the menu—triggering some sort of effect, can be modeled as a **clicked** state with its own internal logic and goals.

As you might guess, a entity state looks similar to a game state, with its own internal logic L , a series of transition conditionals C , and transition mappings T :

$$E = \{L, C, T\}$$

One example of an entity state is **jumping**, which may transition from **idle** or **walking** or **running**. When transitioned to, the **jumping** state might, according to L , play a jumping sound effect. When a condition in C is satisfied (the jumping entity finally touches the ground again, or perhaps falls into a pit and dies), then the transition function T moves to the next state E_N :

$$T(C) \longrightarrow E_N$$

Summary

To summarize, a game M can be conceptualized as:

1. A collection of attributions (A), which describe aspects of the game’s overarching meta-state.

2. A set of game states (G), which each contain internal game-logic (L), state-specific attributes (A), as well as rules for when to transition to another state (C) and exactly which state to transition to (T).
3. A set of entities (I) which serve as the game’s passive and active subjects, each having their own set of attributes (A) and states (E). Each state compasses the same qualities of a state enumerated above (A , L , C , and T).

$$M = \{A, G, I\}$$

$$G_S = \{A, L, C, T\}$$

$$I = \{A, E\}$$

$$E = \{L, C, T\}$$

Limitations

While this construction is useful for our purposes here, I do not contend this definition as a end-all definition of games. It is already known that several games—both in physical and video formats—are Turing complete, which exists outside the definitions given here. Previous researchers however have likewise made attempts in formally defining games with similar bounds on their definitions—leaving titles like *Magic: The Gathering* [20] or *Minecraft* [87] out of scope [31]. Virtually any game can also be made Turing complete if methods of arbitrary code execution (ACE) are discovered, which has been demonstrated using *Super Metroid* [91], *The Legend of Zelda: Ocarina of Time* [39], and numerous others. Despite this, we find that our definition of games covers a sufficiently useful scope to demonstrate ReAL Sound’s utility.

Conclusion

Here, I have provided a general outline of games in formal notation. I do not guarantee nor contend that these structures form a complete closure over the entire concept of video games—whose definition is still fiercely debated to this day [67]—but these structures will serve as a useful framework for our following definitions and concepts.

3.2.2 State Models

Introduction

In this section, I briefly review automata theory theory. Through concepts like finite state automata, I produce the Game and Entity (GSM and ESM) state machine constructs, which I will continue to utilize in following sections.

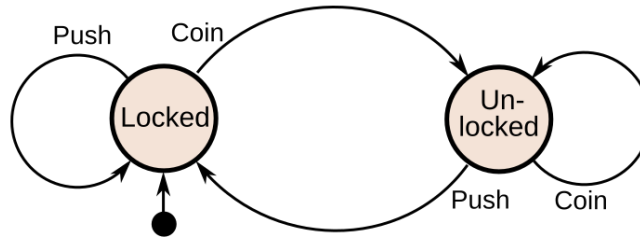


Figure 3.2: An example Finite State Machine (FSM or FSA). This FSA models a turnstile, which takes a coin as input while **locked**. The coin **unlocks** the turnstile. A user then pushes through the turnstile, which once again **locks** the turnstile [18].

Automata Theory

The previous section's underlying theory will likely seem familiar for those with a background in fields such as theory of computation or advanced linguistics. I am, of course, referring to the theory of *automata* a core concept which underpins numerous fields—including computing and, by natural extension, interactive software such as video games.

Automata are generally categorized into four major types as defined by the Chomsky Hierarchy—ranging from Type-0, describing the classical Turing machine, to Type-3, which describes finite state automata [19]. Games, as I have described them above, can (in general) be described as a Type-3 automata—an automation of finite state and quality.

Finite State Automata

Introduction and Concepts Carroll defines (deterministic) finite automata as the "mathematical model of a machine that accepts a particular set of words over some alphabet Σ . [16]" He conceptualizes FSMs as "black boxes" that combine an input tape (consisting of symbols in the alpha Σ), a 'read head' which processes this input tape of symbols, and an 'acceptance light' which indicates the acceptance/rejection of an input symbol, and whose activity is governed by the read head's reaction to the given input. In essence, the machine accepts input, makes decisions for the acceptance light based upon the input, and then moves to a new position to receive another input. The machine's acceptance/rejection of a symbol, as well as the position it moves to next, is contingent on its own internal logic—some rule that dictates "*If symbol X is read, turn on (if accepted) or off (if rejected) the acceptance light, and move read head to position Y.*" The machine may also read a symbol which finally halts its operation.

Comparison with Turing Machines This conceptual structure bares some resemblance to a Turing machine [146], although there are some distinct differences. A Turing machine has an infinite amount of memory and is capable of accepting languages with recursive qualities—as the Turing machine is able to ‘write’ to tape—modifying a symbol after reading it. A Turing machine is well known to be capable of implementing and computing any possible computing algorithm [147]. It is consequently classified as a Type-0 (also known as a “recursively enumerable” or an “unrestricted”) grammar.

An FSM, by contrast, is limited in several ways. As the name implies, FSM have a *finite* set of states. More importantly, the system is incapable of storing memory—lacking the ‘writing’ mechanism of a Turing machine [16].

Deterministic and Non-deterministic FSAs The FSMs we detail here should also be distinguished as *deterministic*, as opposed to *non-deterministic* FSMs. Some FSMs are considered non-deterministic when their states are capable of transitioning to several possible outputs given the *same input*—making their reaction to an input non-deterministic in nature [124]. For our purposes, we define games as a type of deterministic FSM—where the same inputs always yield the same results.

Obstacles and Limitations You may be wondering how the provided definition of games fit the category of FSM considering the inclusion of attributes A —which may persist as a game or entity transitions from state-to-state. This may appear like a kind of ‘memory’ on first glance, but is actually just a shorthand notation that simplifies our FSM conceptualization. In reality the proper FSM of a game has *many* more states than we describe here.

To provide some insight into this concept, imagine that for each state G_S , the conditions of C and the transitional rules of T also encode the attributes of A . This means that each state G_S actually has numerous variations—each relating to the specific attributes of A and the conditions of C . For example, the aforementioned sport game’s **match** state may really have 10 different **match** states, one describing each possible value of the scoring attribute S . If the game has just begun, the first scoring condition C_0 will transition the machine to $match_1$, and so on. Eventually, the final **match** state’s C_9 condition will transition the FSM to the **finish** state.

Useful Adaptions As you can see, this more-accurate representation of FSMs is considered unproductive in many domains where FSMs remain useful concepts [132]. Thankfully, numerous alternative notations have arisen to address these shortcomings.

For example, FSMs which produce output contingent on a given input—often termed *transducers* [16]—are frequently conceptualized as *Mealy machines*. The output of a Mealy machine state is also contingent on its input (IF **input**(X) AND IN **state**(Y) THEN **transition**(Z)). Later designs—such as the *unified*

modeling language—have extended this notation further to leverage the theoretical benefits of FSAs while avoiding their limiting notation schemes [108]. In a similar fashion, I make use of FSAs on a conceptual basis while abstracting away these notational hindrances through the usage of the attributes A of each object.

Automata and Games

It is possible to define games more clearly through their relation to FSAs using our definitions. For example we can consider a game to be a collection of two principal state machines:

Game State Machine (GSM) The game state machine comprises the general 'flow' of a game—its levels, combat encounters, win conditions, story sequences, game-over menus, boss fights, item inventories, and any other 'macro' loop that is reasonably distinguishable within the gaming experience.

Entity State Machine (ESM) An entity state machine comprises the general 'life' and 'activities' of a given entity—every action it can take, as well as its response to any given input. A player may run, jump, or die, while a stage platform may move or vanish.

These two concepts—the GSM and ESM—prove to be crucial in the conceptualization of ReAL Sound. We will continue to analyze them in later sections using other frames of analysis.

Conclusion

In this section I reviewed numerous aspects of automata theory—namely finite state automata—in order to present the concept of the Game and Entity state machines (GSM and ESM), which will prove to be useful constructs in the total construction of ReAL Sound.

3.2.3 How Players Understand Games

Introduction

In this section, I explore how games are conceived from the player's perspective. I illustrate my points via a few examples and ultimately generate the concept of the **State-Rule** relationship, which has consequences for ReAL Sound's design.

How Players Understand Games

Introduction Drifting slightly from the previous formal definitions, let us consider here exactly how a human player understands a game. To begin, let us consider a simple game familiar to most readers—the playground classic *rock-paper-scissors*.

Simple Case: *Rock-Paper-Scissors* In rock-paper-scissors, each player is allowed one input—the ‘hand’ they choose to play—and the game produces one ‘output’—the outcome of the match. Experienced players have an intuitive understanding of the rules which govern the game, which are simple enough to enumerate here:

Rule One: Each Player produces a hand symbol, which may be one of:

1. **Rock**
2. **Paper**
3. **Scissors**

Rule Two: The outcome of the match is decided by these stipulations:

1. **Paper** ‘beats’ **Rock**
2. **Rock** ‘beats’ **Scissors**
3. **Scissors** ‘beats’ **Paper**
4. If both players produce the same hand, the game is a draw

These rules comprise the entire experience, allowing a full play of the game to occur over the course of mere moments.

To use the constructs defined previously, it is clear that rock-paper-scissors (hereafter *RPS*) has only one possible game state, a fact which highlights the game’s simplicity. In other words, players require no greater sense of *context* to play RPS—they only need to understand the base rules described above. Consequently, the game has no memory requirements, as each play is decided in one step and has no consequences on future or past matches.

Average Case: *Rock-Paper-Subsequent* The previous definition of RPS can seem trivial, leading to an unfulfilling experience for gamers. As you likely know, many performances of RPS actually append on additional rules to improve the play experience. For example, competitions often involve some ‘*best-of*’ requirement—stipulating that a player must win a certain amount of matches before being declared the ‘overall’ winner.

This example of RPS, which I will term *Rock-Paper-Subsequent*, has added additional complexity to the game’s structure. The logic of each round is now contingent on the last, requiring players use a sense of context and memory to successfully complete the game. For example, forgetful players competing in a ‘best-of-one-thousand’ variant of RPS may eventually lose-track of the game’s score. This mistake would render the entire competition invalid, as it makes the question of who will win the match undecidable.

In automata terms, *Subsequent* has gained a sense of state—one for each possible configuration of game score. This case can be described in terms of an FSA, and, I conjecture, is categorically equivalent to my definition of games above.

Advanced Case: *Rock-Paper-Scheherazade* Imagine now that instead of having one simple match or even a simple 'best-of', we created a new version of RPS named *Rock-Paper-Scheherazade*. In this version of the game, a tied match (where players produce the same hand) requires that players perform a 'best-of-three' sub-match of the game. Whichever player wins this sub-match will 'win' the match. This logic also applies recursively, meaning that players who tie during a sub-match are forced to perform yet another sub-match *within* the sub-match—a sub-sub-match, if you will. This logic can naturally extend to an infinite recursion of context and memory: *sub-sub-sub-sub-sub...-matches* are possible in *Scheherazade*, each requiring the player to keep track of the game's current 'match', as well as all of the matches that exist 'above' it.

This rendition of RPS is, in some senses, comparable to a Turing machine—allowing for infinite memory and recursive processes—which extends beyond the definition of games we contend with in this paper. In this case, RPS may have an infinite amount of game states—each contextualized by the particular score of the current sub-match as well as all of the sub-matches that exist in the levels above it.

The purpose of these examples was to illustrate in clear terms the two key parts of a game's structure from the player's perspective—**Rules**, and **States**.

Rules Rules constitute the bulk of a player's understanding of a game. Without rules, even simple titles like RPS are rendered non-deterministic—as an infinite amount of hands outside of **Rock**, **Paper**, and **Scissors** may be produced, each with arbitrary rules about which hand 'beats' which, which may also change on a match-to-match basis. In general, players expect a specific reaction to a given input.

It should be noted that this is true even in cases where the output is seemingly random to the player. For example, a player always expects to be dealt a set of playing cards when performing an ante in Poker, even if those cards are given to the player in a seemingly random sequence. The player is not, by contrast, expecting to receive a collection of random objects that *differ* each ante in lieu of playing cards—a gun for one hand, a hot-dog for the next, and perhaps the entirety Shakespeare's *Hamlet* the last.

States Despite this, rules are not the end-all-be-all of understanding most games—which are usually comprised of many different states. In these cases, the rules of the game actually best described as the *rules of the state*. This means that the same rule, applied in different states—may actually produce a different output given the same input. This provides rules with a greater sense of context that games often demand.

The State-Rule Relationship Consequently, one may imagine the intertwining of these two concepts: **Rules** which define **State**, and **States** which identify **Rules**. This **State-Rule** relationship often relates to problems in *context sensitivity*, a domain which extends far beyond the realm of video games.

Understanding this relationship, as well as context sensitivity, is key to ReAL Sound’ design. We will now explore the problem further through a well known example: *The Legend of Zelda: Ocarina of Time*, which pioneered advanced solutions to context-sensitive game design.

Context Sensitivity and *Ocarina of Time* As three-dimensional games saw increasing feasibility in the early-to-mid 1990s, developers were suddenly given an extra degree of spatial freedom. Many designers quickly came to view this boon as a bane, as developers new to 3D struggled in translating these newfound liberties into sensible game controls—which are limited by both physical design and human physiology. In essence, designers often believed they needed twenty buttons to accomplish in 3D space what was once possible with only four buttons in 2D. This was a consequence of the exponential growth of **States** and the **Rules** which governed 3D experiences.

The Legend of Zelda: Ocarina of Time effectively solved this problem by standardizing a clear sense of context-sensitive design—both in terms of control design and UI design. In *Ocarina*, each button on a game controller may have more than one in-game action (a control) assigned to it. Exactly which action is triggered in game is dependant on the game’s current state. In this sense, the specific **Rule** of what happens when a button, say **A** is pressed is unique to the current **State** the game is in. For example, the **A** button may allow the player to ‘talk’ to another character—if they are standing near one. They may also be able to push a box, open a door, read a sign, or use a fishing rod—all from just pressing the **A** button, who’s **Rule** is contextually dependant on the current **State**. This was also made clear through the game’s novel UI design—where the contextual action of the button (the button’s **Rule** in the current **State**) was clearly displayed on-screen at all times.

This marriage context-sensitive design with clear visual indicators of the **State-Rule** relationship significantly aided players—who were still new to the concept of 3D games and their control schemes—and cemented *Ocarina*’s status as a model for modern game design. In essence, these choices clarified the relationship of **Rules** and **States** to players. I conjecture that an obfuscation of this relationship (on the part of the developer) or a lack of understanding of this relationship (on the part of the player) inhibits the completion—or, at the very least, *the enjoyment* of the play experience.

Conclusion

In this section, I elaborated on the experience of playing a game from the player’s perspective, making use of the structures defined in previous sections. Here, I illustrated the distinction between a game’s **States** and **Rules**, as well as the relationship between them using the example of *rock-paper-scissors*. I then considered the important of this relationship via the problem of context sensitivity, which I explored using the case study of how *The Legend of Zelda: Ocarina of Time* handles similar problems.

3.2.4 Visual Analysis of a Game

Introduction

In this section, I provide a framework for analyzing a game’s state information based on the its visual output using the constructs detailed in the previous sections. With this framework, I argue in the next section that a user who is aware of a game’s external logic (the ‘rules’ a player intuitively learns via playing the game) can wield this visual analysis in building their own useful state model of the game.

Context Sensitivity and ReAL Sound

As detailed in the previous section, we recognize the importance of context sensitivity in the design of ReAL Sound, which is tasked with understanding a game’s ‘rules’ and ‘contexts’ just as a real-world player is. Unlike a real human, however, ReAL Sound cannot leverage the semantic power of the human mind. This means that visual concepts familiar to the average human—the shape of Pac-Man, or what color Mario’s hat is—ultimately boils down to a collection of mere binary 1s and 0s for a computer.

Consequent to this fact is the first problem we must solve:

Problem 1. *How can ReAL Sound achieve the same level of context awareness that a human being reaches when visually analyzing a game?*

To answer this problem, we must sub-divide it into two: the analysis of a game’s **States** and **Rules**.

Analyzing Game State

Analyzing a game state is arguably the easier task, so we will begin our work here.

Differentiating State To analyze a state, a player must be aware of what makes it *distinct* from other states. Usually, this is done through some obvious visual cues. A game menu, such as a pause screen, is generally designed to look different than the in-game experience—clearly signaling to players that the game is not currently active. In a similar vein, the unique shades of blue and distinct green coral patterns of *Super Mario Bros*’ underwater levels indicate that a different play experience is to be expected.

And so, in order to understand a game’s active state, the player must:

1. Look at the game.
2. Detect the different objects that constitute the elements on-screen.
3. Categorize these objects into known entities.
4. Relate this assortment of entities to a known **State**.

Logically then, we must translate this process into one that is computer-friendly.

Looking at the Game Thankfully, the task of **1.** is simple enough. The current visual data of a game (hereafter referred to interchangeably as the game’s active *frame*) is easily copied from the buffers of memory which display it on-screen. We can then send this copy of the frame to whatever computer software we please.

Object Detection Unfortunately, it is at step **2.** that we hit our first major roadblock. Or, to be more accurate, *would* have hit a roadblock just a handful of years ago.

As discussed in section 2, the history of computer-automated object detection is as long and storied as the problem is complex and layered. Work in the field has generally focused on specific identification tasks with a well-known utility—such as the detection of human faces or cancer cells. Technologies and algorithms powering object detection were often bespoke and incredibly arcane, despite their limited use-cases. Only in recent years has the field matured into a respectable domain, owing mainly to modern AI and Machine Learning discoveries. Consequently, the identification of objects via visual input is now a fairly trivial task given some basic programming knowledge. Efficient cross-platform detection algorithms such as YOLO (*You Only Look Once*) have popularized object detection in a wealth of fields thanks to their relative ease-of-use [153].

Object Categorization: *The Journey* There is yet another problem, however. **3.** stipulates that we not only detect the *existence* of objects, we must also *categorize* these detected object into known entities. In other words, we must attach some sense of semantic meaning to these objects. In other words, the machine must not only recognize a collection of pixels as being identical from frame-to-frame, but must also recognize those pixels as ‘Mario.’

A purely automated solution to this problem is, unfortunately, outside the realm of this thesis (and likely modern science). There are certainly examples where an advanced, general computer vision algorithm may be able to recognize a popular entity like ‘Mario’—who it has likely seen thousands of times within its training data—but this approach is not scalable for the domain of *all* or even *most* video games.

In reality, these algorithms recognize popular gaming characters through many means external to games—advertisements, movies, t-shirts, etc. Even if games were a common aspect of training data, the bespoke nature of games means training off of one game will not guarantee success in another. To put it simply: being able to tell apart Mario, Luigi, and a *Goomba* will do no good in understanding the difference between Cloud, Tifa, and a *Tonberry* in *Final Fantasy VII*.

Object Categorization: *The Answer* Consequently, ReAL Sound requires human intervention to overcome this problem. Even though ReAL Sound would not likely function using a general machine learning model, it is very easy to obtain good results using a *specialized* model instead. Specialized detection models augment a model (pre-trained on general data) with new specific training examples provided by the implementor—200 images of the target game, for instance. Using this new data, the model can acquire an understanding of the target game after only a modicum of training—allowing for the reliable detection and categorization of in-game objects.

Of course, this process is not without its difficulties. For one thing, the act of providing image data to a machine learning algorithm is fairly laborious. Each image must be accurately annotated with labels describing the relevant objects contained within the image—requiring each picture to be manually edited and reviewed by a human being. On top of this, the exact number of annotated images required for good results is not consistent—with more complex games generally requiring larger datasets. Moreover, the *quality* of the provided data is also crucial. Images of a low fidelity, as well as poorly annotated data, will often yield lackluster model behaviors.

Despite all of these issues, the act of annotating training data has become commonplace in modern machine learning. There exist a wealth of tools—such as the open source *Make Sense* project [137] or the corporate projects of companies such as *RoboFlow* [130]—which streamline this process significantly. Moreover, the act of image acquisition is often trivial, merely demanding an implementor record a few minutes of active gameplay. The implementor could then simply pull a still frame from every few seconds of the video to use as data for annotation.

Given a few hours and some basic manual labor, a machine learning algorithm can be augmented with enough game-specific knowledge to enable the high-quality categorization of objects via computer vision—solving the hurdles of **3**.

Despite this, it should also be noted that machine learning, computer vision, and AI are not at all required to accomplish this task. A programmer willing to perform these operations manually—to design their own program which attaches semantic value to objects—can achieve the same results. ReAL Sound is ultimately indifferent to the exact approach to this problem—it only asks for good semantic information regardless of its origin.

Relating to State Now that we have all of this information—the game frame, the objects within the frame, and the semantic meaning of those objects—we must now perform the final and most crucial step: building a distinct mapping of this information set to one **State**. This process also requires human intervention, as it yet again demands the input data with additional semantic meaning.

This is the first area where ReAL Sound sees use—providing the format and framework for building these semantic relationships. This process—known as the *Design Phase*—is detailed in a later section. For now, it is sufficient to say the implementor provides ReAL Sound with information about the game’s

semantics.

Summary With these challenges overcome, ReAL Sound is now able to 'see' and 'understand' the state of the game much like a user can—translating visual information (the game's current frame) into meaningful semantic information (the game's current **State**).

This puts us one step closer, but we are still missing something: an understanding of the **Rules** which underline the active **State**.

Analyzing Rules

Compared to the lengthy process required for **State** analysis, **Rules** are deduced in a far simpler fashion. One may observe that the structures and consequences of **Rules** are often non-visual and potentially abstract in nature, meaning our computer vision approach is yet again trumped by semantic and context sensitive information. But, as detailed in previous sections, even a game's abstract **Rules** are generally intuited by the player. For example, a player might understand that a switch flipped in one room of a game's level will have some effect on an entity in a different, yet unseen area.

And so, we require human intervention yet again through the **Design** phase of ReAL Sound's implementation process. The implementor must provide some simple input which describes the given **Rules** of a game's state in formal logic—which serves as the 'language' the computer is capable of understanding.

Conclusion

In this section, I dissected the visual experience of analyzing a game from the player's perspective. From this, I built a set of requirements that ReAL Sound must satisfy in order to successfully 'understand' the game like a real human player would based on the same visual information. I then elaborated the challenge presented by each requirement, as well as how ReAL Sound overcomes these challenges in order to successfully intuit a game based using the same visual input a human player uses.

3.3 Structural Outline

3.3.1 Overview

Introduction

In this section, I provide an overview of the three 'layers'—**Vision**, and **Decision**, **Audification**—which constitute ReAL Sound's structure.

Turning Theory into Practice

Now that we have explored the concepts needed for understanding ReAL Sound, it has come time to finally explain the structure of the framework in detail. It

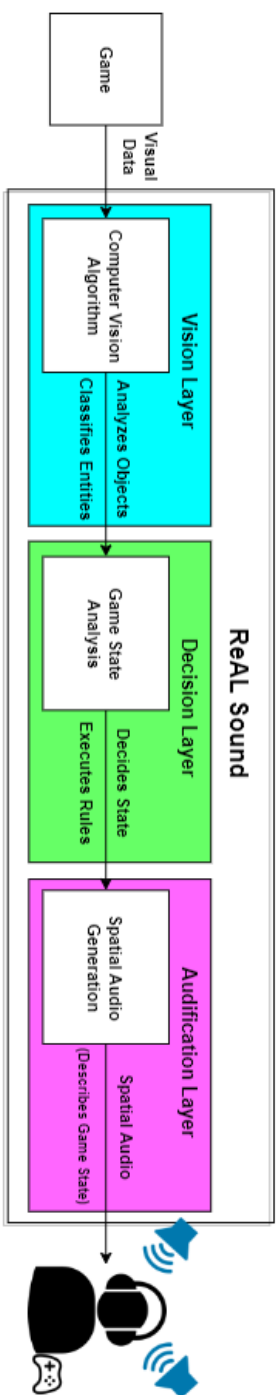


Figure 3.3: An extension of fig. 3.1 including **Layers**

should be clear by now that ReAL Sound reaches into several areas of study for its theoretical basis—performing tasks in numerous domains to achieve our goals. This multi-modal approach makes explaining ReAL Sound’s internal workings difficult. In order to ease this process, I will subdivide ReAL Sound into three key ‘layers.’ Each layer is tasked with solving problems in different domains and consequently has its own distinct traits and behaviors. The layers are:

The Vision Layer Converts a game’s visual information in formal game logic.

The Decision Layer Converts formal game logic into ReAL Sound’s own internal logic.

The Audification Layer Converts ReAL Sound’s internal logic into spatial audio, which is output for the end user.

It is hopefully clear how these three layers interact—and how the **Vision** layer cascades downwards towards the **Audification** layer, starting with visual data input and eventually generating audio data output. I will now elaborate on each layer in detail.

3.3.2 Vision Layer

The **Vision** layer can be imagined as the starting point of ReAL Sound’s execution loop. In it, the game’s frame data (its visual information) is provided to a computer vision algorithm. The frame can be provided in a host of ways.

A programmer with direct access to the game’s engine may choose to copy the framebuffer directly from its original in-engine source to ReAL Sound. Those without access to the game’s internal logic may instead opt for a simple screen capture implementation, which essentially sends screenshots of the game window to the computer vision algorithm. This may seem inefficient at first glance, but all major operating systems have support for screen capture built-in at the kernel level, which makes this option a very feasible and attractive route for users with relatively modern computer hardware¹.

This algorithm is responsible for the detection and classification of game entities contained within the frame. The algorithm may achieve object detection by any means desired by the implementor (machine learning, corner detection, template matching, etc.).

This same freedom is not extended to the act of object classification, however. The implementor must provide the **Vision** layer with object semantics. These semantics associate a detected object with a specific entity type—a mushroom in *Mario*, a slime in *Dragon Quest*, or a potion in *Zelda*, for example. Semantic associations can be included in any way the implementor pleases—training data for a machine learning model, custom programming logic, etc.

¹See `PrintWindow` in the Windows API [26] and `XGetImage` in the X11 API [43] for modern examples.

The **Vision** layer is, through this processes able to translate a visual image of the game into semantic information about the game’s active **State**—abstracting visuals into game logic.

To summarize:

Vision Layer	
Receives input from:	The game’s graphical frame buffer.
Input:	An image of the game’s current on screen graphics (<i>frame data</i>)
Behavior:	Uses a computer vision algorithm to perform object detection. Then uses semantic information provided by the implementor to classify these objects into game entities.
Output:	Semantic information about the game’s current state—as defined by the entities currently visible on screen.
Sends output to:	The Decision layer.

3.3.3 Decision Layer

The **Decision** layer is arguably the most complex layer, at least upon initial inspection. After receiving semantics about the game’s current frame from the **Vision** layer, this layer is tasked with translating these semantics into the ‘course of action’ ReAL Sound takes in response to this frame. In other words, this layer is where ReAL Sound’s *decision making* occurs. In essence, this layer can be considered the ‘heart’ or ‘core’ of ReAL Sound—as it takes place entirely within its own confines.

Of course, a program can only make decisions imbued in it by its creator—and ReAL Sound is no different. In reality, the implementor supplies ReAL Sound with a collection of simple conditional statements that translate into the range of possible decisions ReAL Sound is capable of making. For example, a supplied conditional statement may look something like this:

```

Data: Game semantics S from the Vision layer
Result: Procedural calls to generate Audification objects
if Entity 'Mario' is in S and Entity 'Mushroom' is in S then
    | if distance(Mario, Mushroom) ≤ dMushroomCollision then
    | | play mushroom power-up SFX;
    | end
end

```

Algorithm 1: A simple semantic **Decision**

In this simple case, the semantic data was used to check on existence of two entities—defined semantically as 'Mario' and 'Mushroom.' If these two entities are found to be on-screen at the same time, another check is made to see if they are close to each other—within a given distance d , as defined by the implementor. If so, ReAL Sound is told to play a specific spatial audio cue—a 'power-up' sound effect, which indicates what is already clear to a sighed player: that Mario has collided with a mushroom, giving him a power-up effect.

Of course, this is a trivial example—as most games already provide auditory feedback for simple actions like powering up. But the extensible functionality of ReAL Sound allows for the translation of many non-auditory features into sound. For example, the position of a specific entity could be mapped from an physical point in game-space to a spatial point in the user's audio-space—generating sounds around the user's head to indicate their position in a clear, intuitive fashion.

Utilities and Extensions

The totality of these features allow ReAL Sound—under the imperative command of the implementor—to make active **Decisions** about what sound objects to generate in a given situation. More that, **Decisions** also allow the implementor freedom in controlling the 'How?' 'When?' and 'Where?' of **Audification**. More generally, the implementor is able to perform a host of non-audio related tasks using this functionality as well—updating state, entity, and meta attributes, generating other forms of user-understandable feedback, etc.

To achieve this, several simple constructs are provided by ReAL Sound to streamline the implementation of **Decisions**. For example, a $distance(x, y)$ function was used above to quickly calculate the distance between two entities. This can be applied to any given entity in ReAL Sound, as each entity's semantic is bundled with its screen-space cartesian location. Other foundational functions—such as a $amount(x)$ function (for calculating how many instances of an entity type are on-screen) and a $size(x)$ function (for quickly figuring the scale of an entity)—are also provided.

To summarize:

Decision Layer

Receives input from: Vision Layer

Input: The game's current state, as described in semantic terms defined by the implementor.

Input: Conditions and imperatives as defined by the implementor.

Behavior: Evaluates the given semantic data against the provided conditions. If a condition evaluates to **true**, the associated imperative is executed.

Output: Various commands. Primarily the command to generate spatial audio objects, with behaviors defined by associated imperatives.

Sends output to: The **Audification** layer.

3.3.4 Audification Layer

Finally, we arrive at the **Audification** layer, which is responsible for the generation of spatial audio objects that are played back to the end user—enabling them to ‘hear’ the information usually ‘seen’ in playing a game.

This layer can be considered the simplest of the three, as most of its duties are delegated to any number of third-party audio libraries which exist outside of ReAL Sound. As mentioned in previous chapters, spatial and 3D audio technology has matured to the point of trivializing the generation of dynamic spatial audio—like in our use-case here. Consequently, the actual specifics of this layer are left up to the implementor—provided they accurately generate the audio as stipulated by the **Decision** layer.

The comparatively trivial nature of this layer might leave one wondering exactly why it is defined as a distinct layer at all. This is because **Audification** serves a clear and distinct purpose within ReAL Sound (*it is the system’s name sake, after all!*). **Audification** also has its own set of unique behaviors and exists in a different domain (audio and sound) when compared to the previous two layers (vision and internal game semantics). Lastly, **Audification** produces our final and ultimate output—the sounds being played back to the end user. For these reasons, I have decided to distinguish it as its own pillar of ReAL Sound’s structure.

To summarize:

Audification Layer

Receives input from: **Decision Layer**

Input: A collection of spatial audio objects, as well as rules for how to instantiate them.

Behavior: Generates spatial audio based on the conditions provided as input.

Output: Spatial audio, which is outputted to the end user’s headphones via a framework supporting 3D and spatial audio technology.

Sends output to: Spatial/3D Audio playback libraries, the operating system, and eventually, the end user.

3.3.5 Summary

In this section, I provided a in-depth analysis of ReAL Sound’s structure. I subdivided the system into three core layers which have unique behaviors spread across different problem domains. These layers—**Vision**, **Decision**, and **Audification**—ultimately work as a pipeline: taking a game’s visual information, converting it into semantic data about the game’s state (**Vision**), converting those semantics into actionable imperatives (**Decision**), and finally converting some of those imperatives into spatial audio objects (**Audification**). Figure 3.4 summarizes the entire structure of ReAL Sound.

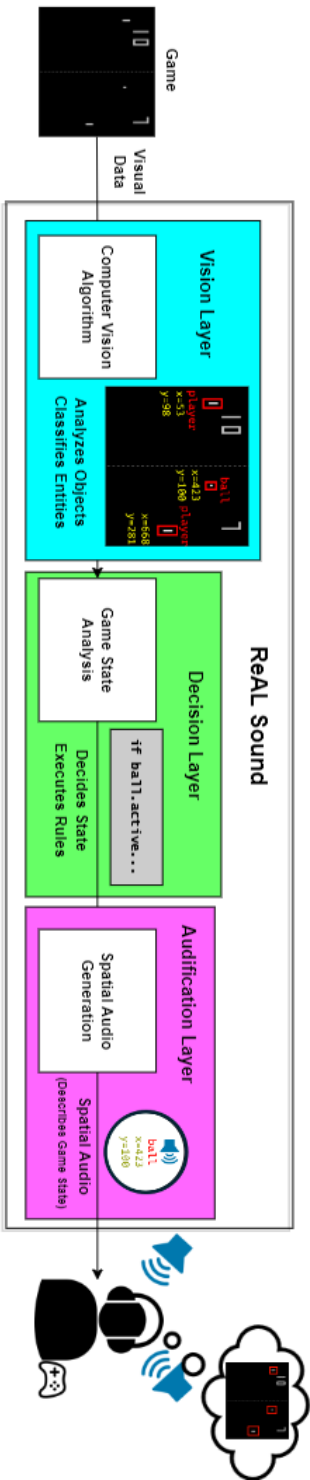


Figure 3.4: An full overview of ReAL Sound

3.4 Implementation Process

In this section, I outline ReAL Sound’s principal operational process—the **Planning, Training, Design, Execution** loop. To conclude, I consider the structure’s strengths and weaknesses.

3.4.1 Overview

In the previous section, I detailed ReAL Sound’s internal structure, and provided some hints as how an implementor interacts with it. But this structural overview paints a theoretical picture of software, not a full picture of how it can be used in the real world. In this section, I go into specifics on how an implementor applies ReAL Sound and an end user actually interacts with the software. To do so, I conceptualize the **Planning, Training, Decision, Execution** process loop—which serves as a useful guide to understanding how ReAL Sound may be implemented into a given target game.

3.4.2 Planning

And so, we begin with the first phase—**Planning**—which occurs before a implementor even touches a computer. First, the implementor must come to understand the target game’s internal logic and plan the semantic relationships which ReAL Sound will use for operation.

This phase may appear simple at first glance—as it requires no direct programming or technical applications—but is actually the most difficult phase. To borrow the old Sagan aphorism:

“If you wish to make an apple pie from scratch, you must first invent the universe.”

In order to make the ‘apple pie’ that is ReAL Sound’s implementation into a target game, the **Planning** phase must first ‘invent’ the universe which contains it. The quality of the implementation depends upon a successful plan—earning this step the title of ‘most critical.’ If the foundational plan is poor, then the following phases will no doubt yield poor results as well.

This phase involves several steps, which will we now investigate in detail.

Analyzing Game Semantics

First, the implementor must come to terms with their target game—the, **States**, **Rules**, and **States** that it consists of. For a simple target game—such as a classic arcade game or early home-console title—this process is fairly trivial. The implementor may break the game down into a series of entities, **States**, and **Rules** which apply to each state.

***Pac-Man*: A Worked Example** For example, one may observe *Pac-Man* consists of only a few key states—an **attract** screen that plays before the user starts the game, a clear **opening** before which plays before each life or level begins, the **normal** gameplay cycle, the special **power-up** cycle (where ghosts are vulnerable), a **win** phase when a player successfully completes a level, a **death** sequence which occurs whenever the player collides with a ghost and a **game-over** phase. One may also easily observe the few entities which comprise the game—*Pac-Man*, all four of the *Ghosts*, the standard *Pellet*, the *Power Pellet*, the several *Items* which occasionally appear on-screen, etc.

Planning Transitions Using these two groupings—the **States** and the Entities—one may finally attach **Rules** to each state, as well as the transitions between them. For example, the **PRESS START** text which appears during the **attract** phase could be classified as an entity. The presence of this entity on screen must mean the game is currently in the **attract** phase. The sudden appearance of other entities on screen—such as the *Pellets* or *Pac-Man* will indicate a player has started the game—taking us to the **opening** of the game. Observations like these are how an implementor builds the transitional function T for each **State**. It should be noted that these transitions serve as a special kind of **Rule** for each **State**—one dictating how and when to move between **States**.

Desinging Rules

State Rules Now that we are able to move between **States**, we must also define the **Rules** which comprise each one. For some **States** such as **attract** or **game-over**, this is a trivial operation—usually requiring only transitional **Rules**. For the advanced **States**—the ones which usually comprise the core gameplay loop—a more careful treatment is required. An implementor needs to intelligently organize in-game actions into useful logical constructs. Some **Rules** are simple—such as checking if the distance between *Pac-Man* and any of the *Ghosts* is less than some small value, which would indicate a need to transition to **death**. Other rules may have more nuance, such as understanding when the player achieves an extra-life after acquiring enough score points. One may notice that I did not detail any sort of **extra-life** state in the previous section. Nor have I detailed the **Rules** needed in ascertaining a transition from the **normal** game state to the eventual **death** state.

This is because we have only considered half of the **States** and **Rules** of the target game. We now consider the other half—those belonging to each Entity.

Entity Rules Like the game itself, each entity has its own collections of **States** and **Rules** which govern it. These constructs describe behaviors an entity has over its lifetime. To return to the example above, we can consider the behaviors (i.e. **States**) of the *Pac-Man* entity.

As most readers are no doubt aware, *Pac-man* is famous for his voracious appetite—**eating** pellets as he traverses the game world. He also enters a

power-up state, where he is able to destroy ghosts. As alluded to above, he also **dies** whenever he collides with a *Ghosts* while not **powered-up**. These three states—**eating**, **power-up**, **death**—cover *Pac-Man*’s obvious behavior, but there are less opaque **States** we must also consider. How can we reconcile this oversight in our plan?

Transient States For example, *Pac-Man* is able to gain additional lives if a specific score is achieved—a behavior not yet recognized by our **State** model. To account for this, we must also consider the notion of *transient States*, which generally persist for only a single frame of in-game time. To illustrate this point further, consider the case when *Pac-Man* touches an item object. The player is awarded a score boost, which has in-game consequences. Naturally, the player should be notified about this special occurrence. Of course in this specific case the game provides several kinds of feedback—removing the item object from screen, replacing it with a numeral score visual, and playing a special sound effect.

But these transient events are not always so clearly signaled. A player might receive audio feedback when *Pac-Man* consumes an item, but they will not hear any indication of when that item first spawns on screen—obscuring key **State** information that a sighted player has easy access to. And even though item consumption has audio feedback, it does not paint the full picture—obscuring the score value of the item, which is only signaled to the player visually.

Through these considerations, it becomes clear that transient **States** also play a key part in **Planning**. Thankfully, most of these problems are solved by the inclusion of audio cues indicating transient events to the player. For example, the implementor could plan special audio cues that are generated whenever a item first appears on-screen—creating a unique sound that differentiates each item type. On top of this, a unique pitch could be generated each time an item is consumed—the frequency of the pitch being related to the item’s score value.

State Delegation The existence of Entities and transient **States** presents one more interesting problem: the *delegation* of responsibilities between Entities. In the previous example, we considered the interactions between *Pac-Man* and the various items that occasionally appear on-screen. I generally implied that *Pac-Man* is responsible for maintaining the relevant **State** information about these interactions. But is that really the case?

A state machine (at least as we define them here) can only exist in one state at a time. This means that *Pac-Man* would need to leave his **normal** state in order to respond to special events such as when an item is spawned or consumed. This is generally unproductive to the overall experience. For example, if a important sound was being generated every frame by *Pac-Man*—perhaps a sound indicating his on-screen position—the sound would need to be temporarily paused in order to handle the transient **State** change. This may be a non-issue in most cases, but it is easy to imagine situations where even short lapses in the consistency of key information spells trouble.

And so, the implementor must also take great care in delegating the responsibilities and behaviors of each entity. In this case, the aforementioned tasks could instead be handled by the items themselves—leaving *Pac-Man* free to perform other **States** which are more relevant to the overall experience. An item could have states like **spawned** and **consumed**, which handle the situations described previously. On top of this, an item may also have a **normal** state—playing a spatial sound which describes its current location relative to *Pac-Man*.

Summary We have wandered down a bit of a semantic rabbit hole via our *Pac-Man* example, so let allow us to return to the big-picture. In the **Planning** phase, the implementor considers the characteristics of the target game and plans a 'model' of the game. The model comprises the **States**, **Rules**, and Entities of the game.

It should be stated yet again that this plan has not yet rendered a single line of code. The model instead serves as the *guidelines* which drive the following steps of the implementation process.

To summarize:

Planning Phase

Requires: A target game. Game analysis skills.

Involves: The planning of a model replicating the target game's characteristic behaviors. This model consists of

Game States A list of states that comprise the entire game experience

Game State Rules A collection of rules that characterize what occurs during each game state. This also includes rules for how to transition between game states.

Entities A collection of entities that comprise all of the actors within the game.

Entity States A collection of states that describe the actions each entity is capable of.

Entity State Rules The rules which characterize each Entity state's behaviors—how an entity dies, what happens when it is collided with, etc.

Purpose: Generates a plan which guides for following phases.

Notes: This model is purely 'on-paper.' No code or software is required for this phase.

3.4.3 Training

I remarked at the start of this section that we were finally going to put theory into practice—a promise we have yet to fulfill. Let us amend that now with the **Training** phase. In this phase, the implementor is tasked with choosing and implementing a real means of analyzing the game’s frame data. There are many different ways of accomplishing this task. We shall consider a few of primary methods now.

Computer Vision - Machine Learning

As we have no doubt made clear by now, virtually every useful method falls somewhere under the umbrella of computer vision. In recent years, this has meant the implementation of machine learning based object detection algorithms such as *YOLO*. In this case, the implementor is tasked with generating the relevant training data needed to tune their machine learning model to the target game. As detailed in the previous section, this generally involves the creation of **Training** data which is used to train the model (hopefully making clear the provenance of this phase’s name).

It is hopefully clear to us now why the **Design** phase precedes **Training**. As discussed in earlier sections the implementor must annotate images of the games state in order to create a training data set. Each photo must clearly and consistently label game entities that appear within the frame. Without a clear **Design**, the implementor risks creating inconsistent annotations and a mediocre dataset. This would likely yield subpar training results and consequently a lackluster computer vision algorithm—impeding future steps.

At any rate, we suppose that the implementor manages to successfully train a machine learning model. This model is capable of receiving a image of the game as input and outputting a list of entities which appear in the image—each one annotated with distinguishing information like the entity type and their position on screen. This suffices to satisfy the **Training** process.

Computer Vision - Alternative Methods

But machine learning is not the only answer to this problem. Depending on the target game, other methods may prove to be easier to implement and more efficient in handling the **Training** requirements. Some examples include the previously discussed techniques of template matching and corner detection. Each uses a simple algorithm that performs a comparatively naive analysis of the frame, but may still yield successful results at a far more efficient speed than machine learning alternatives.

There are still drawbacks to these alternatives, even when they prove to be successful. The appeal of machine learning is its ease of implementation into a wide variety of problem domains—a benefit not extended to alternative solutions. Instead, these methods demand bespoke software solutions that are often byzantine in nature.

An implementation of a corner detection algorithm (which we will demonstrate in the following section), for example, also demands custom semantics for translating detected corners into objects. This translation process is extremely specific to each game—and may involve painful pixel-level calculations that are often unreliable. Other methods like template matching allow the implementor some reprieve, but ultimately come off as an incredibly naive form of machine learning. However, there may still be cases where these simple approaches have utility, as we will see later.

Summary

In essence, the **Training** phase is where we first put theory into practice—choosing and implementing a computer vision algorithm capable of transforming a game image into a collection of well-defined entities which are processed in later stages. ReAL Sound aims to generalize this process by putting very few restrictions on the implementor, who may choose any current (or future, from the perspective of this writing) techniques they please.

To summarize:

Training Phase

Requires: A computer vision algorithm. Semantic data as created in the **Design** phase.

Involves: The implementation of a algorithm capable of translating a game's frame data into a list of entities which appear on-screen.

Purpose: Translates visual information into game semantics—serving as the **Vision** layer described in previous sections.

Notes: Machine learning models are generally used, but the implementor is free to choose any method they please, provided it generates the correct output for later phases.

3.4.4 Design

Now that the implementor has successfully transformed frame data into entity information—successfully setting up the **Vision** layer of ReAL Sound—they must now prepare the **Decision** layer. Similar to **Planning**, a **Design** is now needed. This design marries the semantic model from **Planning** with our data from **Training**. In other words—the implementor must now explicitly design the conditions and actions which ReAL Sound uses to make **Decisions** on how it generates audio objects. We will now explore the nuances of this phase in detail.

Understanding Game Design

First, we begin with a simple question:

How does one design a game?

Or, to be more pertinent to our aims:

*How does one design a **good** game?*

Of course, open questions like these have been debated since time immemorial, and we do not intend to throw ourselves into the fray of discourse here. But it is worth considering some history and context before moving to our next point.

Origin of Multisensory Game Design Video games—as we know them today—ultimately derive from mid-century electro-mechanical amusements (EMAs) such as *Skee-Ball*, shooting games, or the various ‘test-of-strength’ games that can still be seen at fairs to this day [105]. These EMAs often lined the walls of popular leisure spots—penny arcades, nickelodeons, amusement parks, etc [104]. In order to maximize their novelty status (and to grab attention in loud, crowded venues), most EMAs employed a wide range of sensory experiences to attract customers. Flashing lights, explosive sounds, and electric jolts of vibration were just some of the many ways that EMAs amused their customers.

As the introduction of the transistor and integrated circuit made computer games a reality, many EAM designers transitioned into this new field of amusement—taking their years of industry experience with them. Tomohiro Nishikado—the designer of *Space Invaders*—as well as Gunpei Yokoi—creator of the *Game & Watch* and *Game Boy*—serve as examples of this phenomenon [138].

Games as Multisensory Design Experiences This historical trend illuminates a key fact for this discussion: video games evolved into multisensory experiences the moment they exited the computer laboratories of universities and entered the arcades of the common man. From the dazzling visuals of arcade pioneers like *Dragon’s Lair*, to the avantgarde audio experiments of Kenji Eno², to even the unique control schemes that propelled the Nintendo Wii to worldwide success—game design has been about the multisensory marriage of audio, video, and tactility since its earliest days.

Unfortunately, this harmonious relationship is often a detriment to those with sensory impairments. The complex nature of game design often requires the encoding of key game data into only *one* of the principle senses—Color-coded items, pressure sensitive buttons, timing-based audio cues, etc. These instances of mono-sensory data encodings³ are often the ‘problem spots’ that render games inaccessible for impaired persons.

²Whose game *Real Sound: Kaze No Regret* serves as the inspiration and namesake of this research.

³Specifically referred to as mono-visual, mono-aural, and mono-tactile hereafter.

We highlight these points to make clear the sort of challenges the implementor faces in **Design**. Games are often designed with a visuals-first approach, which is supplemented and reinforced by audio/touch design second. This means many types of key game data are encoded through visuals alone—necessitating the need for a careful redesign. To be clear: **Design** does not mean redesigning the entire game from scratch, but to craft an *audio-centered redesign* of the game.

To achieve this without ReAL Sound, re-programming the entire game (or at least modifying some source code directly) is a rigid requirement. The entire point of ReAL Sound is to abstract this process away from real games programming. We will discuss exactly how this is done in a moment, but for now, let us consider the consequences of re-designing a game for ReAL Sound.

Audio Design as a First Class Citizen

At the heart of any good ReAL Sound **Design** is the belief that audio is a '*first class citizen*'⁴ of design. This may seem obvious considering the problems ReAL Sound seeks to solve, but the simple observation has a critical impact on crafting a good **Design**.

Player-Oriented Sound Design Let us illustrate a few examples to build our intuition.

A Worked Example: *Super Mario Brothers* Imagine that we are trying to implement ReAL Sound with *Super Mario Brothers* and are currently in the process of designing Mario's behaviors. We want to convey mario's on-screen position to the player, which is mono-visual in nature. A naive thought would be to play a repeating sound which is 'panned' to the left or right of the user's ears, relative to mario's position on-screen. If Mario were on the far left side of the screen, the sound would play only in the left channel, while a dead-center position would play the sound equally in both channels, etc.

This may seem like a good idea at first, but critical inspection reveals weaknesses. For one thing, *Super Mario Bros.* is a platforming game—where the camera constantly shifts from left to right as the player traverses the level. This means Mario will never occupy the right side of the screen—the camera is constantly shifting with him! In the incredibly cluttered and limit realm of audio-space, oversights like this produce subpar **Design** that not only impair ease-of-use, but also produce discomfort for the player—who is forced to hear Mario in their left ear for their entire playtime.

Player-Centered Audio Design The solution is to rethink our understanding of *Super Mario Brothers*—to see it from a new, audio-first perspective.

⁴The notion of first (as opposed to second) class citizens derives from programming language convention—where specific constructs (functions, objects, messages, etc.) are given center focus in the language's design. See [1] for more information.

The nature of 2D visuals (as well as the technical limitations of 1980s game hardware) demands a 'flat' or 'head-on' perspective—where the player sees the world of Mario as if they were looking at a picture book. This paints Mario as equally important as the entities around him.

This is no longer true in an audio-first world. Here, Mario stands as the *center* of his universe—with everything else revolving around him. This means a good **Design** of *Super Mario Bros.* would likely re-frame the entire conceptualization of Mario's world—from a 'head-on' 2D experience, to a 'first-person' 3D one.

Why? Because this allows us to highlight the world *around* Mario, instead of focusing on the character himself. *Mario* is a game about Mario—he will always be on screen. He is the universal constant that binds the rest of the game together, meaning his position on screen is hardly relevant to the player—who instead must focus on enemies, items, pitfalls, etc. These other entities are not relevant in themselves—but in their existence *relative* to Mario. For example, a *Mushroom* which slides past Mario—first appearing in front of him (to his right) before eventually sliding behind him (to his left)—might have spatial audio framed in the same way: starting in the player's right ear and slowly panning to the left.

This redesign process—shifting and translating the game's vision-first design into one that intelligently makes efficient use of aural space—is what it means to treat audio as first-class citizen of **Design**. Specific considerations will vary from game-to-game, but the core details remain the same: Translating visual semantics into useful audio ones—even if they significantly deviate from the game's original design paradigms.

Crafting Design

Equipped with a better understanding of what makes for good **Design**, we now consider its practical implementation.

In a low level sense, implementing **Design** involves programming rudimentary logic conditions—which are checked against the entity information provided by the previous **Training** phase—and actions—which instantiate spatial audio objects and handle other logic—in response to the conditions. This should bring the **Decision** layer to mind, as it is where our **Design** plays its largest role.

Simple Case: Custom Code To put things into programming terms, we could imagine the simplest **Design** implementation being a series of **if-else** statements which compare the provided entity data against relevant conditions. Pseudocode 3.1 demonstrates three simple examples using the **python** language. The first example echoes one used in describing the **Decision** layer—playing a sound effect when Mario collides with a mushroom.

The second example is more complex, generating a special 'warning' sound cue when Mario approaches an on-screen pitfall. To avoid cluttering audio-space, we have opted to only warn the player if they come dangerously close to a pitfall, a metric defined by the **WARN_DIST** attribute. We can also presume the function used to trigger the sound effect, `play_pitfall_warning_sfx()` is dynamically

```

1 if(mario is not None and mushroom is not None):
2     if(dist(mario, mushroom) < COLLISION_DIST):
3         play_powerup_sfx()

```

(a) On Powerup

```

1 if(mario is not None and pitfall is not None):
2     if(dist(mario, pitfall < WARN_DIST)):
3         play_pitfall_warning_sfx(dist(mario, pitfall))

```

(b) Approaching Pitfall

```

1 if(mario is not None and mario.state is STATE_DEATH):
2     play_lives_remaining_sfx(--mario.lives)

```

(c) On Death

Pseudocode 3.1: Three samples of rudimentary **Design** logic.

generating audio based on Mario’s current distance from the pit—as the function takes the distance between the two entities (`dist(mario, pitfall)`) as input. The function may use this distance value metric to vary the pitch, volume, or frequency of the sound effect—more clearly explaining when and where danger awaits the player.

The final sample makes use of the entity **States** discussed in previous sections. When mario’s death is detected (presumably his death animation was associated with a state named `STATE_DEATH` during the **Training** phase), a sound cue indicating Mario’s remaining life count is played for the user—as this information is mono-visual in the original game.

One may wonder the purpose of ReAL Sound if the implementor is forced into programming lines upon lines of conditional statements. Although this may be monotonous work, it is no doubt simpler than the act of programming the entire game itself. There is no need to concern oneself with physics logic, graphics processing, data management etc. Instead, the implementor uses rudimentary logic that even a novice programmer could easily understand—more resembling textbook pseudocode than production-level programming. More importantly this has merely been the most rudimentary example of **Design**. We can easily conceive of higher-level implementations which abstract much of the boilerplate programming seen here.

Advanced Case: Domain Specific Language An implementor could craft their own *domain-specific language* (DSL), for instance. As opposed to general programming languages (GPL), DSLs are purpose-built for specific problem

domains [95]. Notable examples range from Wolfram’s *Mathematica* language—which is used in advanced mathematics—to even HTML, which is merely a method of annotating hypertext. DSLs are often known as ‘*minilanguages*’ because they are frequently used by developers to streamline workflows and internal tool usage [126].

```
1 mario touches mushroom: play mushroom touch;
```

(a) On Powerup

```
1 mario close to pitfall: warn pitfall;
```

(b) Approaching Pitfall

```
1 mario dies: lower lives;
```

(c) On Death

Pseudocode 3.2: Three samples of a hypothetical DSL used to **Design** a *Super Mario Brothers* implementation. Entities are highlighted in green, DSL keywords are purple, and SFXs are rendered in red.

We can plainly see the utility of DSLs when combined with ReAL Sound’s **Design** process. An implementor interested in a wide variety of target games could build a generalized DSL, while a different implementor could just as easily build a specific DSL to streamline **Design** for a particularly large and complex game. Pseudocode 3.2 provides some examples of a hypothetical DSL specifically written for *Super Mario Brothers*. Much of the previous code has been reduced into far friendlier (both for reader and coder) format.

Pseudocode 3.3 showcases DSL with grammar accepting two command stings. Those unfamiliar with formal language theory or metacompilers may struggle with the syntax, but suffice it to say that this DSL streamlines the **Design**

```
1 ENTITY VERB ENTITY: VERB [ENTITY | ATTRIBUTE] [SFX];
```

(a) Grammar 1

```
1 ENTITY VERB: VERB (ENTITY | ATTRIBUTE);
```

(b) Grammar 2

Pseudocode 3.3: Sample DSL grammar

process significantly—replacing lengthy code segments with basic **noun verb noun** phrases that are as easy to write as they are to parse.

Advanced Case: GUI Of course, we could go one step further and create an entire graphical user application to aid the ReAL Sound implementation process. In this case, a DSL would likely be supplemented with simple graphical tools that visualized the **Design** process—allowing even a novice user to efficiently create **Design** logic. We will table this idea for now and discuss the concept further in the following chapter.

On Audification

We have made passing references to spatial audio in previous paragraphs, so we must now take a moment to discuss audification in more detail. Despite being described as one-third of ReAL Sound’s structure (**Audification**) and ultimately the final output of this entire process, the actual specifics of spatial audio generation are (for the most part) outside the scope of this thesis. This omission is not due to the subject’s complexity or nuance, but instead for the sake of brevity. In truth, support for spatial audio in contemporary libraries has become so ubiquitous as to render discussion of the topic irrelevant. Virtually any programmer, regardless of experience, can generate spatial audio using any off-the-shelf library and a few lines of code. Consequently, we have chosen to emphasize the problem of **Design** in this section instead. The task of physically generating audio may be left as an implementation afterthought.

Summary

In essence, the **Design** phase overlaps the **Decision** phase of ReAL Sound’s structure. The implementor designs and implements a collection of conditional statements and programmatic actions which are executed after a condition renders true. Conditions are based on the list of entities generated by the **Training** state, and actions generally create spatial audio, which ties into the **Audification** layer. There are many possible ways to handle **Design**, from rudimentary and bespoke code, to custom domain specific languages and potentially even entire applications with graphical user interfaces to streamline the **Design** process. To summarize:

Design Phase

Requires: A design for converting entity information into actionable, programmatic decisions. The implementation of this design via some software means.

Involves: Translating the visual-focused nature of video games into audio-oriented design. Conditional logic that accurately converts entity data into useful audio information. Software implementation.

Purpose: Translates entity information into game-state semantics via conditional logic. Converts those semantics into spatial audio data via defined actions.

Notes: Software implementation particulars are left up to the implementor. Anything from rudimentary programming to advanced GUI applications are acceptable.

3.4.5 Execution

We have at last reached our final phase—**Execution**—which serves to synthesize the efforts of the previous steps into a final, cohesive experience. This section will hopefully clear up any lingering questions regarding ReAL Sound’s *modus operandi* as we finally bring together all of our hard work.

The previous three phases have awarded us a **Plan**, which dictated our **Training** of a computer vision algorithm capable of transforming visual frame data into entity semantics, which were then combined with a audio-first **Design** that transformed our game state semantics into spatial audio cues. All that’s left to do is package the ReAL Sound framework into a final product—something **executable** on an end user’s computer.

If the implementor is adding ReAL Sound to their own game, then the entire package could be bundled in with their end product. In this case, most of the implementor’s job is already done. The only real question remaining is exactly how ReAL Sound’s functionality is exposed to the end-user. One could imagine that ReAL Sound lives happily in the accessibility settings of the game’s options menu—toggleable with the press of a single button.

If the implementor is instead adding ReAL Sound into an existing game as an after-market feature, then they must concoct their own method of distributing the software without violating relevant copyright laws. ReAL Sound of course lives entirely outside of the game itself, so this should present no problems. A fastidious implementor would likely create a simple application that a user runs alongside the target game—requiring they supply their own copy of the title.

To further illustrate our point, we will demonstrate a sample application—which attaches ReAL Sound as an aftermarket functionality—the following section.

Summary

The execution phase, much like the planning phase, serves more to illustrate a point: that the implementor must combine all of their efforts—their planning, computer vision model, design logic, and audio tools—into one concrete application which somehow reaches an end user’s hands as a simple, easy-to-use application.

To summarize:

Execution Phase

Requires: A proper software deliverable—capable of being run by an end user.

Involves: The bundling of the results of the previous phases into some sort of final product.

Purpose: Completes the ReAL Sound implementation process, finally allowing an end user access to the software.

Notes: ReAL Sound can be distributed as the implementor sees fit—bundled inside their own game, or released as a stand-alone software that runs alongside a target game.

3.4.6 Conclusion

In this section, I explored the process of implementing ReAL Sound into a target game. To accomplish this, I divided the process into four principle phases: the **Planning** phase—where the implementor analyzes the semantics of the target game before even touching a single line of code—the **Training** phase, where the implementor uses any number of computer vision techniques to translate frame data into entity detection and categorization techniques based upon the aforementioned plan—the **Design** phase, where the implementor creates an audio-first redesign of the game experience and actuates their design using conditional logic and programmatic imperatives—and finally the **Execution** phase, where the implementor packages the entire process into some deliverable executable that can be run on an end user’s device.

We will be honest in saying that this process may seem convoluted upon first glance. But one must remember that it is far less complex than the act of creating a video game itself. Through ReAL Sound, the implementor is free ignore most of the considerations that encumber development such as physics, engine-specific APIs, graphical libraries, etc. Moreover, the process of implementing ReAL Sound can essentially be restated in a few simple bullet points:

1. Understand the target game.
2. Teach a computer vision algorithm to understand the target game.
3. Translate visual-specific game design into audio-specific game design.
4. Create conditional logic to connect the output from the computer vision algorithm you designed to your audio-specific game design.
5. Package the entire experience.

These simple points obviously fail to capture the process fully, but work as a sufficient set of guidelines to summarize the implementation process. We hope this section has also successfully demystified the process for all readers.

3.5 The End User Experience

Lastly, it is pertinent to briefly discuss the experience of interacting ReAL Sound from the perspective of the end user. As the target user of ReAL Sound is visually impaired, significant care must be put towards delivering ReAL Sound in a fashion accessible to the visually impaired. General software development accessibility guidelines (such as those maintained by the W3C [142]) should be respected to ensure good ease-of-access.

ReAL Sound software should, ideally, 'just work' out-of-the-box. Even in the case of an aftermarket fan project, the end user ideally executes a small application adored with the bare essential settings required to fine-tune the user experience. From there, ReAL Sound should be considered a 'plug-and-play' software by the end user—one that allows them to simply launch the application, launch the target game, sit back, and enjoy.

3.6 Conclusion

In this chapter, we proposed ReAL Sound—a framework for generalizing and streamlining the creation and implementation of game accessibility features for the visually impaired.

We began by laying the conceptual groundwork of ReAL Sound. First, we limited our definition of video games using semi-formal notation. We then discussed how we may breakdown the gaming experience into formal, discrete states using automata theory. Following this, we considered how players understand games—taking a look at the semantics which underline game design and game performance. We concluded our discussion of conceptual topics by giving an in-depth look at how games are analyzed from the perspective of human vision.

We then proceeded with a structural outline of ReAL Sound. To to accomplish this, we broke the framework into three distinct 'layers:' the **Vision** layer, which is responsible for translating the game's visual data into semantic information—the **Decision** layer, which takes game semantics and assess it against pre-defined conditional statements, executing programmatic imperatives based on these conditions—and finally the **Audification** layer, which uses the imperatives from the previous layer to generate the spatial audio heard by our end user.

Following this, we then considered ReAL Sound from the perspective of an implementor—breaking down the process of implementing ReAL Sound into a target game. We subdivided the process into four stages: **Planning**—where the implementor first builds a model of the game's internal logic—**Training**, where the implementor trains a computer vision algorithm to understand the game based upon their previously model—**Design**, where the implementor creates a

design to translate game semantics into spatial audio outputs—and **Execution**, where all the previous phases are packaged together into a final executable, capable of being easily run by the end user.

Finally, we concluded by considering the experience of using ReAL Sound from the end user’s perspective. We provided clear stipulations on ReAL Sound being an accessibility-focused experience from the ground-up which requires accessibility and ease-of-use to be center stage during every step of the end user’s journey in interacting with ReAL Sound.

In the following chapter, we provide some sample implementations of ReAL Sound to further illuminate on the concepts presented here.

Chapter 4

Sample Implementation of ReAL Sound

In this chapter we present a sample implementation of ReAL Sound using common off-the-shelf programming tools. We present a worked example of a ReAL Sound implementation using *Pong* as our target game.

We begin first by discussing implementation specifics before outlining considerations in choosing a target game. We then discuss our design plan for *Pong*, followed by a detailed examination of the code specifics of our implementation.

We explore guidelines for future implementations in the next chapter.

4.1 Background

4.1.1 Choosing a Target Game

Although ReAL Sound is devised to be used in many games, it is true that certain games are better candidates for implementation than others. We particularly recognize the challenges and complexity in implementing ReAL Sound into a 3D game—where the added layer of spatial depth complexifies sound localization. Moreover, 3D games often feature intricate graphical styles, which presents challenges for the **Training** process and **Vision** layer. None of this means that 3D games are unfit for ReAL Sound, but instead provide additional challenges that are outside the scope of an introductory work. And so, we will limit our choice to 2D games.

But even limiting ourselves to two dimensions, we still have many other considerations to make. For one thing, how many entities and game **States** will we need to account for in our **Planning** and **Design**? A game like *Super Mario Bros.*, for example, features a litany of enemy and types and stage contexts. Consequently, we will also avoid complex games in the interest of brevity—although we have interest targeting more complex 2D games in our future work. We will consider future implementations in a later section.

New Challenges in Translating Design

A major question we must consider when choosing a target game is how feasible the translation of the game’s visual display is into a comprehensible audio display. A game like *Pac-Man* may have few entity types, but it still demands a nuanced understanding of multiple entities and spatial relationships. When the game begins, for example, how will the existence of hundreds of yet-undevoured pellets be represented to the user? How will the twists and turns which characterize *Pac-Man*’s maze? How will the player understand the exact location of that *one* remaining pellet—located halfway across the stage—which is needed for a stage completion? These questions are certainly solvable, but they provide significant **Design** challenges.

All of this brings us to a key point—that ReAL Sound provides new and interesting design challenges in the realm of games accessibility. Particularly, the challenge of translating visual displays into audio displays on a scale spanning the entire medium. *Just as game design has evolved through the development of the games industry, so too must we evolve audio game design.* For now, however, we leave this as an exercise for future work.

In the absence of a well-developed audio game design language, we will demonstrate our prototype using one of the simplest (yet most iconic) games available to us—the 1972 Atari game *Pong*. We shall build our own audio rendition that we call *Audio Arena: Pong Reloaded*.

4.1.2 *Pong*: A Brief Overview

Pong was developed by Allan Alcorn and released by Atari 1972 for arcades [107]. The game’s spartan design and early position in gaming history have lead it to be erroneously considered by many to be the ‘first’ video game ever released¹ [125].

Pong is a primitive simulation of Tennis in computer gaming format. In it, two players—represented by rectangular entities referred to as a **paddle**—are positioned on opposing sides of the screen. The paddles are locked to a one-dimensional axis, moving only upwards or downwards. A **ball** also appears during regular play. The ball begins at the center of the screen before moving in a random direction at a constant speed. The ball is capable of two-dimensional movement—ricocheting off of the top and bottom ‘walls’ (i.e., the screen-space) when struck. The ball’s velocity may change upon striking a player or wall, but it never has a meaningful acceleration.

If a paddle makes contact with the ball, it is reflected in the opposite direction. When the ball moves behind a player—going ‘off-screen’ on either the left or right side—the opposing player receives a goal point—which is indicated by a score displayed at the top of the screen. Upon scoring, the stage is reset

¹Despite the relatively recent introduction of video games, much of its history—including what the definitive ‘first’ video game is, is still a matter of open debate. Candidates include 1958’s pong-like *Tennis for Two* [30] and 1952’s *Draughts*, produced by noted computer science pioneer Christopher Strachey. For further information, see the excellent documentary *The First Video Game*, available at [4].

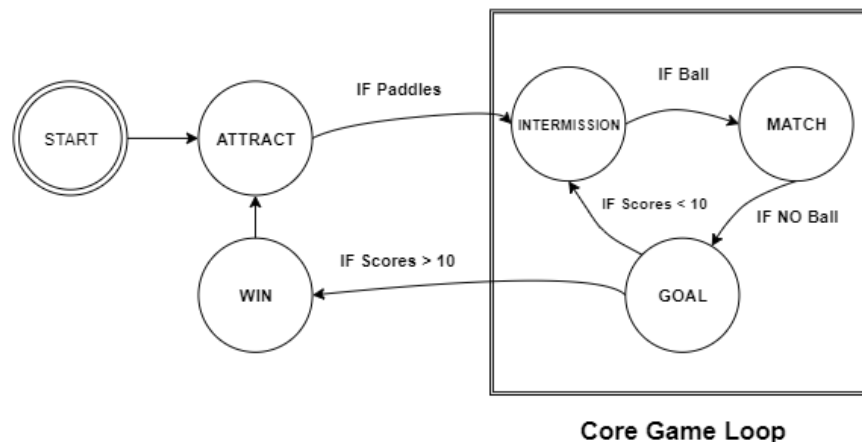


Figure 4.1: A **FSA** describing *Pong*

and a new match follows. The first player to receive ten points is declared the winner—returning the game to an infinitely looping **attract** sequence, which awaits player input (i.e., a coin to be inserted into the arcade machine).

With these two paragraphs, we have accurately described *Pong*’s entire operation in reasonable detail. We shall now consider how to translate the rules of *Pong* into a sensible audio display. This process resembles the **Planning** and **Design** phases.

4.1.3 Translating *Pong*

Planning

We start, then, with **Planning**. First, let us consider the entities, **States**, and **Rules** of *Pong*.

Entities As described above, *Pong* comprises two key entities—the player **paddle** and the **ball**. The scoring system, which is prominently displayed on-screen during matches, may also be considered an entity. However, our **Design** does not count the score signs as entities, so we will omit them from our discussion. This simplifies our implementation (arguably introducing weaknesses which we discuss in later sections), but other implementations may have use for them.

Each entity—no matter the type—has a **position** and **active** attribute. The first describes the entity’s on-screen position as a 2D cartesian coordinate, while the second attribute indicates whether or not the entity is currently extant on-screen. An entity may disappear from the screen (or, at least, from the algorithm’s analysis) for several consecutive frames, which renders the entity **inactive**, as we will discuss later.

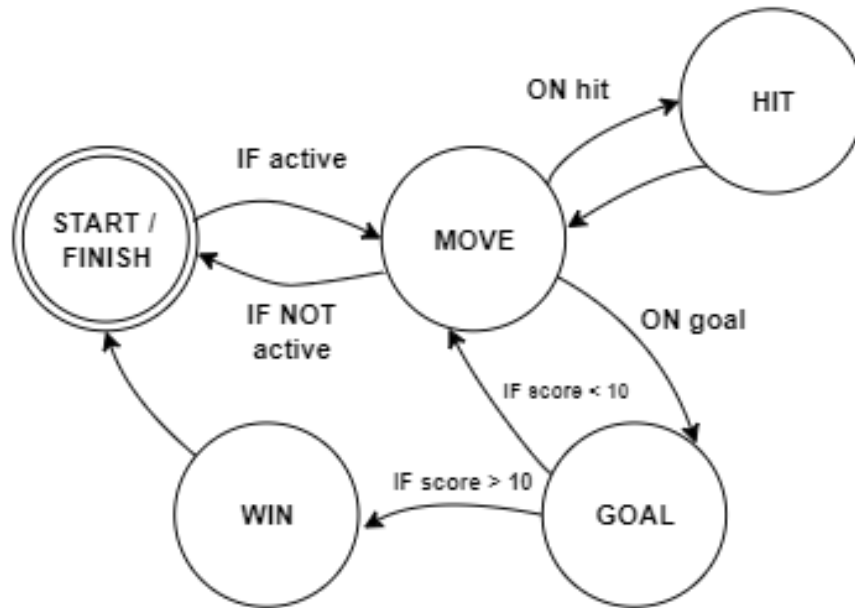


Figure 4.2: **paddle** FSA

Paddles Each player controls a **paddle** entity, which has a few key behaviors. The **paddle** is able to **move** up and down, and is capable of **hitting** the **ball**. Depending on the implementation, it may be worth noting that the paddle may also remain **idle** on-screen. Taking things a step further, some implementations may also wish to account for the fact that a **paddle** is capable of **scoring** a point.

Speaking of **attributes**, each **paddle** has an on-screen **position**, which we may measure in cartesian coordinate (x, y) relative to screen-space. Depending on the implementation, we may also want to note the **player numer** (i.e., player one or player two), as well as the player’s current **score** value.

Ball The ball may also **move** around the screen. When it when it strikes the top or bottom of the screen, it may **ricochet**. When a ball advances past the left or right sides of the screen, it becomes **inactive** and triggers a **goal** event. A **ball**, like all entities, has an on-screen position attribute. Other attributes could be included as-needed—such as ones describing the ball’s speed or direction.

States *Pong*, like many classic arcade games, can be easily understood as a finite state automaton. When the game launches, it continuously loops an **attract** screen—performing an automatic demonstration of the game (to *attract* spectators) until a player inserts a coin and presses the *start* button. **Attract** then transitions to what we will call the **pause** state—a brief pause that plays

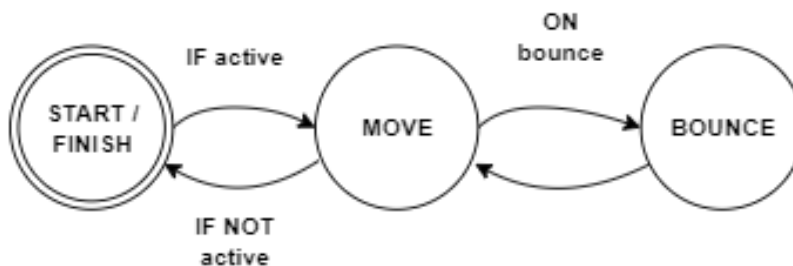


Figure 4.3: **ball** FSA

between each round of the match. After a few seconds, the normal **match** state begins. When a **goal** is scored, we return to the **pause** state for a few moments before starting another **match**. When enough points are scored, the set finally enters a **win** state, before transitioning back to the **attract** screen—waiting for the user to input more coins.

Entity States As hinted at above, entities have their own states, such as **move** and **goal**. These comparatively primitive state machines describe simplistic behaviors. A **paddle**, for example, may transition between any of the **moving**, **hitting**, and **idling** states at almost any time. The same is true for the **ball**.

One may arguably subdivide behaviors even further, but we find these description to be sufficiently detailed for our purposes. But despite the simplicity, there are still some semantic knots we must untangle before achieving competent **Planning**.

Delegating Responsibilities

Several key game semantics—such as the **hitting** of a ball or the **scoring** of a point—are behaviors shared between several entities. This leads to a question of semantic *responsibility*. Take, for instance, the act of scoring a **goal**. Who is responsible for handling this event? Is it the **ball**—which physically moved off-screen, causing the **goal**? Or is it the **paddle**—who was responsible for scoring the **goal**? A similar problem arises when a **ball hits** a **paddle**. Will the **ball** transition into a **hit** state? Or will the **paddle**?

We must *delegate* (to borrow the software engineering term) these responsibilities to the entities we see fit. There is no single correct answer in delegating, but there are still intelligent decisions to be made. For example, if we delate the **hit** state to the **paddle**, we can provide unique sound cues for each player’s hit—providing useful information for our audio display. On a similar note, we could delegate the responsibility of **goals** to the overarching game state machine, who can use this information to keep tally of the game’s score. We could also

delegate **goal** logic to the paddle entities—which could trigger a unique sound for each player’s score.

As in software architecture, we must consider the issues of *modularity* and *encapsulation* when drafting our **plan**. We take heavy influence from the seminal *Design Patterns* by the so-named *Gang of Four* [48]. In general, we aim to keep behaviors closely associated with the relevant entities without seeking external knowledge. In the case of **goals**, it makes sense to delegate to both the game state machine and the individual entities. The **ball**’s absence from the screen indicates to the game that it must transition from **match** to **goal**—wherein it triggers the **goal** state of the scoring **paddle**. From there, the game’s **goal** state transitions immediately to either **win** or **match** depending on if a score criteria has been met. **paddle** could trigger a special sound effect and increment their score—which has no bearing on the other entities. When the game’s **match** detects a **goal**, it could trigger the **goal** state on the relevant player.

Unlike scoring, **winning** the game does have consequences that affect other entities. When the game concludes, the state machine needs to effectively ‘reset’ itself back to the **attract** mode, meaning information like player scores must also be reset back to zero. Consequently, the **win** must be delegated to game FSM and not to the entities.

```
1 if(ball.active is False):
2     if(ball.x < screen.width/2):
3         p2.goal()
4     else:
5         p1.goal()
6     if(p1.score > WIN_SCORE or p2.score > WIN_SCORE):
7         self.state = win # ends game
8     else:
9         self.state = pause # next match
```

Code Sample 4.1: **Goal** and **Win** logic

There is no one correct answer to the problem of software design. The same is true of **Planning** for ReAL Sound. But it is true that maintaining good encapsulation of entity responsibilities leads to cleaner, simpler, and more reliable design. For instance, if the **ball**, **paddle**, and overarching game FSA each had their own **goal** state, then the implementor runs the risk of state misalignments. Consider a situation where a **paddle** mistakenly reports a **goal**—producing a special sound effect—while the other entities correctly assessed there was no goal and continued without ever transitioning to the **goal** state. This thorny problem crops up frequently in complex codebases. A common strategy to sidestep the issue is termed the **single source of truth** principle—which states data should only be modified in ‘one place’ (i.e., by one entity)².

²Examples of this approach can be seen all over the industry—perhaps most curiously with

Synthesis

To summarize our **plan** for *Pong*:

Game states We divide the game into four game states:

Attract The infinite loop which occurs before players begin a game.

Pause The short pause between matches.

Match The regular in-game loop.

Goal Trigger's paddle SFX and chooses transition to either **match** or **win**.

Win The 'end-state' of the game, occurring after one player scores ten goals.

Game Entities We recognize the existence of these entities and their respective states:

Paddle The player-controlled paddles which are constrained to one axis of vertical movement on the left and right sides of the screen. Capable of hitting the **ball** and scoring goals.

Moving Paddle is moving in an upward or downward direction.

Hitting Paddle is striking the **ball**.

Idle Paddle is unmoving.

Goal Paddle has scored a point. Plays audio cue uniquely indicating this player scored.

Ball Non-player controlled. Moves around the screen both horizontally and vertically. Ricochets if it strikes the top or bottom sides of the screen. Also ricochets if it strikes a player paddle. If it moves off the left or right sides of the screen, a goal is scored.

Moving Ball is moving across the screen.

Bouncing Ball is bouncing off the top or bottom walls of the screen.

We will continue to expand upon this **plan** in later sections. For now, we move to our discussion of implementation structure.

4.1.4 Languages, Tools, and Frameworks

A key aspect of ReAL Sound's design is its portable and cross platform nature. Although this spirit of universality is not strictly enforced, we have decided to target cross-platform tools in our own implementation of ReAL Sound. Let us start by reviewing our software choices.

google's monolithic code repository, which stores billions of lines of code making up the entire company's software ecosystem [120].

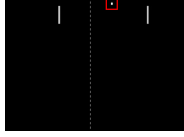
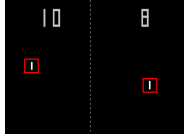

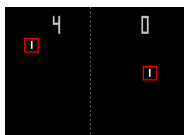
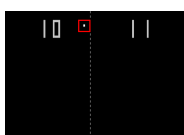
State	Description	Transitions (From)	Transitions (To)	Example
Attract	Pre-game attract screen	Start → Win →	→ Pause	
Pause	Short break between matches	Attract → Goal →	→ Match	
Match	A match of <i>Pong</i>	Pause →	→ Goal	
Goal	Player scores a point	Match →	→ Pause → Win	
Win	Player wins game	Goal →	→ Attract	

Table 4.1: Summary of Game **states**

Attribute	Description	Example
position	Position of entity on-screen	(250, 467)
velocity	Velocity of entity	(2.62, -4.61)
velocity_changed	Flag set if entity changed directions this frame on a given axis	(True, False)
active	Flag set if entity is considered visible on-screen	True

Table 4.2: Summary of general **Entity** Attributes



Name	Attributes	States	Visual
ball	N/A	bounce	
paddle	score	hit goal win	

Table 4.3: Summary of *Pong*’s entities: the **paddle** and **ball**

Python

The **python** programming language is renowned for its legible, high-level writing style as well as its incredibly portable nature. The enduring popularity of the language has also endeared itself to a mature package ecosystem with countless easy-to-access libraries. Combined with the popular **pip** package manager, extensions to the language feel ‘off-the-shelf’ and allow the linking of complex libraries in a matter of seconds.

OpenCV

As referenced in previous chapters, **OpenCV** has become the *de facto* standard for computer vision implementations. The library comes bundled with hundreds of useful machine learning and computer vision algorithms that are deployable using only a handful of lines of code.

OpenCV is also extremely cross platform. The primary C++ libraries would present some complications—but, as luck would have it, a python port of OpenCV is available via the **opencv-python** and **opencv-contrib-python** modules—allowing for simple python implementation of computer vision algorithms on virtually any modern machine.

NumPy

We also require the equally popular **NumPy** python package—which will serve as our stand-in for OpenCV’s C++ native **Mat** (i.e., **Matrix**) construct. Numpy extends python with a rich collection of high-level math functions as well as high-performant multi-dimensional arrays. These arrays will be our primary method of manipulating image data, making NumPy key for our implementation. It is also cross-platform and installable with a single execution of the **pip install numpy** command.

Qt

We also require tools for building and packaging a proper application. We have opted to use the ever-popular **Qt** suite of software libraries and tools. Qt’s functionality is extremely diverse, and we will make use of it to create everything from graphical user interfaces, to video I/O, to even spatial audio.

Although originally written for C++, Qt also has extensive python support via its **PySide6** library. This package is once again cross platform, and is installable via `pip install PySide6`. We make specific use of the **QtMultimedia** library’s spatial audio engine.

Miscellaneous Software

Although not required for implementation, we also make use of **git** and **GitHub** for version control, as well as **Visual Studio Code** for code editing. There are, of course, other dependencies which were used during development—but these packages solve minor problems not worth detailing here. Regardless, we ensured that all utilized packages were cross-platform and sufficiently open sourced for our purposes.

We should also note that a litany of free sound effects were taken from freesound.com, with relevant attributions being placed in the source code.

4.2 Structural Overview

Our implementation is structured as a simple python module named **realsound**. With this approach, we are able to easily update, modify, package, and share our implementation with other developers. We organize our module into a few submodules: **resources**, which stores configuration settings, **core**, which contains core objects and models such as the definitions of **states** and entities, **cv**, which contains our computer vision algorithms, and finally **qt**, which stores all of our application data—the GUI implementation, screen-capture technology, and spatial audio generation tools included.

The **core** files include: **vision.py**, which defines our computer vision functions; **decision.py**, which includes our game state machine; **audification.py**, which manages the spatial audio engine as well as **AudioObjects**; and **entity.py**, which defines the general **entity** concept as well as the specific **ball** and **paddle** entities.

All of these classes are tied together by an overarching **Client** class found in the **client.py** file. The client is responsible for all high-level functions of ReAL Sound—I/O, managing the GUI application, and facilitating communication between the layers when necessary. In other words, the client can be considered the main ‘entrypoint’ of ReAL Sound.

Finally, various testing code is found within the **tester.py** and **dummy.py** classes. The total object dependency structure is described visually in fig. 4.4.

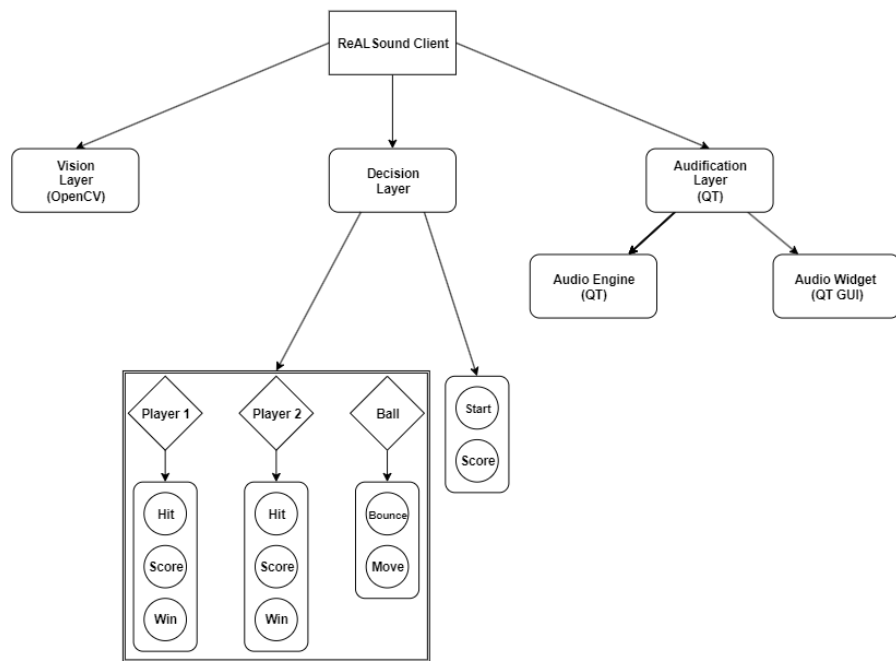


Figure 4.4: An overview of ReAL Sound's object structure

4.3 Planning & The Vision Layer

We begin our implementation with the **Vision** layer.

Algorithm of Choice

We choose to employ the Harris method of corner detection, which we have detailed in the previous chapters. We do so using the `cv.cornerHarris()` function found within the OpenCV library [110]. The function takes several inputs—a data source (provided as an 8-bit grayscale image), as well as several algorithm specific parameters³—and outputs an array of detected corner points within the image. We chose the Harris corner detector for its ease-of-use, ubiquity, and relevance to our target game (as *Pong* entities all have sharp and distinct corners).

We experimented with other common algorithms, such as **template matching**—which compares regions of screen data with image **templates** provided by the user. This system works well for many 2D scenarios with a constant perspective—requiring only one template image per item. Unfortunately, *Pong* is an edge case where the algorithm struggles on account of the game’s graphical simplicity, as seen in fig. 4.6 and fig. 4.5. All shapes in the game are white and rectangular, which easily confuses template matching systems. After all, how is anyone—player included—supposed to distinguish a **paddle** from a score of ‘one,’ which have virtually identical representations.

Providing Input

Of course, our algorithm is only as good as the data it receives. And so, we must conceive of a method for sending the game’s frame data into the Harris corner detector. Numerous methods exist, but many are operating system dependant, which would violate our design principles. Luckily for us, Qt has recently implemented cross-platform screen and window capture support⁴—which we make use of.

The specific details of implementation are lengthy, but to summarize: we make use of the `PySide6.QtMultimedia.QScreenCapture` class and its related classes. We identify the relevant window using user input and copy that window’s framebuffer into memory using a `map()` function. We then pass the copied buffer into ReAL Sound, where it is used as the input for the harris algorithm. Those curious for more information may consult the relevant documentation at [81] as well as a sample implementation at [82]. We attach a ‘window selection tool’ to our final application, which allows users to quickly select the correct window that ReAL Sound will read for image data.

³Being the `blockSize`, `ksize`, `k`, and `borderType` parameters. Whose specific details are irrelevant to our work.

⁴We should note that these features are, in truth, as cross-platform as *possible*. There exist hardware configurations left unsupported, but these tools will work for the vast majority of end users.

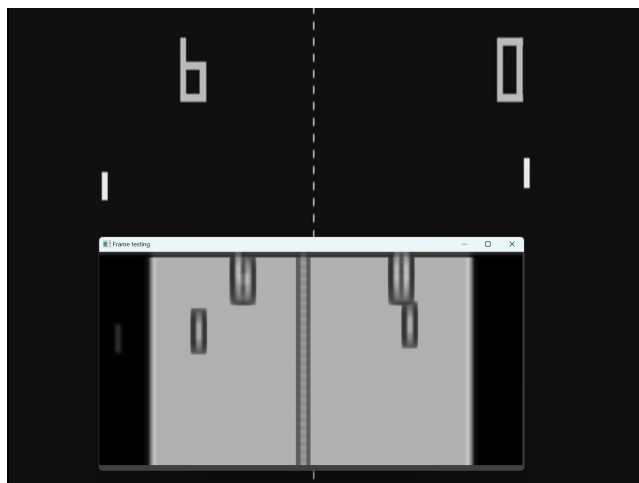


Figure 4.5: Sample of a well-tuned **template matching** 'heat-map.' Notice how the scores are flagged as matching the template

Testing on Video We also included functionality to read a video file as input data for testing purposes. For this, we recorded a short video of a full set of *Pong*. Video playback allows for faster debugging and iteration of our tools, as we may test changes in code immediately without needing to play yet another game of *Pong*. The underlying principles are identical to the process above—simply replacing the mapped framebuffer with a framebuffer received directly from the video file.

Detecting Objects

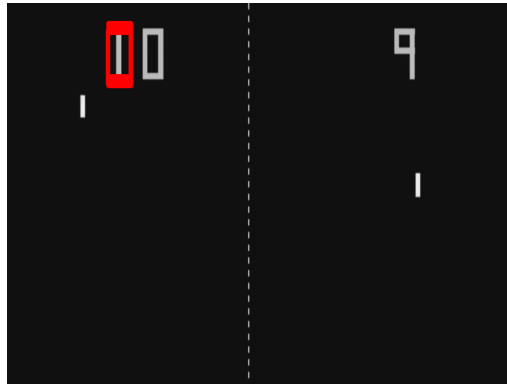
Once the harris algorithm is handed the image data, it promptly returns a collection of points it asses to be the corners within the image. Of course, corners are only the start of our solution. We are searching for *Objects* within those corners. In some cases, object detection would be a task better suited for machine learning. Luckily, the entities of *Pong* are consistent in size and always maintain a basic, rectangular shape. This allowed us to implement our own bespoke method of detecting and classifying objects within a given frame.

The general logic goes like this:

Ball If a collection of four corners resemble a perfect square shape, then a **ball** is detected.

Paddle If a collection of four corners resemble a rectangle with a specific aspect ratio, then a **paddle** is detected

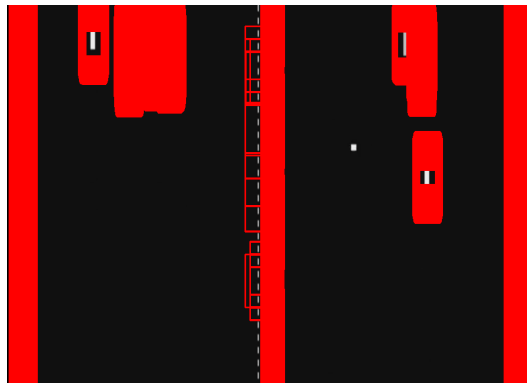
1. If the **paddle** is near the left side of the screen, it is the 'left' (player one) **paddle**. If the **paddle** is near the right side of the screen, it is the 'right' (player two) **paddle**



(a) Under-tuned



(b) Well-Tuned



(c) Over-Tuned

Figure 4.6: Samples of **template matching** with different parameter tunings. Notice how even the 'well-tuned' algorithm misidentifies the score objects as **entities**

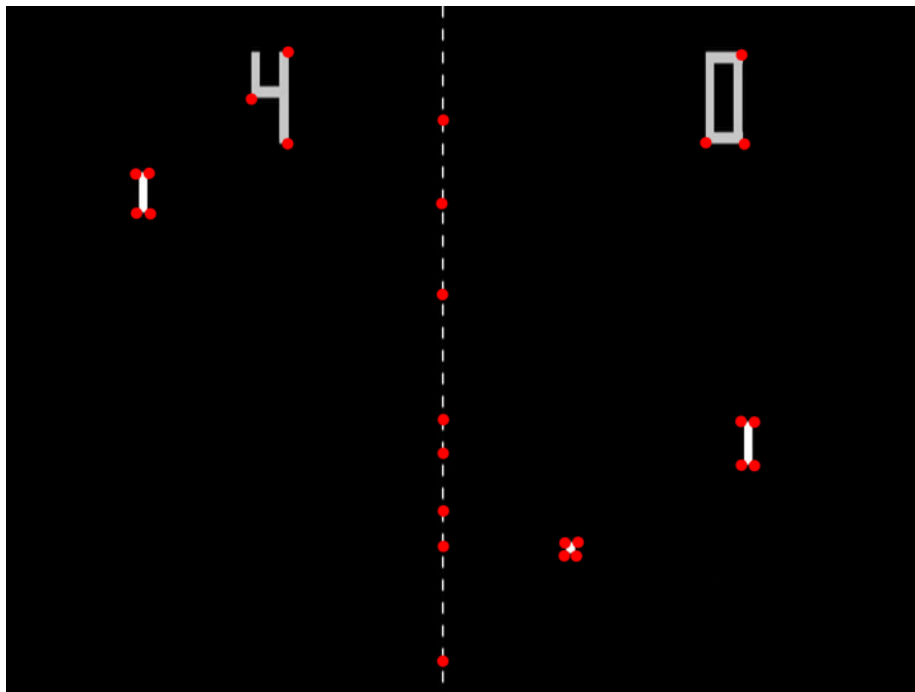


Figure 4.7: An example of Harris corner detection output. Note how numerous non-entity corners are flagged by the algorithm

```

1 # Find similar rect points
2 test_points = candidates[
3     np.argwhere(
4         np.logical_xor(
5             is_close(candidates, i)[: , 0],
6             is_close(candidates, i)[: , 1],
7         )
8     )
9 ].squeeze()
10
11 # Find two that match our rect
12 if (
13     test_points.shape[0] > 1
14     and len(test_points.shape) > 1
15     and not np.any(is_close(test_points[: , 0], test_points
16                          [: , 1]))
17 ):
18     if test_points.shape[0] > 2:
19         # Find the points closest to i
20         test_points = test_points[np.argmin(abs(test_points
21                                                  - i), axis=0)]
22
23         # append our new points to the resulting object
24         result[1:3] = test_points
25
26 # Fourth point by induction
27 p4 = test_points[np.where(is_close(test_points, i) != True)]
28 result[3] = p4
29
30 return result

```

Code Sample 4.2: Abbreviated **detect** algorithm

The specific nature of 'perfect square' and 'aspect ratio' are somewhat fuzzy for the **ball** and **paddle** respectively. We allow for slight (one to three pixel) margins of error on shape calculations. We also consult the aspect ratio of the frame itself to calculate the paddle and square's appropriate aspect ratio.

The code for achieving these categorizations is somewhat obtuse to the untrained eye. This is due to the nature of NumPy array notation, which requires some familiarization. As we strive for optimal performance (thirty or more frames calculated per second), we endeavored to make the most out of this notation. To address potential confusion, we have thoroughly documented our code, and provide snippets of the object detection and classification algorithms in Code Samples 4.2 and 4.3.

We find that these methods successfully identify objects and classify entities roughly 99% of the time. Our testing of 1500 frames saw only 15 go misidentified—

```

1 results = {"p1": None, "p2": None, "ball": None}
2 if len(groups) == 0:
3     return results
4
5 # Get widths/heights to distinguish each object
6 w = groups[:, 2, 0] - groups[:, 0, 0]
7 h = groups[:, 1, 1] - groups[:, 0, 1]
8
9 # Check for ball
10 # height / width < 2 (square)
11 ball = groups[np.argwhere(h / w < 2)].squeeze()
12 if len(ball) == 4:
13     results["ball"] = ball
14
15 # Check for paddles
16 # height / width > 2 (rectangle)
17 paddles = groups[np.argwhere(h / w > 2)].squeeze()
18 if len(paddles) == 2:
19     # Order paddles by direction (left first)
20     if paddles[0][0][0] < paddles[1][0][0]:
21         results["p1"] = paddles[0]
22         results["p2"] = paddles[1]
23     else:
24         results["p1"] = paddles[1]
25         results["p2"] = paddles[0]
26 return results

```

Code Sample 4.3: Abbreviated `classify` algorithm

an error rate of 1%.

Handling Errors

Some of these errors are actually key points for our implementation. Take for example the moment a **ball** strikes a **paddle**. When the two objects collide, their bounding boxes 'become one'—at least, from the perspective of the corner detection algorithm, which will 'lose track' of the objects on the frame of impact. The consistency of this misidentification is used as a trace for detecting **hits** (fig. 4.8)! Of course, we also validate the result by comparing the locations of the impacted objects—if the objects are very close, then they likely struck each other this frame.

Continuity There are, however, other errors that are not so beneficial. These errors primarily occur whenever an object is occluded by another—which usually happens when the ball moves near the large score symbols at the top of the

screen, as seen in fig. 4.9. Much like when striking the **paddle**, the **ball** corners 'merge' with the score corners, creating a frame or two of bad data.

Of course, we cannot allow this rare errors to hamper the playing experience. If a player were listening to the position of the ball, only to hear it suddenly jolt 'in-and-out of existence,' their ability to understand the game's state would be severely reduced. We can address these problems by implementing a basic sense of **cross-frame persistence**. That is to say, we save the data from this frame and persist it into the next one. Using this data, we see if the changes in an objects state are sensible. For example, if an object suddenly goes from the position (900, 540) to (0, 0) in one frame—its almost certain that an error has occurred. If so, we re-use the previous frame's data until the object returns to sensible values in following frames. If the object continues to provide poor data, the system considers the object 'lost' and discounts it from calculations until it once again returns with reasonable data. A sample of this logic is seen in Code Sample 4.4

What exactly constitutes 'reasonable' or 'sensible' data is context specific. In the case of *Pong* we demand that the **ball** and **paddle** move with a sense of continuous motion. Sudden jumping around the screen (due to poor frame data) will not be tolerated. This may sound in theory like a significant drawback, but in practice has very little effect on the experience. As we mentioned above, bad frame data is a rarity and usually occupies only a fraction of a second of play-time. Virtually no players will notice that audio data from 35 milliseconds ago is being reused while we await better results.

Summary

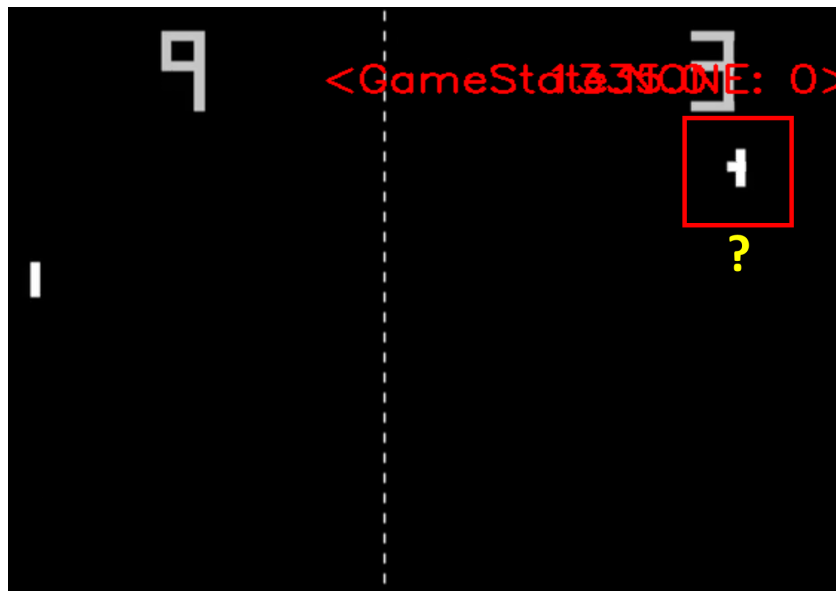
These choices—from OpenCV and the harris algorithm, to Qt and our own bespoke code—comprise the **Vision** layer. In order to discuss the **Decision** layer, we must first take a moment to consider matters of **Design**.

```
1 new_velocity = new_position - position
2
3 # If new velocity isn't impossible
4 # i.e., an incorrect object
5 # was accidentally flagged as this entity
6 if np.any(abs(new_velocity) > VELOCITY_MAX):
7     lost_frame()
8     return
```

Code Sample 4.4: Continuity detection

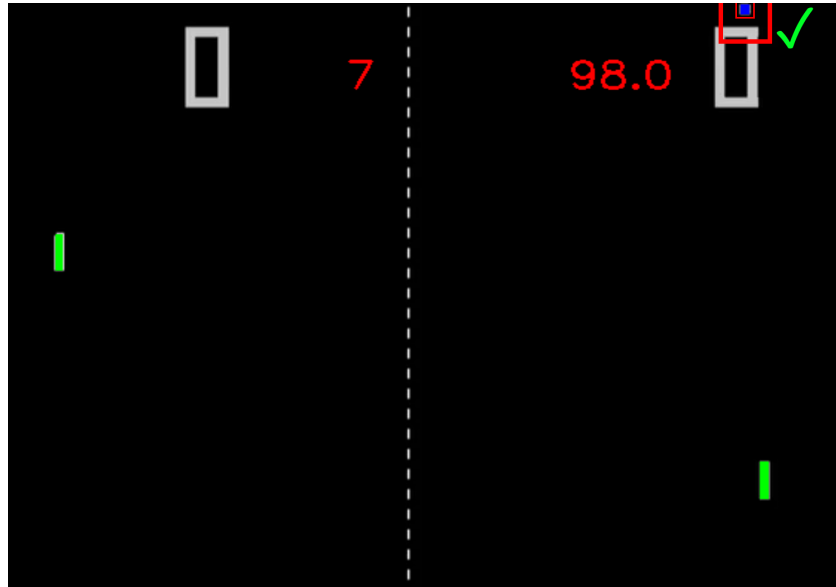


(a) Frame before Player 2's **paddle** hits the **ball**

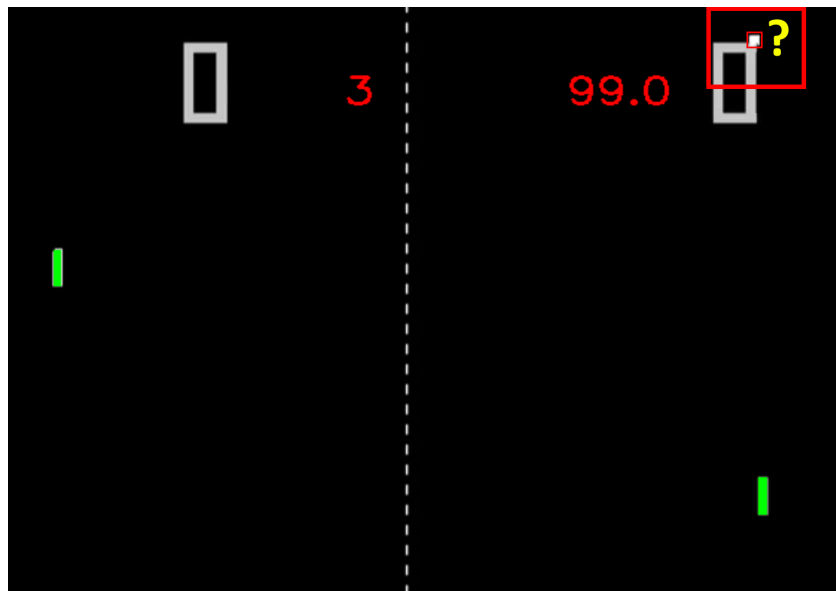


(b) Frame of impact. Notice how object tracking for both Player 2's **paddle** and the **ball** is lost this frame (as their corners overlap)

Figure 4.8: Example of sudden continuity loss during a **hit**—which may be used to detect the event



(a) Frame before losing **ball** tracking (Normal operation)



(b) Frame of lost **ball** continuity. Notice how the **ball** overlaps corners with the score—confusing the CV algorithm

Figure 4.9: Example of tracking continuity issues with the Harris algorithm. The ball is highlighted in red for emphasis

4.4 Design & The Decision Layer

Up to this point, we successfully created **Planning** which categorized the entities within *Pong*. Then we built the **Vision** layer of ReAL Sound using OpenCV, the harris corner detection algorithm, and our own bespoke code. Now, we must create a **Design** of *Pong*'s rules and behaviors, which we will then use to create the **Decision** layer.

Decision Semantics for *Pong*

Designing State First, let us begin by considering the **states** we defined in our **Planning**. More specifically, we must devise how we *distinguish* between each state. Ideally, ReAL Sound runs from *Pong*'s launch, meaning it is able to follow along with the state machine from **attract** all the way to an eventual **win**. However, there are also cases where ReAL Sound may be attached to a game already in progress, necessitating the need for context-invariant state identification.

In the case of *Pong*, we may identify the current state using the presence (or lack) of entities on-screen. When the game is in **attract** mode, for example, only the **ball** is present on screen. Code Sample 4.5 provides some examples of detecting **state** using our semantics.

```
1 if ball.active and not (p1.active and p2.active)
```

(a) **Attract**

```
1 if ball.active and (p1.active and p2.active)
```

(b) **Match**

```
1 if (p1.active and p2.active) and not ball.active
```

(c) **Pause**

Code Sample 4.5: Decision Semantics

Complications arise when we lack key semantic data not currently shown on-screen. Our current implementation does not consider the score tally at the top of the display as entities, for example. Because of this, ReAL Sound requires itself to be attached to the game *before* the game begins to accurately keep score. If the player were to attach ReAL Sound halfway through the match, it would mistakenly begin counting score from zero, even if players had already scored points. This could be solved if scoring symbols were treated as entities—allowing them to be analyzed for their semantics per-frame. We omit this for the purpose

```

1 def lost_frame():
2     lost_frames += 1
3     if lost_frames > MAX_LOST_FRAMES:
4         deactivate()

```

Code Sample 4.6: Sample of object persistence code

of simplifying our demonstration here. And so, we presume the player attaches ReAL Sound to the game during the **attract** screen.

Another issue arises from per-frame contextual analysis. For example, if the **ball** disappears for a single frame—perhaps because it is occluded by the score UI—then what separates it from an **pause** state (**if paddle and not ball**)? To overcome this issue, we provide specific demands on object persistence—verifying that objects are missing for several consecutive frames before considering a state transition. In otherwords, an object may be allowed to ‘disappear’ for, say, six frames. As long as the object ‘returns’ to view before being missing for seven consecutive frames, then the algorithm will simply update the object as-normal—without counting it as being a ‘newly’ appearing **entity**. This adds a slight theoretical delay to ReAL Sound (since we must objects are truly lost over the course of several frames), but the lag is in the magnitude of miliseconds and is generally unnoticeable.

State	Trace	Transition Logic
Attract	ball	p1 and p2 → Pause
Pause	p1 p2	ball and (p1 and p2) → Match
Match	p1 p2 ball	p1 → p2.hit p2 → p1.hit p1 and p2 → Goal ball → Win
Goal	p1 p2	p1.score < 10 and p2.score < 10 → Pause p1.score > 10 or p2.score > 10 → Win
Win	ball	→ Attract

Table 4.4: Summary of **Design** semantics

We summarize our state semantic analysis in table 4.4. The **Trace** column indicates which **entities** comprise the given **state**. For instance, **Attract** is only detected when there is a **ball** and *only* a **ball** on screen. If the **ball**

were to disappear, and both player 1 and player 2⁵ **paddles** appear, then we must transition to the **Pause** state. The **Match** state also does some special handling of the **paddle**'s **hit** state. As discussed in the previous section, a **hit** occurs whenever a **ball** and **paddle** collide. This collision confuses the detection algorithm, which fails to detect the colliding objects—leaving only the *non-colliding* object on screen, which we use to detect hits (if only **p1** is visible, then **p2** must have registered da hit).

State Transitions Although ReAL Sound performs per-frame analysis, we must also provide transitional logic for moving between states. For example, when the **ball** disappears off-screen for several consecutive frames, we transition to the **goal** state, which updates the **points** attribute for the scoring player and plays a unique score sound effect. We then immediately transition to the **break** state, which will transition to **match** once it detects **paddle** and **ball** entities on-screen together. We also consider the score when transitioning from **match**. If a player's **points** attribute is equal to eleven, then we move to the **win** state instead of returning to a **match**.

Entity States and transitions Entity states are handled in a similar fashion. If a **paddle** for example is in the idle state but has performed a movement (i.e., its position has changed when compared to the previous frame), then it transitions to the **moving** state. If a **hit** is detected this frame, it transitions to **hit** for one frame before moving back to either **idle** or **moving**.

Designing an Audio Display

As with **Planning**, there is no one correct way to **Design** an audio display. Regardless of our choices, we must be sure to provide clear, reactive, and understandable audio that accurately maps the game's semantics. A naive approach to an audio display might translate many of *Pong*'s elements directly. Each **paddle** would produce a pitch in either the left or right audio channel depending on their on-screen position. The frequency of the pitch produced would vary depending on the height of the paddle on screen. The ball, meanwhile, would translate its on-screen position into a stereo panning effect while producing a similar pitch to the paddles. Although this **Design** accurately models the behaviors of *Pong*, it is not easily understandable to the average user—who would be forced to contend with three active pitches simultaneously moving about them. Let us now consider some improvements.

Player Uniqueness & Relativity The act of using headphones is an inherently isolating experience—unless a user performs the (fairly irregular) act of sharing their headphones with another user *while* in a state of play. That being the case, we presume ReAL Sound is for one player enjoying *Pong* using

⁵Frequently shortened to **p1** and **p2**

headphones. We use this assumption to improve design. *How?* By removing the second player from the equation entirely.

Although there is perhaps some semantic value gained in seeing what the opponent is doing, it is hard to deny—at least from a design perspective—that the game is, functionally, a single-player experience. If a player’s opponent (say, a perfect AI) is sufficiently good at the game—knocking back every ball the player sends their way—then the player is, for all intents and purposes, playing against a *solid wall* instead of a real competitor. This observation was commonly put into practice by several ‘solitaire’ variants of *Pong* that were released in the pre-AI world of game development. By removing the other player from the equation, we can focus on providing high quality audio data for our two remaining entities: the player **paddle** and the **ball**. But we can go one step further in our reduction as well.

We must now consider a question:

What is the point of Pong?

To which we answer:

*To accurately strike the **ball**.*

Our answer is, of course, a matter of subjective philosophy. And yet, the point still remains—skilled play of *Pong* requires the return of a ball towards the opponent. We should notice something about our answer: *it does not involve the player at all*. In fact, the answer revolves *entirely* around the **ball**. As the player is allowed only one axis of free movement (up-and-down the screen), understanding information about the comparatively complex 2D **ball** movement is more valuable to the player. And so, we perform yet another simplification of our design—crafting a display *from the player’s point of reference*.

Imagine, if you will, that we were were the **paddles** in a real-world game of *Pong*. Lets call this game, for the sake of imagination, *Tennis*. Let us also pretend that the **ball** emits a distinct sound. If the **ball** were being served to us, we would first hear the sound *at a distance*. As the **ball** approaches us, the volume of the sound would increase—reaching its apex as it makes contact with us. The sound then diminishes as the ball returns to the other player. This model of pong—reflecting intuitive, real-world behaviors—is an ideal for an audio display.

4.4.1 Code Samples

Each state condition is fairly simplistic, as we demonstrated with our pseudocode in previous chapters. For the sake of thoroughness, we present all of the state machine’s real logic.⁶

⁶Well, *mostly* real logic. Slight edits have been made for readability and in order to fit the margins of the page—particularly through the removal of python’s pesky **self** keyword. The **pause** state is really named **intermission** in code. This is because **pause** is unused to avoid confusion with audio processes, and the more ubiquitous word **break** is a reserved word in python—meaning we cannot use it on our code.

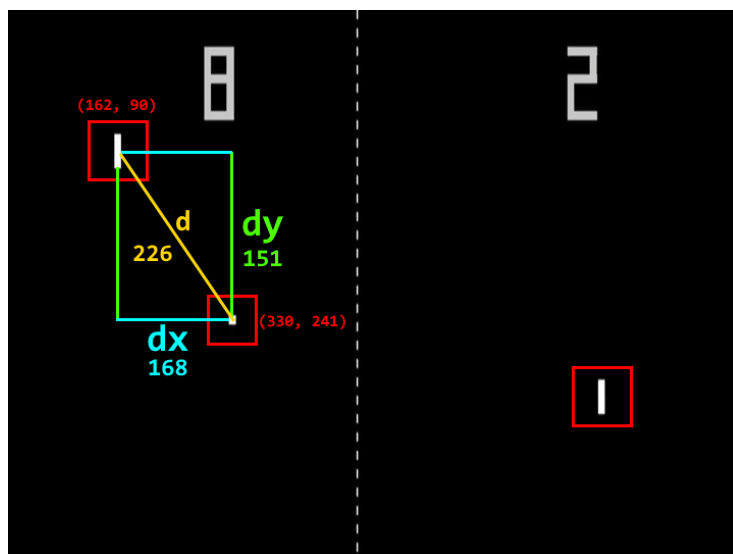


Figure 4.11: Example of key metrics for our player-relative audio display: the distance between player one’s **paddle** and the **ball**.

To begin, the corner data calculated by the **Vision** layer (`new_corners`) is passed to the **Decision** layer via the `decide` function, which uses this data to update entity information and calculate the next state—as seen in Code Sample 4.7.

```

1 # Runs every frame
2 # Takes corner data as input from Vision Layer
3 def decide(new_corners):
4     # Update entity positions
5     for entity, corners in new_corners.items():
6         entities[entity].update(corners)
7
8 current_state = current_state()
```

Code Sample 4.7: **Decision** header

The logic for each **state** (represented as unique functions) follows below `decide`. Each function returns a **state** function, which will execute next frame. The returned function essentially acts as the state we ‘transition to’ for the next frame. Take, for example, the `attract` state seen in Code Sample 4.8.

As we can see here, our **states** resemble the pseudocode described in previous sections. In essence, the algorithm checks for the presence of certain entities (via the `.active` variable) and makes corresponding transitions. Similar examples

```

1 def attract():
2     if p1.active and p2.active:
3         if ball.lost_frames > 0:
4             ball.active = False
5             return pause
6         else:
7             return match
8     else:
9         return attract

```

Code Sample 4.8: **Attract** logic

```

1 def pause():
2     return match if ball.active else pause

```

Code Sample 4.9: **Pause** logic

are seen in Code Samples 4.9 and 4.10.

The **match** function seen in Code Sample 4.11 is comparatively complex, as most of the game's logic is located here. This function must check for 'interesting' events—such as a **goals**, **hits**, and **bounces**. If nothing of note happens this frame (i.e., the game is running normally), then the function simply returns a reference to itself—indicating no transition this frame. A few constants defined outside the function (**HIT_DIST**, **GOAL_BUFFER**, **WALL_DIST**)—which define margins of error (how close the ball needs to be to players, walls, or the edges of screens)—are used. The utility function **dist()**—which measures the distance between entities—is also used.

As you can see, game states are used to update **entity** states—as seen via the invocation of **ball.bounce()** or **p2.goal()**

A similar logic is used for entity updates in Code Sample 4.12. Here, we input the corner data for each entity, which we use to update attributes such as position and velocity:

We also require logic to handle frames where the entity is considered 'active'

```

1 def win(winner):
2     winner.score += 1
3     winner.win() # win SFX
4     return attract

```

Code Sample 4.10: **Win** logic

```

1 def match():
2     # If the ball's X velocity changes
3     # There must have been a hit
4     if ball.velocity_changed.x:
5         if dist(p1.position, ball.position) < HIT_DIST:
6             p1.hit()
7         elif dist(p2.position, ball.position) < HIT_DIST:
8             p2.hit()
9
10    # If the ball's Y velocity changes
11    # There might have been a bounce
12    # If it bounced off a top/bottom wall
13    elif ball.velocity_changed.y:
14        # If near a wall
15        if (
16            ball.y < WALL_DIST
17            or
18            parent().frame.shape.x - ball.y < WALL_DIST
19        ):
20            ball.bounce()
21
22    # If the ball goes missing long enough
23    # There must have been a goal
24    if not ball.active:
25        if ball.x + GOAL_BUFFER > p2.x:
26            p1.goal()
27        elif ball.x + GOAL_BUFFER < p1.x:
28            p2.goal()
29        return pause
30
31    # If the paddles go missing long enough
32    # AND the ball is on a certain side of the screen
33    # The other player has won
34    if not (p1.active and p2.active):
35        if ball.x > (parent().frame.shape[1] // 2):
36            return win(p1)
37        else:
38            return win(p2)
39
40    return match

```

Code Sample 4.11: **Match** logic


```

1 def update(new_corners):
2     if new_corners is None:
3         return lost_frame()
4
5     new_position = (new_corners[0] + new_corners[3]) / 2
6
7     # Special Case when spawning or first discovered
8     # No velocity or other similar data this frame
9     if not active:
10        activate()
11    else: # Otherwise, calculate new velocity
12        update_velocity(new_position)
13
14    corners = new_corners
15    position = new_position
16    # Convenience accessors
17    dimensions = new_corners[3] - new_corners[0]
18    x = position[0]
19    y = position[1]
20    w = dimensions[0]
21    h = dimensions[1]
22    top = new_corners[0][1]
23    bottom = new_corners[1][1]
24
25    lost_frames = 0

```

Code Sample 4.12: The generic **entity** update logic

but is not found in data—as seen in section 4.4.1. Finally, we require code for updating the object’s velocity data, as seen in Code Sample 4.13.

4.4.2 Special Entity Logic

The **paddle** and **ball** naturally have their own unique behaviors which requires additional code. Most behaviors are simple one or two-line functions that represent our entity states, as evidenced in ??.

Code Sample 4.14 meanwhile reveals that the **ball** has more complex behaviors which require additional attributes:

As our design orients around the **ball**, we must keep track of a few metrics—the time between audio playback (**beep_speed**), the time when the last sound was generated (**last_beep**), and which pitch should be generated (**pitch**).

The **update** function requires similar attention—as seen in Code Sample 4.15 and supplementary code found in Code Sample 4.16

Using these conditions, we have successfully built the **Decision** layer using our **Design** principles. The code may be far from perfect, but it is no doubt

```

1 def lost_frame():
2     lost_frames += 1
3     velocity_changed = np.full((2), False)
4     if lost_frames > MAX_LOST_FRAMES:
5         deactivate()

```

(a) `lost_frame` function

```

1 def activate():
2     active = True
3     velocity = np.zeros((2))
4     # No velocity changes
5     velocity_changed = np.full((2), False)
6
7 def deactivate():
8     active = False
9     moving = False

```

(b) `activate` and `deactivate` logic

brief, readable, and generally easy to follow, even as a non-coder.

```

1 def update_velocity(new_position):
2     new_velocity = new_position - position
3
4     # If new velocity isn't impossible
5     # i.e., an incorrect object across the screen
6     # was accidentally flagged as this entity
7     if np.any(abs(new_velocity) > VELOCITY_MAX):
8         lost_frame()
9         return
10
11    # If new and old velocity aren't zeros
12    if np.any(new_velocity) and np.any(velocity):
13
14        # If we're moving this frame
15        moving = np.any(abs(new_velocity) > VELOCITY_MOE)
16
17        # If the velocity changed signs
18        # And is beyond a basic MOE
19        velocity_changed = (
20            (np.sign(new_velocity) != np.sign(velocity))
21            & (abs(new_velocity) > VELOCITY_MOE)
22        ).squeeze()
23    else: # if Zeros, treat like no velocity change
24        velocity_changed = np.full((2), False)
25
26    velocity = new_velocity

```

Code Sample 4.13: Generic **Entity** update_velocity function

```

1 class Ball(Entity):
2
3     MIN_BEEP_SPEED = 0.4
4     MAX_BEEP_SPEED = 0.05
5     on ricochet = Signal() # Bounce event
6
7     def __init__(name, parent):
8         super().__init__(name, parent)
9         last_beep = time.time() # Last sound timestamp
10        beep_speed = Ball.MIN_BEEP_SPEED
11        pitch = Pitch.GOOD

```

Code Sample 4.14: **ball** init logic

```

1 def update(new_corners):
2     super().update(new_corners)
3
4     # If not in a match, exit!
5     if parent().current_state != parent().match:
6         return
7
8     # Calculate new beep speed
9     beep_speed = calc_speed(
10         parent().p1, client.frame_width
11     )
12
13     # Update pitch value
14     pitch = calc_pitch(parent().p1, client.frame_height)
15     if pitch != pitch:
16         pitch = pitch
17         audio_objects["move"].switch_sound(pitch)
18
19     # Update panning
20     audio_objects["move"].update_panning(x)
21
22     # Check if we should beep
23     now = time.time()
24     if now - last_beep > beep_speed:
25         beep()

```

Code Sample 4.15: The **ball**'s special update logic

```

1 def calc_speed(paddle, ball, width):
2     dx = dist(paddle.x, ball.x)
3     # Return value between MIN and MAX speeds
4     return min(
5         abs(
6             Ball.MAX_BEEP_SPEED
7             + safe_ratio(ball.x, width)
8             * (Ball.MIN_BEEP_SPEED - Ball.MAX_BEEP_SPEED)
9         ),
10        Ball.MIN_BEEP_SPEED,
11    )

```

(a) Calculate's the **ball**'s current speed

```

1 def calc_pitch(paddle, ball, height):
2     # Ball is 'hittable' by player
3     if paddle.top <= ball.y <= paddle.bottom:
4         return Pitch.GOOD
5     dy = ball.y - paddle.y
6     if dy < 0: # Ball above player
7         return Pitch.HIGH
8         if dy < paddle.top / 2
9         else Pitch.HIGHEST
10    elif dy > 0: # Ball below player
11        return Pitch.LOW
12        if dy < (height - paddle.bottom) / 2
13        else Pitch.LOWEST

```

(b) Calculate the **ball**'s current pitch

```

1 # Possible pitch values
2 class Pitch(IntEnum):
3     LOWEST = 0
4     LOW = 1
5     GOOD = 2
6     HIGH = 3
7     HIGHEST = 4

```

(c) Defines pitch values

Code Sample 4.16: Supplementary code for updating the **ball**

4.5 The Audification Layer

To begin, let us discuss our final audification concept:

1. The **ball** generates a repetitious beeping sound when on-screen.
2. The sound's pitch represents the distance between the **ball** and the **paddle** on the Y-axis.
3. The sound's volume, (as well as the speed between beeps) is proportional to the **ball**'s distance from the **paddle** on the X-axis.
4. The sound's spatial panning (its location 'around' the listener in 3D space) is defined by the ball's on-screen position. If the ball is near the left side of the screen, beeping sounds are heard on the player's left side in real-world space. If the ball is center-field, it is heard 'in front' of the player. If the ball is on the right side of the screen (near their opponent), it is heard on their right side.
5. When a ball is hit, a distinct sound effect is generated depending on which player hit the ball. The sound's spatial location is also dependent on the hitting player—left for player 1, and right for player 2.
6. In a similar vein, goals generate a unique sound for each player.
7. When the game is at 'match point,' a special sound is played during the **pause**.
8. When a player **wins** the game, a unique sound effect is played depending on the winner of the match.

These simple rules form the basis of our audio display. As you can clearly see, we have made a great number of reductions to *Pong*'s original design. And yet, our audio display gives clear, comprehensible, and reactive stimuli.

Rethinking *Pong* In essence, *Pong* could be re-conceived as a one dimensional task—wherein a player is attempting to 'dial-in' a pitch's frequency by turning a knob before a given deadline. If the pitch is high, the player must turn the knob to lower the pitch to an 'average' value. If the pitch is too low, the player must turn the knob the opposite direction to raise the pitch to the average value. The panning and speed of sound's repetition indicates how much time the player has left before their deadline approaches.

We can consequently conclude that it is possible to significantly re-shape games via their translation from visual to audio displays. Finding good audio solutions often requires rethinking the entire game from the ground-up, which may be non-trivial for complex titles. In general, we believe this to be a matter of **Design** which is outside the scope of this paper. Although we shall mediate on it more during our conclusion.

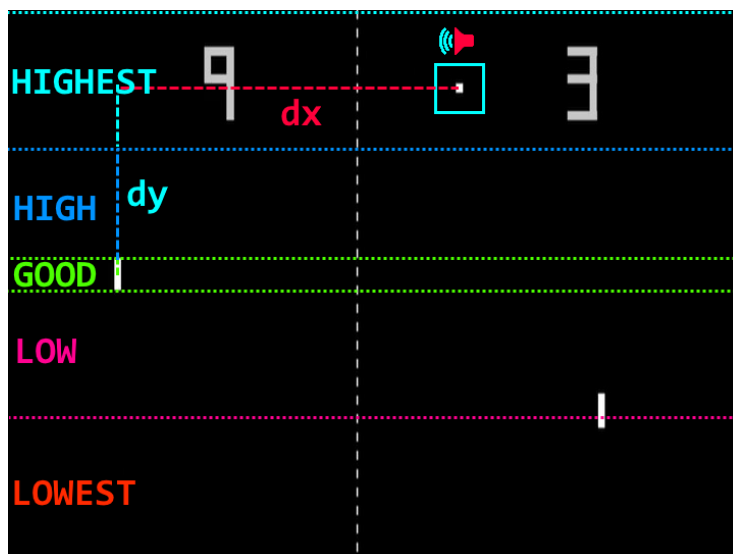


Figure 4.14: Visual of how the display's pitch calculation is made. Pitch regions are calculated relative to player one's current position on the *Y* axis—indicated by the blue speaker output. The speed of the sound is proportional to the ball's horizontal distance from the paddle—indicated here by the red line and speaker symbol.

4.5.1 Code Samples

The **Audiofication** layer is defined via a class of the same name in Code Sample 4.17 which manages the audio engine's configuration.

Using the engine, we playback spatial audio in the form of **AudioObjects**, which store audio and spatial location data. **AudioObjects** are spawned by **entities** in the **Decision** layer. Their spatial properties can be modified via function calls, as seen in Code Sample 4.19.

The `set_position` function can modify the **azimuth** or **az** (*around* the listener), **elevation** or **el** (*above* or *below* the listener), as well as the **distance** or **d** from the listener. These three parameters position the object spherically around the listener—which we describe via a **Quaternion** value. To emphasize spatiality, we must also *point* (i.e., orient or rotate) the object at the listener. This is because the object's sounds are extremely localized—meaning they are only audible within a small 'cone' originating at the object's 'center.' In other words, the listener will not be able to hear the object unless that object is pointed right at them.

To illustrate this phenomenon, imagine that our audio objects emitted light instead of sound. If the object operates like a flood light, it would diffuse light over a large span of space. A flashlight, meanwhile, provides moderate coverage over a circular area, while a laser pointer would only illuminate a tiny and

```

1 class AudificationLayer(QObject):
2
3     def __init__(parent):
4         super().__init__(parent)
5         client = parent
6         # Basic properties
7         _azimuth = 0
8         _elevation = 0
9         _distance = 0
10        _room_dimension = 1000
11        _reverb_gain = 0
12        _reflection_gain = 0
13
14        # Engine/Room Properties
15        _engine = QAudioEngine(self)
16        _engine.setOutputMode(QAudioEngine.Headphone)
17        _room = QAudioRoom(self._engine)
18        _room.setWallMaterial(QAudioRoom.BackWall,
19        QAudioRoom.Transparent)
20        _room.setWallMaterial(QAudioRoom.FrontWall,
21        QAudioRoom.Transparent)
22        _room.setWallMaterial(QAudioRoom.LeftWall,
23        QAudioRoom.Transparent)
24        _room.setWallMaterial(QAudioRoom.RightWall,
25        QAudioRoom.Transparent)
26        _room.setWallMaterial(QAudioRoom.Floor, QAudioRoom.
27        Transparent)
28        _room.setWallMaterial(QAudioRoom.Ceiling, QAudioRoom
29        .Transparent)
30        update_room()
31
32        _listener = QAudioListener(self._engine)
33        _listener.setPosition(QVector3D())
34        _listener.setRotation(QQuaternion())
35
36        _engine.start()

```

Code Sample 4.17: The **Audification** engine


```

1 class AudioObject(QObject):
2     def __init__(parent, sound=None):
3         super().__init__(parent)
4         # Need to cut across the dependency graph
5         # The curse of OOP
6         client = parent.parent().parent()
7
8         # Default location values
9         _azimuth = 0
10        _elevation = 0
11        _distance = 8000
12        _occlusion = 0
13
14        # Audio file paths
15        _paths = []
16
17        _sound = sound
18
19        # Force hard spatial directions
20        _sound.setDirectivity(1)
21        _sound.setDirectivityOrder(1000)
22
23        # set volume
24        _sound.setSize(1)

```

Code Sample 4.18: Definition of the AudioObject

```

1 def set_position(az=0, el=0, dist=100):
2     x = dist * math.sin(az) * math.cos(el)
3     y = dist * math.sin(el)
4     z = -dist * math.cos(az) * math.cos(el)
5     _sound.setPosition(QVector3D(x, y, z))
6     _sound.setRotation(
7         QQuaternion.fromDirection(_sound.position(),
8         QVector3D(0, 1, 0))
9     )

```

(a) Set object position using 3D spherical data

```

1 def set_position_simple(dx, dy):
2     az = (dx / 720) * math.pi - (math.pi / 2)
3     el = _elevation / 180.0 * math.pi
4     el = 0
5     d = _distance
6
7     x = d * math.sin(az) * math.cos(el)
8     y = d * math.sin(el)
9     z = -d * math.cos(az) * math.cos(el)
10    _sound.setPosition(QVector3D(x, y, z))
11    _sound.setRotation(
12        QQuaternion.fromDirection(_sound.position(),
13        QVector3D(0, 1, 0))
14    )

```

(b) Set object position using screen-space coordinates

```

1 def set_pan(pan):
2     if pan is Pan.LEFT:
3         set_position(az=-math.pi / 2)
4     elif pan is Pan.RIGHT:
5         set_position(az=math.pi / 2)
6     elif pan is Pan.CENTER:
7         set_position(az=0)

```

(c) Set hard stereo panning

Code Sample 4.19: Some AudioObject functions

focused spot. Returning to sound, we can imagine our spatial objects as *sonic lasers*—ones which direct sound at one specific point in space.

Of course, we can adjust the **directivity** and **directivity order** of our audio objects as we please. In the example above, we tuned our object to act like a laser with high directivity⁷:

```
1 _sound.setDirectivity(1) # MIN = 0, MAX = 1
2 _sound.setDirectivityOrder(1000) # MIN = 0, MAX = INF
```

This summarizes the AudificationLayer implementation. We use simple room metadata for our acoustic design—which is sufficient for our purposes, but may require additional fine-tuning for other games.

Limitations There are, however, a few limitations to our approach. The **QtMultimedia** we make use of is unfortunately still fairly nascent in development—meaning numerous key functions are yet-unimplemented. For example, the **QSpatialAudioEngine** is only capable of static file playback (pre-recorded files such as `.mp3` or `.wav`). While this is sufficient for most cases, we have no means of performing real-time audio manipulating. In other words, we cannot modify a sound’s pitch dynamically.

To sidestep this hurdle, we make use of several pre-recorded sounds—each of a different frequency. When it comes time to generate a sound, we decide which recording (i.e., frequency) to use based on the distance between the player and the ball. This can be seen in the **ball’s update** function in Code Sample 4.15.

This solution limits our frequency resolution (only five pitches as opposed to thousands), but serves as a fine starting-point for future implementations. We will likely re-write sections of the **QtMultimedia** module’s audio engine, provided the package doesn’t receive these features in a later update.

With this, we have successfully built the **Audification** layer using our **Design** semantics.

4.6 Execution: Building an Application

Finally, we must package everything into a useable application. Using Qt, we created a GUI with configurable options such as window settings—which selects which software window’s data is sent to the **Vision** layer. There are also some basic audio settings and a preview window used to verify window capture is working correctly. We also included debug features for development purposes—such as the computer vision algorithm’s data and some sliders for modifying the corner detection algorithm’s parameters.

This application is incredibly bare-bones, but still demonstrates the **Execution** stage clearly. We mediate on more advanced designs in later sections.

⁷We use an arbitrarily high value of 1000 for our directivity order, but any sufficiently large number will do.

4.7 Conclusion

In this chapter, we provided a in-depth look at a sample implementation of ReAL Sound. We began by considering the implementation process before deciding on a target game—*Pong*. We then constructed each **layer** of ReAL Sound and provided code samples to illustrate our process before finally considering some limitations of our current prototype.

Chapter 5

Conclusions

In this thesis, we proposed ReAL Sound: the **Re**-usable **A**udification **L**ibrary. ReAL Sound solves well known problems in the areas of game development, accessibility, and cross-platform software design. By using recent advances in AI, spatial audio, and computer vision, ReAL Sound is able to generalize the process of creating accessibility features for the visually impaired—lowering the barriers of access to games. ReAL Sound’s abstract and general purpose nature also allows a wide variety of implementation contexts—from in-house development to after-market fan modifications.

We began our thesis in chapter one by providing an overview recent trends in world of technology and games. We highlighted growing trends towards accessible, platform agnostic design. We also considered the particular problems faced by the gaming industry—where bespoke and platform specific tools are commonplace. Then, we considered the state of accessibility in games, noting several ‘problem areas’ in need of further research. Using this background, we justified our proposal for ReAL Sound.

In chapter two, we reviewed the relevant literature in the fields of computer vision, AI, audification, and games. Along the way, we examined relevant research in each field and renewed our understanding of key topics such as sound localization and object detection. We also considered the current state of game accessibility research—which features a scant few prior studies with goals similar to our own.

In chapter three we finally proposed ReAL Sound in full. We first began by exploring relevant concepts—where we crafted our own semi-formal definition of video games and considered how games are analyzed by their players. We then explained the structure of ReAL Sound—its **Vision**, **Decision**, and **Audification** layers—in clear detail. Afterwards, we considered the process of implementing ReAL Sound via the **Planning**, **Training**, **Design**, and **Execution** pipeline. We finally considered ReAL Sound from the end user’s perspective.

In chapter four we demonstrated ReAL Sound using our own sample implementation built in **python** using the common **OpenCV** and **Qt** libraries. Our implementation targeted the classic *Pong* using the well-known Harris corner de-

tection algorithm for image data analysis. We elaborated on the implementation process in great detail—considering challenges and solutions for each step as we progressed. We also proposed alternative implementations to further illuminate our understanding.

5.1 Limitations

Of course, the sweeping nature of a proposal such as ReAL Sound is far beyond what may be contained in a singular graduate thesis. Much work is needed to polish and refine the concepts presented within this work.

Although ReAL Sound could be added to most games, there are no doubt limits to the translation of visual stimuli into auditory displays. In reality, ReAL Sound would likely target simple games—generally ‘retro’ titles with simple rulesets equally primitive visuals. Despite this, we believe that ReAL Sound could still find utility as one of many tools used in aiding visually impaired players in more complex games—providing certain ‘hints’ that enhance their understanding of a given game.

5.2 Future Work

Prototype Improvement

Although our prototype functions on *Pong*, it is far from a fully featured, user-friendly software. Beyond improving the built-in audio display, we believe it would be wise to re-tool and generalize our process further in order to allow users the chance to customize their *own* audio displays. We believe this is where ReAL Sound will show its true strength via future work.

It would also be wise to experiment with other computer vision (particularly ML-based) algorithms. We considered a few other algorithms in chapter 4, but we unfortunately did not find the time to fit a full exploration of machine learning and AI within this thesis—although we believe the same process applies for creating the **Vision** layer.

Other Target Games

Pong’s simplicity is helpful for illustration purposes, but one could also argue that the same simplicity also undermines our intents. In order to showcase ReAL Sound’s versatility, we could be interested in seeing further implementations of ReAL Sound into other target games. We concede our system is likely unfitting for complex modern triple-A games, but we would still like to see exactly how far ReAL Sound could go before running into serious implementation hurdles. Regardless, we imagine ReAL Sound is suitable for many classic 70s and 80s games still lacking accessibility support. A sample of this concept is present in fig. 5.1 using *Space Invaders*.

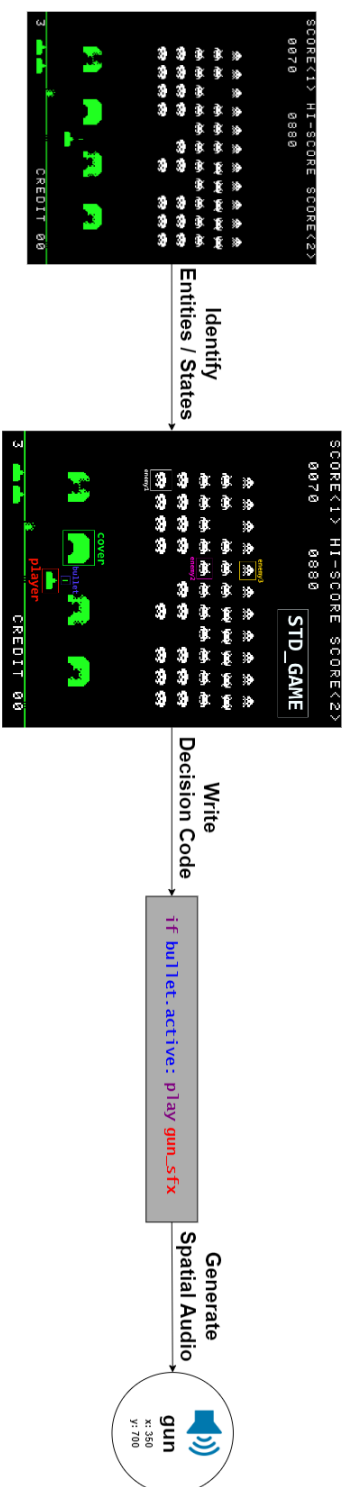


Figure 5.1: Conceptualizing the ReAL Sound implementation process for *Space Invaders*



Figure 5.2: A sample UI for designing **Decision** rules.

Further Generalization

As we hinted to in previous sections, we also believe that ReAL Sound could benefit from further generalization in order to improve inclusive design capabilities. Much like a game’s level creator allows ordinary players the ability to ‘develop’ their own games and levels, we believe that a well-constructed DSL and user-friendly software could further simplify the implementation process. A simple GUI as imagined in fig. 5.2 could greatly simplify the development and customization process. Combined with tools that automatically train ML models given simplified user input, it may be possible to turn ReAL Sound itself into an accessible development platform. We consider this concept to be a promising foundation for future doctoral work.

As the fields of machine learning and computer vision are changing constantly, it is easy to imagine a litany of future projects. Improving the **Vision** layer via high quality machine learning algorithms is one such example. Moreover, the creation of a general application with graphical tools that streamline the **Design** and **Planning** phases is also of interest to the author. On top of this, additional experimentation with more target games could provide reliable data on ReAL Sound efficacy. Finally, implementing ReAL Sound into 3D games could produce interesting results—allowing a better understanding on ReAL Sound’s limitations.

5.3 Thanks

I must extend a gracious thanks to my thesis advisor Hiroyuki Matsuguma for their constant support. I also would like to thank my supplemental advisors Tokushu Inamura and Dr. Yuki Morimoto, as well as Dr. Ho Hsin-Ni for their priceless feedback. Of course, I would be remiss if I didn’t explicitly dedicate this work to the late Kenji Eno—whose works *D*, *Real Sound*, and *Enemy Zero*

directly inspired this research, as well as my long and complex journey from a humble Oklahoma City to the great Kyushu University.

Bibliography

- [1] H. Abelson, G.J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs - 2nd Edition*. Justin Kelly. URL: <https://books.google.co.jp/books?id=MXZQAwAAQBAJ>.
- [2] Emanuele Agrimi et al. “Game accessibility for visually impaired people: a review”. In: *Soft Computing* (2024), pp. 1–15.
- [3] Luis von Ahn et al. “Improving accessibility of the web with a computer game”. In: CHI '06. Montréal, Québec, Canada: Association for Computing Machinery, 2006, pp. 79–82. ISBN: 1595933727. DOI: 10.1145/1124772.1124785. URL: <https://doi.org/10.1145/1124772.1124785>.
- [4] Ahoy. *The First Video Game*. Oct. 2019. URL: <https://www.youtube.com/watch?v=uHQ4WCU1WQc>.
- [5] Elliot Alexander. *How web apps took over the desktop*. en. Apr. 2024. URL: <https://www.xda-developers.com/how-web-apps-took-over-the-desktop/>.
- [6] Matthew T Atkinson et al. “Making the mainstream accessible: redefining the game”. In: *Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames*. 2006, pp. 21–28.
- [7] Matthew T. Atkinson and Colin Machin. “Proof-of-concept 3D level creation tool for blind gamers”. In: (Jan. 2009). URL: https://repository.lboro.ac.uk/articles/conference_contribution/Proof-of-concept_3D_level_creation_tool_for_blind_gamers/9405689.
- [8] Audiogames.net. *AudioGames — Frequently Asked Questions*. 2022.
- [9] Robert Bell et al. “The BellKor 2008 Solution to the Netflix Prize”. In: *AT&T Research* (Jan. 2008).
- [10] Marc Bigas et al. “Review of CMOS image sensors”. In: *Microelectronics journal* 37.5 (2006), pp. 433–451.
- [11] Jasmin Blanchette and Mark Summerfield. *C++ GUI Programming with Qt 4*. 2nd. USA: Prentice Hall PTR, 2008. ISBN: 0132354160.
- [12] G. Bradski. “The OpenCV Library”. In: *Dr. Dobb’s Journal of Software Tools* (2000).

- [13] Roberto Brunelli. *Template matching techniques in computer vision*. en. Hoboken, NJ: Wiley-Blackwell, Mar. 2009.
- [14] Erik Brynjolfsson and Andrew McAfee. “The Business of Artificial Intelligence”. In: *Harvard Business Review* (July 2017). ISSN: 0017-8012. URL: <https://hbr.org/2017/07/the-business-of-artificial-intelligence>.
- [15] Ian Campbell. *Dell’s new 4K QD-OLED monitor comes with spatial audio*. en-US. Jan. 2025. URL: <https://www.engadget.com/computing/accessories/dells-new-4k-qd-oled-monitor-comes-with-spatial-audio-194551957.html>.
- [16] John Carroll and Darrell Long. *Theory of Finite Automata with an Introduction to Formal Languages*. Jan. 1989. ISBN: 0-13-913708-4.
- [17] Anna Cavender, Shari Trewin, and Vicki Hanson. “General writing guidelines for technology and people with disabilities”. In: *SIGACCESS Access. Comput.* 92 (Sept. 2008), pp. 17–22. ISSN: 1558-2337. DOI: 10.1145/1452562.1452565. URL: <https://doi.org/10.1145/1452562.1452565>.
- [18] Chetvorno. *Turnstile state machine colored*. 2012. URL: https://commons.wikimedia.org/wiki/File:Turnstile_state_machine_colored.svg.
- [19] Noam Chomsky. “Three models for the description of language”. In: *IRE Trans. Inf. Theory* 2 (1956), pp. 113–124. URL: <https://api.semanticscholar.org/CorpusID:17432009>.
- [20] Alex Churchill, Stella Biderman, and Austin Herrick. *Magic: The Gathering is Turing Complete*. 2019. arXiv: 1904.09828 [cs.AI]. URL: <https://arxiv.org/abs/1904.09828>.
- [21] G Chursin and M Semenov. “Using computer vision in the gameplay of educational computer games”. In: *Journal of Physics: Conference Series*. Vol. 1989. 1. IOP Publishing. 2021, p. 012011.
- [22] Richard Cobbett. *From shareware superstars to the Steam gold rush: How indie games conquered the PC*. Sept. 2017. URL: <https://www.pcgamer.com/from-shareware-superstars-to-the-steam-gold-rush-how-indie-conquered-the-pc/>.
- [23] The Qt Company. *Qt Spatial Audio 6.8.1*. URL: <https://doc.qt.io/qt-6/qtspatialaudio-index.html>.
- [24] Rodrigo Copetti. *PlayStation 3 Architecture — A Practical Analysis*. Oct. 2021. URL: <https://www.copetti.org/writings/consoles/playstation-3/>.
- [25] Rodrigo Copetti. *Wii Architecture — A Practical Analysis*. en. Jan. 2020. URL: <https://www.copetti.org/writings/consoles/wii/>.
- [26] Microsoft Corporation. *PrintWindow function (winuser.h) - Windows API Documentation*. 2024. URL: <https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-printwindow>.

- [27] Greg Costikyan. *New Front in the Copyright Wars: Out-of-Print Computer Games*. May 2000. URL: <https://archive.nytimes.com/www.nytimes.com/library/tech/00/05/circuits/articles/18aban.html>.
- [28] Joe Cox. *Dolby Atmos is everywhere, and that's not necessarily a good thing*. en. June 2020. URL: <https://www.whathifi.com/features/dolby-atmos-is-everywhere-and-thats-not-necessarily-a-good-thing>.
- [29] Ivan Culjak et al. "A brief introduction to OpenCV". In: *2012 proceedings of the 35th international convention MIPRO*. IEEE. 2012, pp. 1725–1730.
- [30] Angel De La Cruz and John Ryan. "Tennis for Two". In: (2015).
- [31] Erik D. Demaine and Robert A. Hearn. "Constraint Logic: A Uniform Framework for Modeling Computation as Games". In: *2008 23rd Annual IEEE Conference on Computational Complexity*. 2008, pp. 149–162. DOI: 10.1109/CCC.2008.35.
- [32] Denon. *Connecting 11.1-channel speakers*. 2022. URL: <https://manuals.denon.com/AVRX6400H/eu/en/DRDZSYrjomlmpo.php>.
- [33] Christopher Dewey, Austin Moore, and Hyunkook Lee. "Practitioners' Perspectives on Spatial Audio: Insights into Dolby Atmos and Binaural Mixes in Popular Music". In: *AES: Journal of the Audio Engineering Society* 72.7/8 (2024), pp. 504–516.
- [34] J.E. Dobson. *The Birth of Computer Vision*. University of Minnesota Press, 2023. ISBN: 9781452968872. URL: <https://books.google.co.jp/books?id=LKyTEAAQBAJ>.
- [35] AD Dongare, RR Kharde, Amit D Kachare, et al. "Introduction to artificial neural network". In: *International Journal of Engineering and Innovative Technology (IJEIT)* 2.1 (2012), pp. 189–194.
- [36] Hideo Eguchi. "WEA Q Disks to Activate Other Acts & Labels". In: *Billboard* (Aug. 1973). URL: <https://www.worldradiohistory.com/Archive-All-Music/Billboard/70s/1973/Billboard%201973-08-04.pdf>.
- [37] Christopher Erdelyi. "Using Computer Vision Techniques to Play an Existing Video Game". In: (2019).
- [38] Lu Fang, Oscar C. Au, and Ngai-Man Cheung. "Subpixel rendering: from font rendering to image subsampling [Applications Corner]". In: *IEEE Signal Processing Magazine* 30.3 (2013), pp. 177–189. DOI: 10.1109/MSP.2013.2241311.
- [39] Fig02. *Arbitrary Code Execution in Ocarina of Time*. Youtube. 2019. URL: <https://www.youtube.com/watch?v=RoEmGCNsbn0>.
- [40] Gerhard Fischer and Eric Scharff. "Meta-design: design for designers". In: *Proceedings of the 3rd conference on Designing interactive systems: processes, practices, methods, and techniques*. 2000, pp. 396–405.

- [41] Mary Jo Foley. *Microsoft acquires Havok from Intel*. Oct. 2015. URL: <https://www.zdnet.com/article/microsoft-acquires-havok-from-intel/>.
- [42] Mary Jo Foley. *Microsoft takes a step toward phasing out 32-bit PC support for Windows 10*. en. May 2020. URL: <https://www.zdnet.com/article/microsoft-takes-a-step-toward-phasing-out-32-bit-pc-support-for-windows-10/>.
- [43] X.Org Foundation and Christophe Tronche. *XGetImage*. URL: <https://tronche.com/gui/x/xlib/graphics/XGetImage.html>.
- [44] Johnny Friberg and Dan Gärdenfors. “Audio games: new perspectives on game audio”. In: *Proceedings of the 2004 ACM SIGCHI International Conference on Advances in computer entertainment technology*. 2004, pp. 148–154.
- [45] L. S. G. Kovasznay and H. M. Joseph. “Image Processing”. In: *Proceedings of the IRE* 43.5 (1955), pp. 560–570. DOI: 10.1109/JRPR0C.1955.278100.
- [46] Danny Gallagher. *AI learns how to play Super Mario World better than you ever will*. June 2015. URL: <https://www.cnet.com/science/ai-learns-how-to-play-super-mario-world-better-than-you-ever-will/>.
- [47] GMA Games. *Shades of Doom*. URL: <https://www.gmagames.com/sod.shtml>.
- [48] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Pearson Education, 1994. ISBN: 9780321700698. URL: <https://books.google.com.tw/books?id=6oHuKQe3TjQC>.
- [49] Lee Garber. “Game accessibility: enabling everyone to play”. In: *Computer* 46.06 (2013), pp. 14–18.
- [50] Dave Gershgorin. *The data that transformed AI research—and possibly the world*. en. July 2017. URL: <https://qz.com/1034972/the-data-that-changed-the-direction-of-ai-research-and-possibly-the-world>.
- [51] Glosser.ca. *Colored neural network*. 2010. URL: https://commons.wikimedia.org/wiki/File:Colored_neural_network.svg.
- [52] Jason Gregory. *Game engine architecture*. AK Peters/CRC Press, 2018.
- [53] Information Solutions Group. *Survey: ‘Disabled Gamers’ Comprise 20% of Casual Video Games Audience*. June 2008. URL: <https://web.archive.org/web/20170615144453/http://www.prnewswire.com/news-releases/survey-disabled-gamers-comprise-20-of-casual-video-games-audience-57442172.html>.

- [54] Furkan Gursoy and Ioannis A. Kakadiaris. “Artificial intelligence research strategy of the United States: critical assessment and policy recommendations”. In: *Frontiers in Big Data* 6 (2023). ISSN: 2624-909X. DOI: 10.3389/fdata.2023.1206139. URL: <https://www.frontiersin.org/journals/big-data/articles/10.3389/fdata.2023.1206139>.
- [55] C. Harris and M. Stephens. “A Combined Corner and Edge Detector”. In: *Proc. AVC*. doi:10.5244/C.2.23. 1988, pp. 23.1–23.6.
- [56] M Hashimoto and J Sklansky. “Multiple-order derivatives for detecting local image characteristics”. In: *Computer Vision, Graphics, and Image Processing* 39.1 (1987), pp. 28–55.
- [57] Mark Havryliv and Terumi Narushima. “Metris: A Game Environment for Music Performance”. In: *Computer Music Modeling and Retrieval*. Ed. by Richard Kronland-Martinet, Thierry Voinier, and Sølvi Ystad. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 101–109. ISBN: 978-3-540-34028-7.
- [58] Melissa Heikkilä. *Inside a radical new project to democratize AI*. en. July 2022. URL: <https://www.technologyreview.com/2022/07/12/1055817/inside-a-radical-new-project-to-democratize-ai/>.
- [59] Seth Henderson. *MarI/O - Machine Learning for Video Games*. Youtube. 2015. URL: <https://www.youtube.com/watch?v=qv6UV0QOF44>.
- [60] Seth Hendrickson. *MarI/O Source Code*. June 2015. URL: <https://gist.github.com/SethBling/598639f8d5e8afb5453a0b9519be51ff>.
- [61] Thomas S. Huang. “Computer Vision: Evolution And Promise”. In: 1996. URL: <https://api.semanticscholar.org/CorpusID:67163270>.
- [62] Apple Inc. *Airpods: Control Spatial Audio and head tracking*. en. URL: <https://support.apple.com/guide/airpods/control-spatial-audio-and-head-tracking-dev00eb7e0a3/web>.
- [63] JonMcLoone. *Edge Detection*. 2010. URL: https://commons.wikimedia.org/wiki/File:%C3%84%C3%A4retuvastuse_n%C3%A4ide.png.
- [64] Michael I Jordan and Tom M Mitchell. “Machine learning: Trends, perspectives, and prospects”. In: *Science* 349.6245 (2015), pp. 255–260.
- [65] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. “Reinforcement learning: A survey”. In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.
- [66] Will Knight. “Google’s Gemini Is the Real Start of the Generative AI Boom”. en-US. In: *Wired* (Dec. 2023). ISSN: 1059-1028. URL: <https://www.wired.com/story/google-gemini-generative-ai-boom/>.
- [67] Joseph Knoop. *Epic v Apple judge Grapples with the big question: What is a videogame?* Sept. 2021. URL: <https://www.pcgamer.com/videogame-definition-legal/>.
- [68] Stefan Kolb. *On the Portability of Applications in Platform as a Service*. Vol. 34. University of Bamberg Press, 2019.

- [69] G. Kramer and EDITOR *. *Auditory Display: Sonification, Audification, And Auditory Interfaces*. Avalon Publishing, 1994. ISBN: 9780201626032. URL: <https://books.google.co.jp/books?id=WF2qQgAACAAJ>.
- [70] Gregory Kramer et al. “Sonification report: Status of the field and research agenda”. In: (2010).
- [71] Ben Kuchera. *The forgotten, beloved 60GB PS3: why it’s still so popular*. Sept. 2009. URL: <https://arstechnica.com/gaming/2009/09/the-forgotten-beloved-60gb-ps3-why-its-still-so-popular/>.
- [72] Dolby Laboratories. *Dolby Atmos for the Home Theater*. English. Oct. 2016, p. 15. URL: <https://web.archive.org/web/20200521122348/https://www.dolby.com/us/en/technologies/dolby-atmos/dolby-atmos-for-the-home-theater.pdf>.
- [73] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *nature* 521.7553 (2015), pp. 436–444.
- [74] Kelsey Lewin. *87% missing: The disappearance of classic video games*. Nov. 2023. URL: <https://gamehistory.org/87percent/>.
- [75] Song Li and Jürgen Peissig. “Measurement of head-related transfer functions: A review”. In: *Applied Sciences* 10.14 (2020), p. 5014.
- [76] Marco Liboà. “Hardware design and representation of graphics in videogames: A case study: the Sega Saturn”. In: *Transactions of the Digital Games Research Association* 5.1 (2020).
- [77] Google LLC. *Spatial Audio*. en. 2024. URL: <https://developers.google.com/vr/ios/spatial-audio>.
- [78] S. Lloyd. “Least squares quantization in PCM”. In: *IEEE Transactions on Information Theory* 28.2 (1982), pp. 129–137. DOI: 10.1109/TIT.1982.1056489.
- [79] Chris Lomont. “Fast inverse square root”. In: *Tech-315 nical Report* 32 (2003).
- [80] Julia Love. *AI Researcher Who Helped Write Landmark Paper Is Leaving Google*. en-US. July 2023. URL: <https://finance.yahoo.com/news/ai-researcher-helped-write-landmark-030025546.html>.
- [81] The Qt Company Ltd. *PySide6.QtMultimedia.QScreenCapture*. URL: <https://doc.qt.io/qtforpython-6/PySide6/QtMultimedia/QScreenCapture.html>.
- [82] The Qt Company Ltd. *Screen Capture Example*. URL: https://doc.qt.io/qtforpython-6/examples/example_multimedia_screencapture.html.

- [83] Shan Luo, Jianan Johanna Liu, and Botao Amber Hu. “Designing a Safe Auditory-Cued Archery Exertion Game for the Visually Impaired and Sighted to Enjoy Together”. In: *Proceedings of the 26th International ACM SIGACCESS Conference on Computers and Accessibility*. ASSETS '24. St. John's, NL, Canada: Association for Computing Machinery, 2024. ISBN: 9798400706776. DOI: 10.1145/3663548.3688510. URL: <https://doi.org/10.1145/3663548.3688510>.
- [84] John Maeda. *CX Report*. 2021.
- [85] Batta Mahesh. “Machine learning algorithms-a review”. In: *International Journal of Science and Research (IJSR)*. [Internet] 9.1 (2020), pp. 381–386.
- [86] André Marion. *Introduction to image processing*. Springer, 2013.
- [87] D. Marr and E. Hildreth. “Theory of Edge Detection”. In: *Proceedings of the Royal Society of London. Series B. Biological Sciences* 207.1167 (Feb. 1980), pp. 187–217. ISSN: 2053-9193. DOI: 10.1098/rspb.1980.0020. URL: <http://dx.doi.org/10.1098/rspb.1980.0020>.
- [88] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: <https://www.tensorflow.org/>.
- [89] Masaki Matsuo et al. “Inclusive Fighting with Mind’s Eye: Case Study of a Fighting Game Playing with only Auditory Cues for Sighted and Blind Gamers”. In: *Computers Helping People with Special Needs*. Ed. by Klaus Miesenberger, Petr Peňáz, and Makoto Kobayashi. Cham: Springer Nature Switzerland, 2024, pp. 137–145. ISBN: 978-3-031-62846-7.
- [90] Jasna Maver. “Self-similarity and points of interest”. In: *IEEE transactions on pattern analysis and machine intelligence* 32.7 (2009), pp. 1211–1226.
- [91] Ross Mawhorter et al. “Content Reinjection for Super Metroid”. In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 17.1 (Oct. 2021), pp. 172–178. DOI: 10.1609/aiide.v17i1.18905. URL: <https://ojs.aaai.org/index.php/AIIDE/article/view/18905>.
- [92] Leo McCormack et al. *leomccormack/Spatial_Audio_Framework*. June 6, 2024. URL: https://github.com/leomccormack/Spatial%5C_Audio%5C_Framework.
- [93] David K. McGookin and Stephen A. Brewster. “Understanding concurrent earcons: Applying auditory scene analysis principles to concurrent earcon recognition”. In: *ACM Trans. Appl. Percept.* 1.2 (Oct. 2004), pp. 130–155. ISSN: 1544-3558. DOI: 10.1145/1024083.1024087. URL: <https://doi.org/10.1145/1024083.1024087>.

- [94] Sam Meredith. *A ‘thirsty’ generative AI boom poses a growing problem for Big Tech*. en. Dec. 2023. URL: <https://www.cnbc.com/2023/12/06/water-why-a-thirsty-generative-ai-boom-poses-a-problem-for-big-tech.html>.
- [95] Marjan Mernik, Jan Heering, and A.M. Sloane. “When and how to develop domain-specific languages”. Jan. 2005.
- [96] Inc. Meta Platforms. *Meta XR Audio SDK for Unreal - Ambisonics*. 2024. URL: <https://developers.meta.com/horizon/documentation/unreal/meta-xr-audio-sdk-unreal-ambisonic/>.
- [97] John C Middlebrooks. “Sound localization”. In: *Handbook of clinical neurology* 129 (2015), pp. 99–116.
- [98] Klaus Miesenberger et al. “More than just a game: accessibility in computer games”. In: *HCI and Usability for Education and Work: 4th Symposium of the Workgroup Human-Computer Interaction and Usability Engineering of the Austrian Computer Society, USAB 2008, Graz, Austria, November 20-21, 2008. Proceedings 4*. Springer. 2008, pp. 247–260.
- [99] Daniel Miller, Aaron Parecki, and Sarah A Douglas. “Finger dance: a sound game for blind people”. In: *Proceedings of the 9th International ACM SIGACCESS Conference on Computers and Accessibility*. 2007, pp. 253–254.
- [100] George A Miller. “The cognitive revolution: a historical perspective”. In: *Trends in Cognitive Sciences* 7.3 (2003), pp. 141–144. ISSN: 1364-6613. DOI: [https://doi.org/10.1016/S1364-6613\(03\)00029-9](https://doi.org/10.1016/S1364-6613(03)00029-9). URL: <https://www.sciencedirect.com/science/article/pii/S1364661303000299>.
- [101] Shervin Minaee et al. “Deep learning-based text classification: a comprehensive review”. In: *ACM computing surveys (CSUR)* 54.3 (2021), pp. 1–40.
- [102] Henrik Møller et al. “Head-related transfer functions of human subjects”. In: *Journal of the Audio Engineering Society* 43.5 (1995), pp. 300–321.
- [103] MTheiler. *Detected-with-YOLO*. 2010. URL: <https://commons.wikimedia.org/wiki/File:Detected-with-YOLO--Schreibtisch-mit-Objekten.jpg>.
- [104] D. Nasaw. *Going Out: The Rise and Fall of Public Amusements*. Harvard University Press, 1999. ISBN: 9780674417595. URL: <https://books.google.co.jp/books?id=hSEsEAAAQBAJ>.
- [105] Michael Z. Newman. *Atari Age*. en. MIT Press, 2017. ISBN: 9780262035712. URL: https://books.google.com/books/about/Atari_Age.html?hl=&id=Y200DgAAQBAJ.
- [106] Randolph J Nudo. “Plasticity”. In: *NeuroRx* 3 (2006), pp. 420–427.
- [107] G. O’Regan. *Pillars of Computing: A Compendium of Select, Pivotal Technology Firms*. Springer International Publishing, 2015. ISBN: 9783319214641. URL: <https://books.google.co.jp/books?id=RBKcCgAAQBAJ>.

- [108] OMG. *OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1*. Object Management Group, Aug. 2011. URL: <http://www.omg.org/spec/UML/2.4.1>.
- [109] OpenCV. *Harris Corner Detection*. URL: https://docs.opencv.org/4.x/dc/d0d/tutorial_py_features_harris.html.
- [110] OpenCV. *Harris Corner Detection*. URL: https://docs.opencv.org/4.x/dc/d0d/tutorial_py_features_harris.html.
- [111] OpenCV. *Image Thresholding*. URL: https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html.
- [112] World Health Organization. *International Classification of Impairments, Disabilities, and Handicaps: A Manual of Classification Relating to the Consequences of Disease*. World Health Organization, 1980. ISBN: 9789241541268. URL: <https://books.google.co.jp/books?id=QFNrAAAAAAAJ>.
- [113] Ciprian Paduraru, Miruna Paduraru, and Alin Stefanescu. “Automated game testing using computer vision methods”. In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. IEEE. 2021, pp. 65–72.
- [114] Eric Paquin. *I know I’ve spelled this right!* Aug. 2010. URL: <https://web.archive.org/web/20100817065302/http://www.officeformac.com/ms/blogs/blog1/I-know-I-ve-spelled-this-right>.
- [115] Lo-Fi People. *Blind Drive*. URL: <https://blinddrivegame.com/press/>.
- [116] Petibub. *SimpleFreeFieldHRIR-0.3*. 2010. URL: <https://commons.wikimedia.org/wiki/File:SimpleFreeFieldHRIR-0.3.png>.
- [117] M.M.P. Petrou and C. Petrou. *Image Processing: The Fundamentals*. Wiley, 2010. ISBN: 9780470745861. URL: <https://books.google.co.jp/books?id=w3BpSIxN9ZYC>.
- [118] Edwin Pfanzagl-Cardone. “The DOLBY® “Atmos™” System”. In: *The Art and Science of 3D Audio Recording*. Cham: Springer International Publishing, 2023, pp. 143–188. ISBN: 978-3-031-23046-2. DOI: 10.1007/978-3-031-23046-2_4. URL: https://doi.org/10.1007/978-3-031-23046-2_4.
- [119] John R Porter and Julie A Kientz. “An empirical study of issues and barriers to mainstream video game accessibility”. In: *Proceedings of the 15th international ACM SIGACCESS conference on computers and accessibility*. 2013, pp. 1–8.
- [120] Rachel Potvin and Josh Levenberg. “Why Google stores billions of lines of code in a single repository”. In: *Commun. ACM* 59.7 (June 2016), pp. 78–87. ISSN: 0001-0782. DOI: 10.1145/2854146. URL: <https://doi.org/10.1145/2854146>.
- [121] A Prazaru et al. “Overview on visually impaired gamers and game accessibility”. In: *EDULEARN20 Proceedings*. IATED. 2020, pp. 5491–5501.

- [122] The Chromium Projects. *Kernel Design*. Wiki. URL: <https://www.chromium.org/chromium-os/chromiumos-design-docs/chromium-os-kernel/>.
- [123] Weichao Qiu and Alan Yuille. “Unrealcv: Connecting computer vision to unreal engine”. In: *Computer Vision–ECCV 2016 Workshops: Amsterdam, The Netherlands, October 8–10 and 15–16, 2016, Proceedings, Part III 14*. Springer. 2016, pp. 909–916.
- [124] M. O. Rabin and D. Scott. “Finite Automata and Their Decision Problems”. In: *IBM Journal of Research and Development* 3.2 (1959), pp. 114–125. DOI: 10.1147/rd.32.0114.
- [125] J. Raessens and J. Goldstein. *Handbook of Computer Game Studies*. MIT Press, 2011. ISBN: 9780262516587. URL: <https://books.google.co.jp/books?id=ErhNEAAQBAJ>.
- [126] Eric S. Raymond. *The Art of UNIX Programming*. Pearson Education, 2003. ISBN: 0131429019.
- [127] CB Insights Research. *The Two Engines Driving the \$120B Gaming Industry Forward*. Sept. 2018. URL: <https://www.cbinsights.com/research/game-engines-growth-expert-intelligence/>.
- [128] Mark Richert. *FCC Pushes Back on Gaming Industry Accessibility Waiver Request, Consumer Voices Tip the Scales*. URL: <https://www.afb.org/aw/13/12/15859>.
- [129] Stephan R Richter et al. “Playing for data: Ground truth from computer games”. In: *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part II 14*. Springer. 2016, pp. 102–118.
- [130] Roboflow. *Supervision*. URL: <https://github.com/roboflow/supervision>.
- [131] Subhasis Saha. “Image compression—from DCT to wavelets: a review”. In: *XRDS: Crossroads, The ACM Magazine for Students* 6.3 (2000), pp. 12–21.
- [132] Miro Samek. “Who Moved My State?” In: *The C/C++ Users Journal* 21 (Apr. 2003), pp. 28+30–34.
- [133] Anthony Savidis and Constantine Stephanidis. “Inclusive development: Software engineering requirements for universally accessible interactions”. In: *Interacting with Computers* 18.1 (2006), pp. 71–116.
- [134] Michael Schoeffler, Andreas Silzle, and Jürgen Herre. “Evaluation of Spatial/3D Audio: Basic Audio Quality Versus Quality of Experience”. In: *IEEE Journal of Selected Topics in Signal Processing* 11.1 (2017), pp. 75–88. DOI: 10.1109/JSTSP.2016.2639325.
- [135] Alireza Shafaei, James J Little, and Mark Schmidt. “Play and learn: Using video games to train computer vision models”. In: *arXiv preprint arXiv:1608.01745* (2016).

- [136] Brendan Sinclair. *Square Enix signs long-term Unreal Engine deal*. Oct. 2012. URL: <https://www.gamesindustry.biz/square-enix-signs-long-term-unreal-engine-deal>.
- [137] Piotr Skalski. *Make Sense*. <https://github.com/SkalskiP/make-sense/>. 2019.
- [138] Alexander Smith. *They Create Worlds: The Story of the People and Companies that Shaped the Video Game Industry*. C&C Press, Nov. 2019. ISBN: 9780429423642. DOI: 10.1201/9780429423642.
- [139] Valve Software. *Steam Audio*. URL: <https://valvesoftware.github.io/steam-audio/>.
- [140] Myung-Suk Song et al. “Enhancing loudspeaker-based 3D audio with room modeling”. In: *2010 IEEE International Workshop on Multimedia Signal Processing*. 2010, pp. 34–39. DOI: 10.1109/MMSP.2010.5661990.
- [141] Aaron Souppouris. *Artificial intelligence learns Mario level in just 34 attempts*. en-US. June 2015. URL: <https://www.engadget.com/2015-06-17-super-mario-world-self-learning-ai.html>.
- [142] Jeanne F Spellman et al. *W3C Accessibility Guidelines (WCAG) 3.0*. W3C Working Draft. <https://www.w3.org/TR/2024/WD-wcag-3.0-20241212/>. W3C, Dec. 2024.
- [143] Kenneth O. Stanley and Risto Miikkulainen. “Evolving Neural Networks through Augmenting Topologies”. In: *Evolutionary Computation* 10.2 (2002), pp. 99–127. DOI: 10.1162/106365602320169811.
- [144] Joshua Strang. *Image Segmentation Example: Aurora Borealis*. This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by/4.0/>. 2012. URL: https://commons.wikimedia.org/wiki/Image_segmentation#Example:_Aurora_Borealis.
- [145] Ricardo Torres. *GameSpot Presents: GameSpotting*. Sept. 2005. URL: https://web.archive.org/web/20050907192055/http://www.gamespot.com/gamespot/features/all/gamespotting/081102/p6_01.html.
- [146] A. M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265. DOI: <https://doi.org/10.1112/plms/s2-42.1.230>. eprint: <https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/plms/s2-42.1.230>. URL: <https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/plms/s2-42.1.230>.
- [147] Alan Turing. *The Essential Turing*. en. Ed. by B. J. Copeland. Oxford, England: Clarendon Press, 2004. ISBN: 9780198250807.

- [148] Michael Urbanek and Florian Güldenpfennig. “Unpacking the audio game experience: Lessons learned from game veterans”. In: *Proceedings of the annual symposium on computer-human interaction in play*. 2019, pp. 253–264.
- [149] José Ángel Vallejo-Pinto et al. “Applying sonification to improve accessibility of point-and-click computer games for people with limited vision”. In: *Proceedings of HCI 2011 The 25th BCS Conference on Human Computer Interaction*. BCS Learning & Development. 2011.
- [150] Antti J. Vanne et al. “Spatial Audio Reproduction Based on Head-to-Torso Orientation”. 20240357308. Oct. 2024.
- [151] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: <https://arxiv.org/abs/1706.03762>.
- [152] Paul Vickers and Robert Höldrich. *Direct Segmented Sonification of Characteristic Features of the Data Domain*. 2017. arXiv: 1711.11368 [cs.HC]. URL: <https://arxiv.org/abs/1711.11368>.
- [153] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. *YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors*. 2022. arXiv: 2207.02696 [cs.CV]. URL: <https://arxiv.org/abs/2207.02696>.
- [154] Xiaogang Wang et al. “Deep learning in object recognition, detection, and segmentation”. In: *Foundations and Trends® in Signal Processing* 8.4 (2016), pp. 217–382.
- [155] Tom Warren. *Nokia wants Instagram for Windows Phone, piles pressure on with #2InstaWithLove*. en. Mar. 2013. URL: <https://www.theverge.com/2013/3/6/4070934/windows-phone-instagram-app-nokia-pressure>.
- [156] Janete Weinstein. *iPad gets a calculator app after 14 years*. en-US. June 2024. URL: <https://www.nbcbayarea.com/news/national-international/ipad-gets-a-calculator-app-after-14-years/3562483/>.
- [157] Joseph Weizenbaum. “ELIZA—a computer program for the study of natural language communication between man and machine”. In: *Commun. ACM* 9.1 (Jan. 1966), pp. 36–45. ISSN: 0001-0782. DOI: 10.1145/365153.365168. URL: <https://doi.org/10.1145/365153.365168>.
- [158] Weston.pace. *K Means Example*. This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by/4.0/>. 2007. URL: https://commons.wikimedia.org/wiki/File:K_Means_Example_Step_1.svg.
- [159] Kyle Wiggers. *Inside BigScience, the quest to build a powerful open language model*. en-US. Jan. 2022. URL: <https://venturebeat.com/ai/inside-bigscience-the-quest-to-build-a-powerful-open-language-model/>.

- [160] Robert Sessions Woodworth and Harold Schlosberg. *Experimental psychology*. Oxford and IBH Publishing, 1954.
- [161] Rishi Yadav. *Why Are We Still Having ‘Works on My Machine’ Problems?* en-US. Apr. 2021. URL: <https://thenewstack.io/why-are-we-still-having-works-on-my-machine-problems/>.
- [162] Ying Yu et al. “Techniques and challenges of image segmentation: A review”. In: *Electronics* 12.5 (2023), p. 1199.
- [163] Bei Yuan and Eelke Folmer. “Blind hero: enabling guitar hero for the visually impaired”. In: *Proceedings of the 10th international ACM SIGACCESS conference on Computers and accessibility*. 2008, pp. 169–176.
- [164] Bei Yuan, Eelke Folmer, and Frederick C Harris. “Game accessibility: a survey”. In: *Universal Access in the information Society* 10 (2011), pp. 81–100.
- [165] Zhong-Qiu Zhao et al. *Object Detection with Deep Learning: A Review*. 2019. arXiv: 1807.05511 [cs.CV]. URL: <https://arxiv.org/abs/1807.05511>.
- [166] Djemel Ziou and Salvatore Tabbone. “Edge detection techniques-an overview”. In: *Pattern Recognition and Image Analysis: Advances in Mathematical Theory and Applications* 8.4 (1998), pp. 537–559.