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

Meir Arani
Kyushu University
Graduate School of Design

January 12, 2025

Abstract

In a world of ever-increasing software complexity, there has been a growing demand for interoperable, reusable technologies that function in many problem domains. This is especially true in the world of game development, where tools, structures, and architectures often change from title to title. At the same time, the specificity of software user needs has also grown immensely, bringing an increased demand for advanced accessibility tools with it. To address these trends, we propose ReAL Sound: the ReUsable Audification Library, which abstracts the creation of visual accessibility technology for impaired persons in the realm of game design using computer vision and machine learning techniques.

Contents

Chapter 1

Introduction

1.1 Recent Advances in Software Development

1.2 Recent User Experience Trends

1.3 Games and Accessibility

Chapter 2

Literature Review

2.1 Computer Vision

2.2 AI & Machine Learning

2.3 Audification

2.3.1 Spatial Audio

2.4 Games

2.4.1 Games and Accessibility

2.4.2 Games and Audification

2.4.3 Games and Computer Vision

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.

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 [21]. This fact effectively renders the vast majority of published games, *'abandonware'* with little hope of official improvements by original developers [12].

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[3], 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[19][26] has brought an equally intense focus to the development, improvement, and democratization of AI tools and systems[5][17]. Ecosystems like *HuggingFace*[43] and projects like Google's *TensorFlow*[23] 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*[4] and *corner detection*[22]—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*[16], 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*[18] to even computer display monitors from major manufacturers like Dell[6]. Future patents promise even greater advancements through advanced techniques like automatic HRTF adjustments specific to the user's unique physiology[41].

Many new software development libraries been developed to address this boom in consumer demand. Fan driven efforts like the *Spatial_Audio_Framework*[25] as well as company-produced projects such as Valve Software's Steam Audio[37] have seen popular adoption. Legacy libraries with widespread adoption, such as the quarter-century old *Qt* application development framework[2], have also introduced support for modern spatial audio technologies in recent versions[10]. 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!'* onscreen.

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*[9] or *Minecraft*[22] out of scope[13]. Virtually any game can also be made Turning complete if methods of arbitrary code execution (ACE) are discovered, which has been demonstrated using *Super Metroid*[24], *The Legend of Zelda: Ocarina of Time*[14], 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[20]—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.

Automata Theory

The previous section's underlying theory will likely seem familiar for those with a background in fields such at 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 Turning machine, to Type-3, which describes finite state automata [8]. 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.[7]" 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 Turning machine[39], although there are some distinct differences. A Turning 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[40]. 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 Turning machine[7].

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[31]. 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[34]. Thankfully, numerous alternative notations have arisen to address these shortcomings.

For example, FSMs which produce output contingent on a given input—often termed *transducers*[7]—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[30]. 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 cases, 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[42].

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 much 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[35] or the corporate projects of companies such as *RoboFlow*[33]—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 irregardless 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 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 hardware[1].

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.

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 (*frame data*)

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.

[1] See `PrintWindow` in the Windows API[11] and `XGetImage` in the X11 API[15] for modern examples.

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)* $\leq$ *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 X [TODO: ADD!] summarizes the entire structure 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 it 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 intended 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 [29]. These EAMs often lined the walls of popular leisure spots—penny arcades, nickelodeons, amusement parks, etc [28]. 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[36].

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 Eno[2], 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 encodings[3] are often the 'problem spots' that render games inaccessible for impaired persons.

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

[3]Specifically 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.

[4]The 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. Figure 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

```
if(mario is not None and mushroom is not None):
    if(dist(mario, mushroom) < COLLISION_DIST):
        play_powerup_sfx()
```

```
if(mario is not None and pitfall is not None):
    if(dist(mario, pitfall < WARN_DIST)):
        play_pitfall_warning_sfx(dist(mario, pitfall))
```

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

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

used to trigger the sound effect, `play_pitfall_warning_sfx()` is dynamically 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[27]. 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[32].

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

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

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

Figure 3.2: Three samples of a hypothetical DSL used to **Design** a *Super Mario Brothers* implementation

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. Figure 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.

The DSL has a grammar accepting two command stings:

```
ENTITY VERB ENTITY: VERB [ENTITY | ATTRIBUTE] [SFX_NAME];
```

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

Those unfamiliar with formal language theory or metacompilers may struggle with the syntax. Suffice it to say that this DSL streamlines the **Design** 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 glace. 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[38]) 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

4.1 Languages, Tools, and Frameworks

4.1.1 Python

4.1.2 OpenCV

4.1.3 Qt

4.2 Choosing a Target Game

4.3 Pong: A Worked Demonstration

Chapter 5

Experiments

5.1 Pong Demonstration

5.1.1 Results

Chapter 6

Conclusions

6.1 Limitations

6.2 Future Work

6.3 Thanks

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] Jasmin Blanchette and Mark Summerfield. *C++ GUI Programming with Qt 4*. 2nd. USA: Prentice Hall PTR, 2008. ISBN: 0132354160.

[3] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000).

[4] Roberto Brunelli. *Template matching techniques in computer vision*. en. Hoboken, NJ: Wiley-Blackwell, Mar. 2009.

[5] 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.

[6] 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.

[7] John Carroll and Darrell Long. *Theory of Finite Automata with an Introduction to Formal Languages*. Jan. 1989. ISBN: 0-13-913708-4.

[8] 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.

[9] 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.

[10] The Qt Company. *Qt Spatial Audio 6.8.1*. URL: https://doc.qt.io/qt-6/qtspatialaudio-index.html.

[11] Microsoft Corporation. *PrintWindow function (winuser.h) - Windows API Documentation*. 2024. URL: https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-printwindow.

[12] 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.

[13] 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.

[14] Fig02. *Arbitrary Code Execution in Ocarina of Time*. Youtube. 2019. URL: https://www.youtube.com/watch?v=RoEmGCNsbno.

[15] X.Org Foundation and Christophe Tronche. *XGetImage*. URL: https://tronche.com/gui/x/xlib/graphics/XGetImage.html.

[16] 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.

[17] 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/.

[18] 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.

[19] 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/.

[20] 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/.

[21] Kelsey Lewin. *87% missing: The disappearance of classic video games*. Nov. 2023. URL: https://gamehistory.org/87percent/.

[22] 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.

[23] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[24] 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.

[25] Leo McCormack et al. *leomccormack/Spatial_Audio_Framework*. June 6, 2024. URL: https://github.com/leomccormack/Spatial%5C_Audio%5C_Framework.

[26] 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.

[27] Marjan Mernik, Jan Heering, and A.M. Sloane. "When and how to develop domain-specific languages". Jan. 2005.

[28] 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.

[29] Michael Z. Newman. *Atari Age*. en. MIT Press, 2017. ISBN: 9780262035712. URL: https://books.google.com/books/about/Atari_Age.html?hl=&id=Y20ODgAAQBAJ.

[30] 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.

[31] 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.

[32] Eric S. Raymond. *The Art of UNIX Programming*. Pearson Education, 2003. ISBN: 0131429019.

[33] Roboflow. *Supervision*. URL: https://github.com/roboflow/supervision.

[34] Miro Samek. "Who Moved My State?" In: *The C/C++ Users Journal* 21 (Apr. 2003), pp. 28+30–34.

[35] Piotr Skalski. *Make Sense*. https://github.com/SkalskiP/make-sense/. 2019.

[36] 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.

[37] Valve Software. *Steam Audio*. URL: https://valvesoftware.github.io/steam-audio/.

[38] 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.

[39] 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.

[40] Alan Turing. *The Essential Turing*. en. Ed. by B. J. Copeland. Oxford, England: Clarendon Press, 2004. ISBN: 9780198250807.

[41] Antti J. Vanne et al. “Spatial Audio Reproduction Based on Head-to-Torso Orientation”. 20240357308. Oct. 2024.

[42] 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.

[43] 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/.