Key and Lock Puzzles in Procedural Gameplay

Calvin Ashmore

In Partial Fulfillment of the Requirements for the Degree
Master of Science in Information Design & Technology
School of Literature, Communication and Culture
Ivan Allen College
Georgia Institute of Technology
April 2006

Thesis committee:

Michael Nitsche (chair)
Ian Bogost
Michael Mateas

This research was funded in part through fellowships from the Turner Corporation. Any opinions, findings and conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect those of the sponsor.

Contents

Chapter 1

Introduction

1.1 What is procedural gameplay, anyway?

Before undertaking this research, it was my intent to name the work the "Procedural Gameplay Project", which was a great title, though right away it became apparent that the suggested scope was far too broad. The notion of procedural gameplay derives from procedural content, which is the practice of making content (generally in games) through an algorithm rather than manually. Procedural gameplay is the process of creating things to engage the player through use of an algorithm rather than manual design. The larger goal of this work is to understand this concept through the application of key and lock puzzles.

The goal of this document is to describe a method for creating procedural gameplay in the form of key and lock puzzles, specifically those in the style of game designer Shigeru Miyamoto. I have chosen these games specifically for a two reasons: The first is that key and lock puzzles are very easy to represent algorithmically, and the second is that Miyamoto games tend to capture these puzzles to that they are engaging, but also transparent to the player. Namely, keys do not just open doors, but they also have additional gameplay value.

Gameplay is difficult to define, but it comes down to the notion of what the player does in a game. It depends on the interactive qualities of a game, and the capability of the player to explore, alter, and exhibit agency within the game world. It does not require sophisticated graphics, sounds, or cinematics, but has classically proven to be indispensable for the success of a game. In the past, gameplay has proven itself to be as elusive as it is essential.

Procedural gameplay must be defined as a method in which gameplay is created algorithmically within a game world. The purpose of the algorithm would be to structure generated content so that clear goals and objectives become visible to the player. This involves presentation of long term goals, obstacles, and forming a way that the player can overcome the obstacles. In some sense, this is procedural design, and the application is the simulation of the designer.

Many of us have played games in which there were poorly implemented key and lock puzzles, consisting of many useless items that one must collect before entering the next area and collecting more useless items. A better way to handle these is to have a key that serves another purpose. For instance, the power gloves can move a boulder that blocks the player's path, but also may be used elsewhere. The magic hammer can permit movement past the

wooden posts, but also can help defeat some enemies. The ice beam opens the white doors, but also freezes some enemies.

This perspective on key and lock puzzles provides a unique way to address the larger problem of procedural gameplay. The interaction between keys and locks is comparatively straightforward to formalize, but simultaneously can produce successful gameplay.

1.2 Why go procedural?

A good topic to touch on before going further is the necessity of procedural research. In electronic games, everything is procedural, since the game takes the form of a program. However, the difference becomes significant regarding resources and assets.

In Hamlet on the Holodeck [16], Janet Murray described four properties of digital environments, of which, procedural is one. The other characteristics are spatial, participatory, and encyclopedic. Electronic games are spatial in that there is a game world to be explored, and participatory in that the player interacts with the game world. The encyclopedic character has steadily grown to be the most important in game development.

Generally, manually created content is significantly superior to anything that might be created procedurally. One may compare elaborately written, natural dialogue (which tends to appear in every standard console RPG), to the often absurd text machine ELIZA [20], and see the obvious benefit of prewritten dialogue. The ELIZA example is a little extreme, but similar arguments can be made for other procedural forms of content. In general, procedural content is always inferior to something that a human could produce by hand.

Very early on, the quantity of data that could be stored in memory was small, so procedural generation was used frequently, leading to games such as Rogue[?] and Nethack[?]. But after the CD-ROM era, there was enough memory to rely entirely on the encyclopedic character of digital media, giving way to games such as Myst[?]. Gradually the database became the dominant form of digital media [13]. Eventually this meant that in games, that all forms of content and gameplay needed to be developed explicitly by the artists and designers, and the role of the programmer is to build tools to host this content and make it available to the player.

This trend has continued for a long time, leading to an exponential growth in development costs for contemporary games. This growth is being met with outrage by some game designers [1], and this has lead to a significant resurgence in interest regarding procedural content.

1.3 The organization of this paper

The ultimate goal for the paper is to approach this material scientifically, by observing, building a model, and then testing the model. This approach is divided into three stages. The first is the definition of the problem space and an analysis of existing key and lock puzzles. Chapters 2, 3, and 4 address the observation and analysis of the games.

The following three chapters (5,6,7) take the concepts and properties as listed in the previous chapters and transform them into forms that may be abstracted and simulated by an algorithm.

The implementation is divided into three distinct parts, represented by the next chapters. The first part is the model, which provides a mathematical formalization of the notion of keys, locks, and areas informally described earlier. The model can be used for describing the key and lock features of an existing game, or it may be used to describe something that does

not even exist yet. Following that is the chapter on the renderer, which can take a model of a game and transform that into something that is actually playable. Since not all models correspond to playable games, it is necessary to address the capabilities of the renderer used for this project.

The last chapter of the implementation addresses the simulator, which describes how to create a valid model for keys and locks starting with only a few given components. The simulator bears the most complex role in this system, since it must balance between games that are renderable, but not valid according to the key lock model, and between valid models that are not renderable.

Chapters 8 and 9 examine the results of the work, and discuss the potential applications and importance of the findings.

Chapter 2

Background and Foundation

The following describes key and lock puzzles in a number of games. In these sections, I have categorized the different keys and locks available, and described in some detail how they relate to each other and their impact on gameplay.

2.1 About the games

The games I am concerned with are platform or action games in which the player controls a character in the game world, and they can navigate the world, and interact with other entities in the game world. These games also involve an upgrading process that primarily takes place by the player finding and using items, increasing his or her strength, mobility, and other properties in the game world. This is an important distinction from other games which rely on experience based systems in which the player increases powers and abilities (either consciously or automatically) by performing tasks and fighting enemies (Almost all RPGs, God of War). Other games do not allow the player to increase power within the game world at all, and rely only on the increase in the player's skill (Street Fighter, Katamari Damacy). Many more games employ elements of skill, experience, and item acquisition (Prince of Persia, Psychonauts). The games that I am most concerned with are those in which the primary method of advancement comes from acquisition of items.

Specifically, I am studying two series of games, Zelda and Metroid*. Both of these series were designed by Shigeru Miyamoto, and both employ similar means for presenting keys and locks to the player.

2.2 Why key and lock puzzles?

The foremost reason for the use of key and lock puzzles is that they are very easy to model mathematically. A game may be thought of as a collection of areas, in which adjacent areas have connections. These connections may have a lock that bars the player's entry without the appropriate key or key combination. This is very malleable from an algorithmic perspective.

*Specifically, the following titles: The Legend of Zelda[Z1], Metroid[M1], The Legend of Zelda: a Link to the Past[Z2], Super Metroid[M2], The Legend of Zelda: The Ocarina of Time[Z3], and Metroid Prime[M3].

Additionally, key and lock puzzles have had a long history in games, and there have been many examples in which they have been done well and done poorly. Using these, one can derive a large pool of information on the ways in which key and lock puzzles have been executed successfully or unsuccessfully, and form a collection of principles regarding the best way to employ the puzzles in a game.

The combination of algorithmic malleability and clear pool of examples is an ideal platform on which to base a system for generation.

2.3 Cartesian products

Key and lock puzzles are naturally classified into the category of "puzzles". This is important, because ludologists generally make a strong distinction between puzzles and games. Crawford has argued that puzzles are non interactive, since they do not respond to the player [2]. This is true for key and lock puzzles as well. The keys and locks do not respond to the player inherently, locks remain in stasis until a key is found. However, this may be overcome by building gameplay around the keys and locks.

One could not deny that The Legend of Zelda is a game. However, taken on its own, the key and lock behavior that operates in the game is very puzzle-like. Taken abstractly, the docks from which the player can use the raft are impenetrable locks until the player finds the raft in the third dungeon. However, the process of getting to that item is filled with challenging environments and responsive enemies.

The process by which the complete game occurs though is by applying a cartesian product between the puzzle and ordinary game behavior. Basically, this means that the normal gameplay and the puzzles overlap and coexist. This is a very common practice, and also applies to the story in the game as well.

In the games that I am discussing, there are two layers of gameplay. The first is the low level involving the player moving in the space, talking to non player characters, lifting boulders, fighting enemies, and the like. There is also a higher layer involving more cognitive activity, in which the player must identify goals and obstacles, and devising methods to find the next key and overcome the obstacles. This higher layer is the space that I am seeking to address in simulating the key and lock puzzles.

The coexistence between game, story, and puzzle can be separate or even orthogonal, but experience has shown us that orthogonality can lead to some fairly awful games. The classical example of this is the old laserdisk game Dragon's Lair[DL], which was an animated cartoon that at several points had a question, that, unless answered correctly, would cause instant death. The quality of the animation led it to be was very popular at the time, leading to a massive progeny of terrible spin offs and sequels. The biggest problem with these games is that there is no real cognitive behavior taking place, no planning or thinking on the part of the player. The best games tend to have an intricate interrelationship between story, puzzles, and the mechanics of the game itself.

In games that make use of the key and lock puzzles, it is this intricacy that makes them successful, as well as interesting to study. Part of the intent of this paper is to detangle the relationship between game and puzzle in these titles.

Chapter 3

Concepts and Terminology

In some places, my terminology may seem slightly awkward, as I classify many items in games as keys. Common convention is to refer to these as simply "items" or "powerups". I am making an important distinction here though. For the purposes of this document, a *key* is any item that has an operation on the player's behavior in the space in the game. A *lock* is a property of a space in a game that makes an adjacent area inaccessible until the appropriate key is found. The dynamics of the simulator will primarily address the matter of accessibility with keys and locks.

3.1 Keys and Powerups

Keys must necessarily be items, I am not concerned with levers or switches. These are interesting in their own right, and have been used in fascinating ways to great effect in many games, but items must be directly used by the player, and have some direct effect in enhancing the player's ability to interact with the game space. In other words, keys are obtained, and then permanently become part of the player's state. Most of the rudimentary keys are necessary to proceed to further areas, additional ones are not necessary, but provide additional tools with which the player can interact with the game space. These are keys, because they essentially unlock possible interactions and behavior within the game that was previously inaccessible, even if they are unnecessary to complete the game.

This definition of keys also restricts keys that can be found and then lost again, such as one use keys in the Zelda games. I am restricting the use of these because they complicate the model, and also are generally intended for use in a specific location. The function of having these is to allow the player a choice of where to use the key, but in order for the levels to be completable, some of the doors must have extra passages around them, or contain another one use-key for the other doors.

The fundamental consideration is that keys enable new *verbs*. When a new key is found, they transform the player, allowing new behaviors, abilities, and ways to move through and interpret the world. In Zelda: A Link to the Past, the ice wand is a key in the game. It does not open up any new areas that were inaccessible without it, but it does give the player the new verb *to freeze*. Additionally, frozen enemies smashed by the magic hammer drop full magic refills, giving the ice wand an important and useful place in gameplay.

There are two types of keys: There are unlocking keys, which open new areas. There are also optional keys, that do not open new areas, but rather affect what the player can do in already accessible spaces. Optional keys may also be thought of "tension keys", because they usually reduce the tension experienced in moving through existing areas. Generally, keys are unlocking, so it should be assumed that a key is unlocking unless otherwise specified.

Additionally, I use the term "powerup" to reflect items that improve statistics or build upon items already retrieved by the player. These are things such as heart containers in Zelda, or missile expansions in Metroid. These items are also very useful, often indispensably so, but generally, like with optional keys, finding all of them is unnecessary to complete the game. Powerups also do not alter the player's possible actions or movements within the game space.

Both keys and powerups fall under the larger umbrella term of "collectables", defined in the Game Ontology Project [7]. After completing the system, powerups came to be seen as easy to implement, though this was never an intentional design goal. Representation of powerups would be a very feasible direction for future work, though.

Levers and switches are items that function in the game world, but may never become part of the player. These are rooted into the space itself, making the player make choices and decisions regarding how to access the switch. These are interesting in other games, especially platformers, because they make the space behave like the puzzle (changing structure of space, typically undoable, also time running out ones, etc). My focus is on items that the player can keep that have direct control over space.

3.2 Space

The notion of keys and locks in a game is rooted foundationally in the bearing of space in the game. Locks are properties of the space in the game. When a key is obtained, it changes not only the verbs usable within the space, but also the player's understanding and model of the space. Because of the usability of keys, a key changes space not just by opening inaccessible areas, but by reconfiguring existing areas. Thus, to the player, the game can reshape itself with each key.

This reshaping operates on not just the player's access and movement within space, but also in the player's perception of the space. The matter of perception is difficult to work into a model for simulation, however. Perception relates to the player's understanding and mental model of the space. Some of this may be encouraged in the way that the simulated space is created, but for the most part, changing the readability of the space is the responsibility of the manner in which it is presented to the player.

Movement through space is also often one-way. Many levels in Metroid and Zelda both feature doors that lock, or become inaccessible when the player moves through them. There is always a route around, however. In other situations, an exit to an area becomes functionally locked when the player enters the area, but the key needed to escape is nearby. For instance, in every Matroid title, the high-jump boots are *always* down some hole or passage that requires their use to get out.

3.3 Tension and Flow

Tension is very important to the understanding of action games, and has been studied in many circumstances. Tension relates to risk and reward theory [19], skill and difficulty [18],

and also has its roots in Hollywood 3 act, 9 part movies [4]. More recently, tension has been used in the project Facade to regulate the dramatic structure[15].

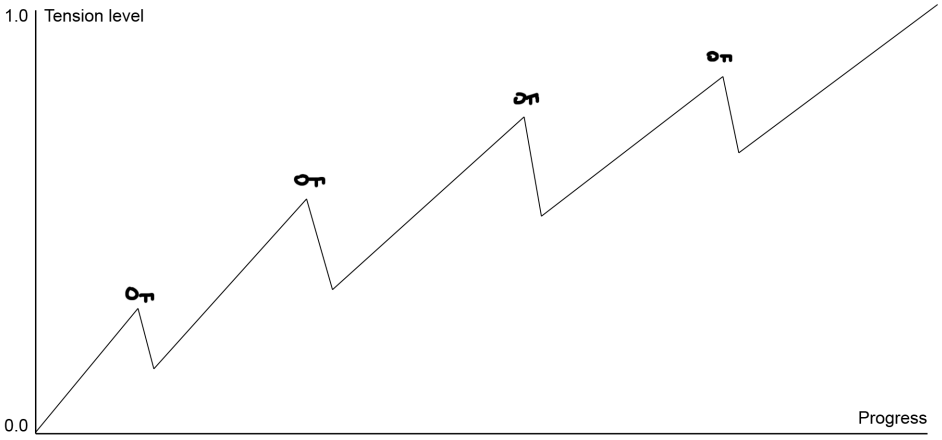
In games, tension is generally described as difficulty, but difficulty is another nebulous term that is hard to pin down. Furthermore, the difficulty of the game fluctuates as the player becomes more skilled, and obtains keys and powerups. Over the course of the game, the spaces tend to be more challenging, but when balanced by the player's skill and state in the game, apart from boss battles, the difficulty should be more or less constant.

In a good game, the player should always feel progress, but also challenge. In Legend of Zelda, consider a player in the first dungeon with only three hearts and a wooden sword facing three blobs. The player in this situation is likely also inexperienced with the game and controls and may wind up easily getting hurt. The same player much later in the game, with many more hearts, bombs, and better equipment will feel similarly challenged facing several of those nasty walking suits of armor.

I will use the term tension here to refer not to the general feeling of difficulty, but rather the intensity of the tasks and opponents facing the player, before compensating for the player's skill and powerups. Key items do affect the tension, though. When some enemies are weak against an item, that lowers tension, but sometimes, after finding an item, stronger enemies appear in previous areas, increasing tension.

For our purposes, tension must be a measurable amount. When the player is in a particular spot in the game, the tension should have some value. Tension relates to the pace and flow of the game, and keys both serve as rewards, and also affect the tension of encountered areas.

There is a tension curve, which reflects the tension experienced by the player over time. This should be bumpy with small spikes that lead to boss battles and finding keys. This curve should grow gradually over time until it reaches the climax of the game.



Chapter 4

Functional Properties of Keys and Locks

The following describes in detail the various important properties and characteristics that make up successful key and lock puzzles. I assembled this catalogue by analyzing Metroid and Zelda games to find the properties that seemed to make the puzzles work in the games.

It makes the most sense to group keys and locks by their functions, rather than other external characteristics. While the properties listed here were specifically obtained from looking at Miyamoto games, but they may doubtless apply in many others.

Additionally, many of these properties are highly debatable. It should be understood that they are not meant to be an exhaustive list of all properties of key and lock puzzles in these games, nor are they meant to apply 100% of the time in every possible instance. However, they should represent the most important properties of keys and locks, and apply about 95% of the time.

What makes these key and lock models successful? There are a few things to consider:

Property 1: Keys have bearing on low-level gameplay.

This is something I'd like to stress. A common complaint regarding key and lock puzzles is that they operate orthogonally to the real play of the game, essentially forming tedious distractions, or unconvincing motives to keep playing. When executed properly, keys should take the form of items that are directly usable by the player, and make the game more interesting by creating more game play opportunities that also happen to unlock new areas. Gameplay here means localized, low level activity, such as using items, moving around, fighting enemies, lifting boulders, and so on.

Some keys open new play options, others do not. A good example of this is in the Metroid series. Many items that the player can get are new weapons, which can also open some doors that were previously inoperable. These weapons also respond in new ways to enemies and the environment. Thus, in addition to getting a new beam to play with, locked doors that have been sprinkled throughout the area are suddenly accessible.

This point is interesting, because it will be shown later to be impossible to represent directly within the simulator. The low level gameplay cannot be represented outside of the game, but it is important for it to exist there. Instead, the simulator must focus on patterns in the higher level cognitive gameplay.

Property 2: Almost always, several visible locks precede the appearance of a key.

Locks are presented to the player before the key is made available. Partly, this occurs so that the player can locate and identify various places to return to after the key is obtained. It also serves to overshadow the acquisition of the key itself, when more and more locks are encountered that require a certain key, then it is time for that key to appear.

When a key is presented to the player, often it is immediately necessary to use it in order to move on from the area in which the key was kept. Some times, a key may also be necessary to defeating a certain kind of enemy, and once the key is obtained, the player is suddenly faced with a dozen of these enemies.

Property 3: Keys are rewards for players.

Before a key is obtained, there is usually a swell of the tension in the game, and once the key is obtained, that subsides for a time, corresponding to a reward. Very often, a new item may be obtained after a boss fight. This is not always the case, however.

A good counterexample are high tension moments that occur immediately *after* the key is found. In such situations, the player may find some key, and then a boss battle occurs, one that requires quick use of the newly found key. However, these situations are not in conflict with the idea of rewards, as after the boss battle, the player has been through a high-tension situation, and also has a new key.

These types of games tend to have gradually increasing tension throughout the game, and be interspersed with spikes and dips, the severity of which is determined by the reward factor of the keys.

Property 4: Keys are encountered with some regularity.

This property is surprisingly relevant to traditional game design. This has a lot to do with the pace and flow of the game. The player should feel accomplishment and rewards with regularity so that the game stays interesting and doesn't get too hectic or dull. The pacing of challenges and rewards in games varies from game to game, but is a common phenomenon[3]. Failure to practice this property involves odd bursts of finding lots of keys so that it's hard to keep track of them, and then long lulls in which few or none are found.

Generally, keys are encountered regularly, but also in conjunction with more traditional powerups, which are more frequent. Keys tend to be more useful and interesting, and thus are slightly more rare. A good ratio for keys to powerups is 1:2 or 1:3. These patterns are closely related to risk and reward theory in games [19].

Property 5: Old keys are still useful.

In the Zelda: Ocarina of Time, the final dungeon had several sections filled with lock puzzles. By that point in the game, the player has obtained everything that there is to get, but the puzzles were presented in different ways, requiring various combinations that had not been previously necessary. The use of the keys themselves is the essence of gameplay, and almost all of the recent Miyamoto titles have some span of time in which the player does not obtain anything new, but rather explores the possible uses of the available keys.

Property 6: Approximately between 3 to 10 distinct inaccessible locks are opened with each new key.

This property is hard to articulate, but it is rather significant. The corresponding question for this property is: *How many new (currently inaccessible) distinct locks are opened when one key is found?*

This is a tricky matter. When a key is found, it does not open just one lock, but a number of them that have been spread throughout the game so far. Also, keys continue to be used afterwards, as per the previous property. For this property, I am only concerned with those locks that may be encountered (but not breached) without the key. After the key is found, the locks may still be used in other unexplored areas of the game, since the act of using the key item is gameplay and make things interesting for the player.

When the player finds a key, there must be a number of previously explored areas that had locks in them. There should be enough of these so that the player can feel like there are options in where to go next, but not so many that the player has a hard time keeping track of them. This property is especially important for generation, because it will inform the generator how many late-stage locks must go in the early areas of the game.

The actual 3 to 10 figure is a general estimate, but reflects what I have found in Miyamoto games. Balance is important here, because too few new openings will be leading the player down a narrow path, but too many will also be distracting, and make it hard to identify where to go next.

Property 7: Locks protect small rewards, and new areas.

There must always be a reward of some kind behind every lock the player encounters. Rewards may be a smaller powerup, a new area, a bit of story, anything beneficial. Usually, one of the locks will lead to the next area of the game where the player is intended to go. This lock is often near where the key was found. Having it be much further away leads to backtracking, which can irritate the player.

Some degree of backtracking is okay, because it encourages the player to reevaluate the game space from the perspective of having the new key. Additionally, when the player does backtrack, it should be possible to find some previously inaccessible areas that yield small rewards, so that there are more periodic rewards in the previously explored areas.

Property 8: Keys improve movement through previously explored space and change the player's perception of it.

When the game space is large and complicated, it is difficult to move through the game without doing at least some backtracking. Fortunately, some types of keys are also useful because they improve movement within a space. Consider a room with some unpleasant enemies that continually show up. A recently obtained item could allow the player to move past them undetected, saving time and frustration.

Some keys allow for movement in the space in drastically different ways than before, such as the morph and boost ball items in the Metroid series, or the masks in Zelda:

Majora's Mask[?]. These have an added affect of not only altering the player's movement of a space, but they also alter the reading of the space to account for the new mobility.

Property 9: Locks are not always immediately visible.

A classical element in action and adventure games is the notion that something may be a lock, but is not immediately recognized as such. This lets the player go ahead and ignore the obstacle, moving on to whatever seems to be pressing at the moment. Later, though, the key is obtained and the locks are recognized, allowing the player to revisit the original space with the keys necessary to master it.

This property serves a double purpose: the first is to prevent the player from attempting to unlock the area, when the key is meant for much later in the game. It is exciting to some degree to see a barrier and recognize it as such, and then get the key for it in a little while. On the other hand, it is infuriating to recognize a barrier in the beginning of the game, and not be able to do anything about it until the end. The second purpose is that it allows the old space to seem new and expansive, when it is really just the player's perception changing.

Property 10: Some keys are non-essential.

Many keys can be found in various Miyamoto games that never need to be used at all. They may be helpful, but are non-essential. The problem with addressing these is that it launches a slippery slope, in that ordinary many powerup items, like health bonuses, or extra ammo, could be considered non-essential keys. Additionally, many Super Metroid virtuosos pride themselves on being able to circumvent locks that are clearly intended for specific items*. These are rather extreme examples, but, there are many other items in both Metroid and Zelda games that are never required (The spring ball in Super Metroid, the ice rod in Zelda 3). However, they are nonetheless very useful in certain situations.

To compensate for this, there are two points. One, as I have mentioned before, keys fundamentally different from powerups, because keys open new gameplay options. And two, Keys affect the flow of tension within the game. So, a key may be non-essential, but it may effectively halve the tension of moving through a particular area. If a player only requires five keys to complete the game, but there are more implied but not necessary, that is fine, but the net tension curve of the play through will be high and awkward. On the other hand, if a player has trouble in one stage of the game, the introduction of a key could balance out their tension curve, even if the appearance of the key was unanticipated.

*I'll explain in detail: With immense patience, it is possible to learn how to time bombs as to bounce the morph ball and reach an arbitrarily great height. This neatly invalidates the need for any of the height boosting items, such as the high jump boots, and the space jump. In fact, for the determined, it is possible to get through the game with a mere 15% of the items[10]. This is endlessly fascinating to me because it is in such defiance of the implied key and lock structure of the game.

Chapter 5

The Model

Ludologists have consistently referred to games as systems of one form or another [1, 2, 18]. Though it is near impossible to pin down a single model that fits everything, due to the nebulous nature of games. Attempts have been made using various techniques, such as Petri Nets, generalized game theory, abstract control systems, and other sorts of things. It is worth addressing some of these systems to establish why I am choosing not to use them.

Petri Nets are a type of network that easily represent states and dependencies and relationships. These have been used in a variety of places to represent narratives and actions in games[17]. Petri Nets are ideal at representing small and tight systems, but that makes them untenable here due to the large nature of the game space and the in-character manifestation of keys.

Another modeling strategy is to use abstract control systems, which are significantly more generalized [6]. These are control systems in the sense that the player is modeled via a controller or external input to a system. This approach is versatile, but is unnecessarily abstract for use here, since many simplifications are made in the treatment of keys and locks.

Generally, models are used to formalize the important parts of a system, but abstract away everything else. Due to the particular nature of key and lock problems, the model I describe here is not concerned with representing all games, or even the narrative or gameplay outside the role of keys and locks.

5.1 Understanding the model

The model given here is used to describe the interworking of keys, locks, and space, but not address the concern of gameplay or player input. What is described below does not describe one model, but rather a system for deriving or producing them. This is to clarify that the model is not the game, and the game is not the model. There are two types of important transformations: One can *interpret* a game to produce a model of it (which will be described in the various functions below), and one could *render* a model into a game, filling in all of the details that have been abstracted away.

$$\begin{array}{ccc} Game & \overset{interpretation}{\longrightarrow} & Model \\ Model & \overset{rendering}{\longrightarrow} & Game \end{array}$$

A model will consist of $(G, keys, locks, K, L, U, Tension, x_0)$, which is all that is needed to represent any game for my analysis. The features and constraints described below can be used to describe infinitely many games that have these behaviors. One way of looking at it is to think of this as representing the set of all models that have valid key and lock behavior.

There are many important concepts that must be visible to the model, but which the model should not need to control directly. Tension is a good example of this: The model should be able to represent tension numerically, but not address the specific implementation of it, which could be anything from spatial puzzles, to enemies, to boss encounters. These qualities would need to be analyzed by an interpreter, or represented in a game world by a renderer.

One could interpret one game into several different models. It is also possible to produce multiple games from a single model, a process known as "skinning". An interesting example of skinning is the game Legend of Zelda, in which there are really two games. After completing the game once, the player is treated to restart from the beginning in a very similar game, but one in which all the dungeons are rearranged, the placement of the keys are different, but otherwise the mechanics are the same. Under this analysis, the games would overlap significantly, since $keys, locks, U, x_0$ are the same, but the space, pattern, and ordering are wildly different.

5.2 Mathematical Foundation

The following describes the mathematical terms that I am going to use to describe the properties of these games. Those who are math wary will probably wish to gloss over the math intensive sections. I will do my best to provide a plain-english explanation after each math heavy section, indicating the relevance and implications of the formulas.

5.2.1 General Terms

The game G is a simple directed graph $G = (X, E)$, where X is the set of all locations in the game, and E is the set of all edges (x, y), indicating that there is connection $x \to y$ for $x, y \in X$.

It is important to note that this definition is extremely vague, and one could provide several different valid formulations for X and E corresponding to a single game. Values of X could be potentially continuous (pixel sized), or the could be the size of entire sectors of the game. These are both valid, but not very useful. I would like to interpret X as being reflective of the natural partitioning of the areas in the game.

The set of locks $locks = \{lock\}$ is the collection of all possible locks that can block one edge. The set $locks$ must contain a zero element 0, indicating that there is no lock needed for an edge.

The set of keys $keys = \{key\}$ represents all the available keys in the game.

For this paper, keys are kept when found. In games this is not always the case. However this treatment is consistent with the definition of keys that I have laid out in the chapter on terminology.

The function $L : E \to locks$ gives the lock that blocks any given edge. For edges that are not blocked in any way, L points to 0.

A function $K : X \to P(keys)$ maps each location in the game to a subset of the keys that may be obtained at that location. Both K and L are responsible for keeping track of the arrangement and behavior of the keys and locks in the game space.

The unlocking relation U is best represented as a set:
$U = \{(kset, lock) : kset \subset keys, lock \in locks\}$

This definition allows a great deal of flexibility in defining how to deal with locks. For instance, if $(\{k1, k2\}, lock1) \in U$ means that both $k1$ and $k2$ are necessary to unlock $lock1$. However, there may also be $(\{k3\}, lock1) \in U$, which would imply that $k3$ would unlock $lock1$ all on its own.

The following properties of U must hold:

1. Directed: $kset1 \subset kset2$ and $\exists(kset1, lock) \in U$, then $\exists(kset2, lock) \in U$.
2. Handling of the zero element: $\exists(\emptyset, 0) \in U$.

Tension is a function $T : X \times P(keys) \to \mathbb{R}^+$. Tension represents the player's tension at a particular area, given a set of keys.

This is tricky, due to the importance and nebulous quality of tension. However, this must be a quantity that is measurable by the simulator in some way. Some keys will reduce the tension of moving through various edges. But often, others will have little effect. Non-essential keys should affect T, if they have no bearing on U. Tension is represented numerically here, the manner of the actual implementation is not specified by the model.

Additionally, it is necessary to be very careful with the tension function, since the simulator will optimize on it. The simulator may interpret a human specified tension function in a manner unlike the intended meaning.

The player is $(x, kset)$ for $x \in X$, $kset \subset keys$. x is the player's location, and $kset$ are the keys possessed by the player.

This is, oddly enough, all we need to represent the player. All of the other details, such as attributes, parameters, health, and the like are abstracted away by the system.

5.2.2 Movement Terms

The following may seem overly complex, but some of this terminology is necessary to establish the model for the simulator. Much of it is derivative of the fact that G is a directed graph, so there may be one-way edges. This fact is very important, since most all of these games feature paths that are one way. However, these paths eventually let the player back out into the whole space of the game again. This is tricky to represent mathematically, since special considerations need to be given for the directed nature of the graph. On the other hand, it is necessary since the simulator should create one way passages, but it should abide by rules in allowing the player to return.

Adjacent is a function $X \to P(X) := x \mapsto \{y \in X : \exists(x, y) \in E\}$.

Solvable is a function $P(keys) \to P(locks) :=$
$kset \mapsto \{lock \in locks : \exists(kset1, lock) \in U,\ kset1 \subset kset\}$.

Accessible is a function $X \times P(keys) \to P(X) :=$
$(x, kset) \mapsto \{y : y \in Adjacent(x),\ L((x, y)) \in Solveable(kset)\}$.

A path is finite sequence $\{x_1, x_2, \cdots, x_n\} \subset X$ such that
$x_{j+1} \in Adjacent(x_j) \forall j \in 1..n-1$.

KeysOf is a small function $P(X) \to P(keys) := A \mapsto \bigcup_{x \in A} K(x)$.

LocksOf is, correspondingly, $P(X) \to P(locks) := A \mapsto \{L((x, y)) : x \in A,\ (x, y) \in E\}$.

An accessible path with key set $kset$ is a path such that
$x_{j+1} \in Accessible(x_j, kset) \forall j \in 1..n-1$.

It is convenient to say $\exists Path(x, y)$ or $\exists Path(x, y, kset)$ to imply that there is a path or an accessible path from x to y. These terms are very useful in discussing zones and movement through game space.

AccessibleZone is a function $X \times P(keys) \to P(X) :=$
$(x, kset) \mapsto \{y \in X : \exists Path(x, y, kset)\}$.

ReturnableZone maps $X \times P(keys) \to P(X) :=$
$(x, kset) \mapsto \{y \in X : \exists Path(x, y, kset),\ \exists Path(y, x, kset)\}$.

ObtainableZone maps $X \times P(keys) \to P(X) :=$
$(x, kset) \mapsto \{y \in X : \exists Path(x, y, kset),$
$\exists Path(y, x, kset \cup KeysOf(ReturnableZone(y, kset)))\}$.

The goal of establishing these various functions and relations is to establish three types of zones: accessible, obtainable, and returnable. An *accessible* zone is the set of all areas that the player can go, but from which it may not be possible to return. A *returnable* zone is the set of all areas that the player may visit and return. The trickiest of these is of course the *obtainable* zone, which is the set of all places that the player may visit, but from which it is possible to return, given the keys found there. They follow a nice ordering:

$$ReturnableZone \subset ObtainableZone \subset AccessibleZone$$

Nicely enough, this holds true for all $x \in X$ and $kset \subset keys$. However, for the games that I am seeking to address in this paper, it always* the case that, if the player $(x, kset)$ is in a reachable configuration,

$$ObtainableZone(x, kset) = AccessibleZone(x, kset).$$

This is a significant claim and it improves the mathematics. This means that the player can never get stuck. Furthermore, it enforces that the player finds only one key at a time. This is especially important, as it is something that must be implemented in the simulator.

Were that not the case, then the game could be very interesting, because certain paths could be definitively one-way. This could be beneficial, in which there were separate paths that

*Actually, this is not completely true. In the original NES Metroid, it was necessary to get the ice beam before getting the bombs, and the bombs let the player fall into some pits, which required the ice beam to get back out. However, it was also possible to change to the wave beam, which would make the player get stuck after falling into one of those pits. Those situations would force the player to reset the game, often leading to a rotten mood. That's why I'm omitting their use.

contained unique keys, increasing the replay value for the player. or it could be frustrating, disorienting, and possible to get stuck. The condition that accessible zones are obtainable forces a certain understanding and safety in the space. Regardless of where the player goes, even if it is through a one-way door, the player will always able to find a return route, or find one using a key that found past that one way door.

The notion of a reachable configuration must be clarified, however. A reachable configuration is a player (position and key set) that can be reached by starting at the beginning of the game $(x_0, \emptyset)$. Essentially I am excluding situations like taking young Link from Zelda: Ocarina of Time, and putting him in the middle of Gannon's tower in the future. Young Link may be able to get into a room, but he probably won't be able to get back out.

5.2.3 Zones

In the analysis of the course of play, it is possible to think of the game world as something that is gradually opened up, bit by bit, as the player progresses and obtains keys. Since I am excluding the possibility of the player being able to lose keys, the possible accessible zones continue to increase. A good way to think of this is as a finite sequence:

$$z_0, z_1, z_2, \cdots, z_n \subset P(P(X))$$

In this sequence,

$$\begin{aligned} zonekset_0 &= \emptyset \\ z_0 &= ObtainableZone(x_0, zonekset_0) \\ z_n &= X \end{aligned}$$

We may define z_i, and $zonekset_i$ iteratively for $i \in 1..n$ as follows,

$$\begin{aligned} zonekset_i &= zonekset_{i-1} \cup KeysOf(z_{i-1}) \\ z_i &= ObtainableZone(x_0, zonekset_i) \end{aligned}$$

The sequence of zones is of course strictly ordered, as well:

$$z_0 \subsetneq z_1 \subsetneq z_2 \subsetneq \cdots \subsetneq z_n = X$$

Essentially this lets us take a large complex game and peel off the layers of its space one by one. It allows linear analysis of an otherwise qualitatively nonlinear medium. The ability to do this is heavily dependent on the fact that accessible zones are obtainable. This allows for the space in the game to be gradually unwound by the player. Furthermore, the player may always return to the starting location [†].

Although the ordering presented here is linear, with some effort it does not have to be. I have used a linear ordering because the games I am modeling are usually linear, and it allows for a great deal of simplification in the analysis. Also possible would be to expand the sequence into a simple directed acyclic graph, in which the connections would represent dependency between zones.

[†] In some games, there is an introductory sequence that takes place in an area separate from the ordinary game space, and may not be returned to. For purposes here, the introductory sequence is not considered a part of the game. This may be a painful omission, but the introductory area generally does not contain any keys and may be excised quite naturally.

5.2.4 Tension curves

The above describes a natural progression of keys in addition to the progression of zones. This progression generally occurs one key at a time, but could allow for more than that. It is thus possible to obtain a loose ordering of all of the keys. If key_i precedes key_j then $key_i \prec key_j$. This ordering should be defined by the sequence of $zonekset_i$'s above.

For each $key_i \in keys$, there is a corresponding $x_i \in X$ such that $key_i \in K(x_i)$. There is also an accessible path $path_i$ from x_i to x_{i+1} for $i > 0$ with the key set $kset_i = \bigcup_{j<i} key_j$. The special case of $path_0$ is the path from the starting location x_0 to x_1. Using this, we can map out the tension curves

TensionCurve maps a path and key set to a sequence of numbers, representing the tension of the path to that key. $TensionCurve : P(X) \times P(keys) \to P(\mathbb{R}^+)$

$$TensionCurve(path, kset)_j = Tension(path_j, kset)$$

It is also useful to say $TensionCurve_i$ to represent $TensionCurve(path_i, kset_i)$.

NetTensionCurve is the tension curve of the entire game, given our ordering of keys from above. Because each of the tension curves is finite, I will use the union operator to represent appending of the sequences.

$$NetTensionCurve = \bigcup TensionCurve_i$$

KeyZone maps a key to the number of the zone in which the key is found. This is useful to make statements about the zones later. $KeyZone := key \mapsto \min j : key \in zonekset_j$

Unlocked maps a key to the set of all locks that this key opens when it is obtained.

$$Unlocked := key_i \mapsto \{lock \in locks : (kset_i, lock) \in U, (kset_{i-1} \notin U)\}$$

5.3 Representation of the properties

Many of the properties discussed in the earlier chapter can be articulated mathematically here. While most can be represented as mathematical constraints on the model, other properties rely instead of the composition of the game itself, and how the model is represented as a game. These properties are equally as important, but must be represented by the renderer instead of the simulator.

Property 1: Keys have bearing on low-level gameplay.

The most important property of key and lock puzzles is impossible to directly articulate within the model. Keys must have gameplay value, and they must be usable. It is possible within the model to address the effect of a key on the game space, in terms of its ability to open spaces as well as change movement within them. It is also possible to represent that in certain spaces, keys must be used. However, it distinctly not possible to simulate directly what the keys do themselves. This must be delegated to the renderer.

Property 2: Almost always, several visible locks precede the appearance of a key.

I will narrow this down and make it slightly more specific: There must exist at least three locks for a key key_j preceding its retrieval. These locks should be encountered close to the key, but not necessarily all in the new part of zone $KeyZone(key_j)$. The valid close zone consists of:

$$close_j := z_{KeyZone(key_j)} \backslash z_{KeyZone(key_{j-1})-1}$$

for j such that $KeyZonej - 1 > 0$. Otherwise $close_j = z_0$.

The property takes the form of:

$$\forall key_i \, |Unlocked(key_i) \cap LocksOf(close_i)| \geq 3$$

Note that this definition is rather tight, and does not encourage placement of late locks in the early areas of the games. This property focuses on the time leading up to finding the new key.

Property 3: Keys are rewards for players. Combines with,

Property 4: Keys are encountered with some regularity.

The first of these properties means that there is a risk for each key, which translates into a spike of game tension. The second means that these spikes are regular. Thus, there is some kind of spike in the tension curve leading up to every key, and the tension drops down after the key is found. Games generally have different rankings of keys, corresponding to their impact on the tension curve. I will measure the worth of a key by two things: the number of new areas opened by the key, and the key's impact on tension for explored areas.

KeyValue $KeyValue : keys \rightarrow \mathbb{R}+$.

$$key_i \mapsto accessWeight|Unlocked(key_i) \cap LocksOf(z_{KeyZone(key_i)})|+$$

$$tensionWeight \sum_{x \in z_{KeyZone(key_i)}} |Tension(x, kset_i) - Tension(x, kset_{i-1})|$$

This definition of course depends on two numbers, $accessWeight$ and $tensionWeight$, but these depend on the size of the game space X, and the value of the $Tension$ function.

In order to create the spike, the tension should increase until the key is found. Thus, for each i, $TensionCurve_i$ is increasing. The net increase is:

$$NetSpike_i := TensionCurve_{i,length(path_i)} - TensionCurve_{i,1}$$

There must also be a dip afterwards:

$$TensionCurve_{i+1,1} < TensionCurve_{i,length(path_i)}$$

To factor in the value of the key, the tension curves must obey this constraint:

$$NetSpike_i \propto KeyValue(key_i)$$

Namely, a key will make an impact on the tension curve in proportion to its value in the game. Thus, if key_1 is twice as useful as key_2, then key_1 should be twice as hard to get. The notion of usefulness is relevant only to how many *currently accessible* locks it opens, and the impact on the tension of the currently accessible space. So if key_1 occurs very early in the game, and opens one door, it is not measured as that useful, even if it is used constantly afterwards.

I will add one more constraint here, to give the possible tension curves the shape they need: Tension must be confined between 0 and $MaxTension$, where $MaxTension$ is the tension of the final area of the game, x_{final}. It is necessary that $Tension(x) < MaxTension$ for $x \in X$ and $x \neq x_{final}$. This ensures that the net spikes for all keys will balance out properly.

Property 5: Old keys are still useful.

This property means that old locks are still encountered after their key is found. I will say that there must be at least 3 occurrences where old locks are still used, but this number may of course be greater. The following must be valid for every key except for the last one.

$$\forall key_i \, |\{lock \in LocksOf(X \backslash KeyZone(key_i)) : (kset, lock) \in U, \, key_i \in kset\}| \geq 3$$

Property 6: Approximately between 3 to 10 distinct inaccessible locks are opened with each new key.

This test wants there to be about 3 to 10 connections emerging from one zone into the next, such that the connection is opened by the new key. This may be formalized as follows:

$$\forall key_i \, |\{x \in z_{KeyZone(key_i)} : \exists y \in z_{KeyZone(key_i)+1}, \\ (x, y) \in E, \, \exists (kset_i, (x, y)) \in U\}| \in [3, 10]$$

Property 7: Locks protect small rewards, and new areas.

Every lock leading from one particular zone to another must either hold a small reward, or the area that the player is supposed to visit next. The current model, as it stands, does not support the placement of small powerups. The only available solution is to make sure that there may be some dead-end areas in which such powerups could be placed.

Property 8: Keys improve movement through previously explored space and change the player's perception of it.

This property creates a phenomenon of gateways and barriers. For instance, there may be a river, and a certain high level key is needed to cross that river. There may also be a bridge leading to the other side, and the bridge has a gate in it that requires a weaker lock than what is needed to cross the river. In this situation, the river is the barrier and the bridge is the gateway. In order to cross the river, the player must go to the bridge and cross it, but after finding the key to pass the river, suddenly the gateway becomes unnecessary, and the player is freed within the space.

Not all keys may open barriers, but there should be at least one. This property is expressible as follows:

$$\exists key\, \exists kset \subset keys\, key \notin kset,\, \exists x, y \in X$$

$$\exists path1(x, y, kset),\, path2(x, y, kset \cap \{key\}),\, |path1| > |path2|$$

Property 9: Locks are not always immediately visible.

The matter of visibility is not expressible directly within the simulator, but rather must be manifested within the renderer. The change in visibility occurs to the player a key is found that opens one type of lock, whose signature has appeared in previous areas in the game, but was not recognized as an obstruction. The matter of hiding is dependent on how the renderer represents the locks within the game space.

Property 10: Some keys are non-essential.

This property asserts the existence of optional keys. These do not need special treatment, and already function within the model as it stands. This is because not all keys need to unlock something, since some keys may simply not appear in the unlocking relation U. These keys may affect tension in areas instead.

These properties express constraints and goals for checking the validity of a model. Given a whole, completed model, one can check if it abides by these properties. The implementation, however, must use incremental generation, and these properties will transform from constraints to evaluation goals. This transformation is described in the following two chapters.

Chapter 6

The Renderer (Charbitat)

The notion of a renderer is unusual for game design, but is common in mathematical logic. A renderer is something that may transform a model of something into a tangible entity*. The last chapter was devoted towards a method for obtaining a model for certain games, and deriving a set of rules that held true under those games. This chapter seeks to address how an abstract model may be transformed into a game.

With the goal to create games from arbitrary models, one will soon discover that not all models are renderable, quickly leading to an impasse. To address this, I will say that a model is *renderable* if it may be transformed into a playable game using a renderer.

Most games require substantial development teams to design, create assets, and write the program itself. And still, these games are poorly suited towards building new game areas with wildly varying spaces, tension behavior, and key and lock placement. To address this problem, I have turned to building the renderer off *Charbitat*, an existing project whose goal is to explore procedurally generated environments.

Furthermore, the essential properties of key and lock puzzles that are not expressible in the model may be expressed nicely within the renderer. For instance, the property that keys have gameplay value. This is very important, and it is not possible to express in a mathematical abstraction of the game, but it may be expressed by the renderer itself.

6.1 Procedural space in Charbitat

The original goal of Charbitat is to create procedurally generated environments, that are based on player behavior. Existing games have sought to map the player's actions in game to the player character's appearance (for instance, Fable). But none so far have attempted to map the player's actions to the environment in the game. In Charbitat, the player builds the world by navigating through it. New areas are formed based on the player's state (called the player's element), but they are permanent for the duration of the play through.

Space in Charbitat is discretized into square tiles. Each tile has an element, which reflects the player's element upon first entry. The terrain of the tile, the objects placed within, the types of enemies, and the lights and sounds present are based on the element of the tile.

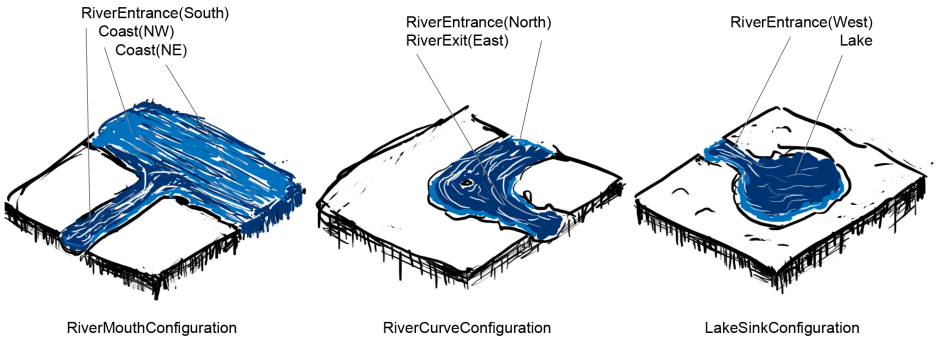
*In mathematical logic, a rendered model would be described as an *interpretation* of that model.

Each object, terrain filter, and other placeable component has its own element associated with it. These are all placed according to simple weighted selection.

Objects and other local features define the local character of the space, but do not provide any global legibility of the space. Kevin Lynch discusses five qualities that are used in the reading spaces and building of cognitive maps [12]. These are: landmarks, districts, paths, nodes, and edges. The tiles in Charbitat represent districts fairly well, but do not convey landmarks, nodes, or paths easily. The solution to this is was to provide an extra layer on top of the tiles consisting of features, constraints, and configurations.

The use of features and configurations allows the formation of global structures such as rivers, bridges, lakes, coastlines, walls, cliffs, and also special tiles. Because locks are a property of the space in a game, locks may be expressed in Charbitat using these configurations.

Features represent the individual parts of these structures. The river features consist of RiverEntrance(Direction) and RiverExit(Direction). Individually, they do not mean anything. However, a RiverEntrance and a RiverExit together (provided that their directions are not the same) represent a RiverCurve configuration.



Each type of feature has a corresponding constraint, which defines whether a given feature may or must occur in a new tile. If the tile to the left of the new tile in question has a river entrance facing the new tile, then the current tile *must* have a river exit on the left to feed into it. Alternately, if the neighbor is blank, then no river features may be placed facing the neighbor. Constraints also may be used to provide probability estimates of how likely a feature ought to be. This may be used to avoid deadlocks, discouraging features that would lead to invalid configurations, though the current version of Charbitat does not implement this.

Features may be combined into configurations, and the set of all configurations represents all the valid possibilities of features that any tile may have. It is the configurations that control the areas as understood by the simulator. In making the areas controlled entirely by Charbitat, we avoid the problem where valid models are not renderable.

The process by which the configuration is chosen for a new tile is as follows:

1. For each Feature f,
2. Determine if f *must* be placed, or if it *must not* be placed within the tile. Place f in a set $included$ or $excluded$ accordingly, otherwise do nothing.
3. For each Configuration c,
4. If c contains a feature in $excluded$, skip c and try the next configuration.
5. If c does not contain all the features in $included$, again skip it and try the next one.

6. Determine the score of c base on the evaluators in the simulator. Place $(c, score)$ in a set C.
7. If C is empty, there is no possible valid configuration, and a failure condition is reached.
8. Build the maximum score $maxScore = \sum_{(c,score)\in C} score$.
9. Pick a random number $value \in [0, maxScore)$.
10. For each $(c, score)$ in C
11. Assign $value := value - score$
12. When $value < 0$, return c.

After the configuration is chosen, the areas that are present in it are connected to the existing network of areas in the game.

6.2 Manifestation of keys, locks, and tension

Locks are built into the space, and so these are manifested as tile configurations that would require some item to pass freely. While locks are embedded in the tile configurations, the keys themselves must be placed by the simulator, though it is up to the renderer to make them interesting. Continuing in the theme of the Zelda and Metroid titles, the keys should consist of items that allow the player to more easily navigate environments, as well as healthy selection of useful weaponry. Specifically there are the following keys and locks:

The keys consist of five generic weapons, corresponding to the elemental theme of Charbitat. Each of these can open a particular type of gate. Gates may be placed in walls, on bridges, or on ramps leading up onto cliffs. Additionally, there are weak walls, that may be blown up using bombs (which are effective against some enemies as well). Another key is the swim gear, which allows players to cross rivers. Finally, the last key in the game allows the player to fly for a short period of time, essentially opening up all the remaining spaces.

Tension is more complicated to represent. In the model, tension is a function that maps a key set to a number representing the general difficulty of an area when those keys are held. Tension is not represented in the structure of the game model, but instead it provides some numbers that must be handled by the renderer. So in order to make tension workable, there should be a number of discrete tensions that may be applied to an area in a tile. Each of these tensions corresponds with some placement or configuration of creatures and objects in the game environment that is understood by the renderer. Based on the tension numbers alone, the simulator decides which of these configurations to instantiate within the world. This way, the renderer sees only the objects and creatures that it must place, and the simulator sees only the numeric values. A human author is necessary to craft these tension configurations, though.

For example, a renderer may know how to express areas with the following tensions:

1. Has a few low level enemies, tension is $.1$, always.
2. The area has some tough enemies that are vulnerable to a specific weapon. The tension is $.5$ normally, but $.2$ after the key is found.
3. Boss encounter, the tension is $.7$, and does not recur, so the tension thereafter is $.3$.
4. Jump puzzle to climb to a ledge. The tension is $.4$, but reduces to $.1$ after the jump enhancing key is found.
5. The final boss, very tough. Tension is 1.0.

Representing the tension in this way allows the simulator to address curve fitting and optimization impartially, without getting bogged down in the actual implementation of how tension is manifested in the game world. Furthermore, it allows for creating small and manageable tension types that will work in any given area.

Chapter 7

The Simulator

This project has undergone one major change since conception, and that change is the role of the simulator. The goal of the chapter on the model was to describe in detail the model used to represent key and lock problems within games. The original intent for the simulator was to create representations of games from scratch that conform to the model.

$$Nothing \overset{simulation}{\longrightarrow} Model$$

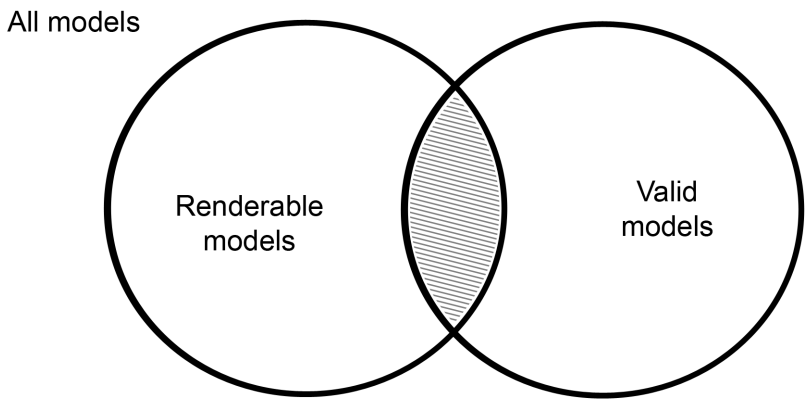
Unfortunately, for the reasons described in the previous chapter, arbitrary models produced by the simulator may not be renderable as games. Rather, what is needed is a *dialogue* between the renderer and the simulator in order to create the model and the game. This requires an intricate balance between the two components.

$$\begin{array}{ccc} Renderer & \rightleftarrows & Simulator \\ & \Downarrow & \\ & Model\&Game & \end{array}$$

The chapter on the model may be thought of as describing the set of all possible games that have these key and lock behaviors. Divorcing the model itself from the various properties that must hold for the model to be valid, it becomes possible to think of the space of all games described by the model, some of which are valid, some of which are renderable, and some of which are both. The intersection of these describes those models that can actually be simulated and turned into actual games, and how to select those is what I will describe here.

While it would appear that the problem here is basically that of selection, the game models that may be produced this way are necessarily going to be very complex and interconnected. It is not feasible to approach simulation without taking an incremental approach. With the idea of a dialogue between the renderer and the simulator, this essentially works by having the renderer create a list of possible new areas in a new tile, and the renderer and simulator negotiate which option is the best suited for placement.

This is tricky because the simulator must judge incomplete models based on whether the model will be valid when it is completed.



7.1 A quick note on the simulator

There are two main methods for running a simulation. The first is to run the simulation *offline*, building the entire game space and then allowing the player to engage it. Alternately, *online* simulation works simultaneously as the player explores the environment, reacting to the player's choices and behavior.

Online simulation is much more responsive and interesting research problem, so the simulation methods described here will all apply to online simulation unless otherwise specified.

7.2 Dialogue

From the earlier chapters, the following is clear:

1. Areas and locks are dependent on the renderer.
2. Keys and tensions must be placed by the simulator
3. There is a method for selecting configurations from a pool of possibilities.

Putting these together, the logical step for the simulator was to create an evaluator that took all the possible configurations and evaluated these depending as to restrict those that would lead to invalid models, adding its own weights towards preferable options, and allowing the renderer to select one and work it into the game.

Because keys and tensions are not directly controlled by the renderer, special considerations must be made to take them into account. As shall be shown momentarily, zones must also be assigned by the simulator as well. It is not feasible to create all possible variants of all of these and select from all of them*, but rather, zones, keys, and tensions must be varied independently, and the best arrangement of zones, keys, and tensions will be worked into the network of areas in the model.

The simulator must have a set of evaluators, each of which tests for some property, and can discard configurations that are invalid, or will inevitably become invalid. Otherwise, evaluators may encourage or discourage certain traits of possible configurations. Because zones, keys,

*Sadly, this I know from experience. It led to slow code and many out of memory errors.

and tensions must be evaluated individually, each evaluator should have a variable describing whether it is meant to operate on any of these specific components, or if it is meant to be general.

7.3 Zoning

Zoning is extremely important for the simulator, because it provides a very clean breakdown of game spaces that are otherwise very difficult to analyze. To the renderer, a zone has no meaning, but in gameplay, the next zone reflects the area that the player is trying to reach.

The chapter on the model addresses zones mathematically. In this sense, zones are sets of areas all of which require some set of keys to become accessible. The zones themselves are ordered in a directed acyclic graph (or just in a sequence). For this project, I have restricted the possible zones to be ordered in sequences, but they by no means have to be. programmatically, zones are aware of the keys they contain, the keys the require to be entered, and their predecessors (or parents) and successors (or children).

The zone graph may be built randomly given a set of $keys$, $locks$, and an unlocking relation U. In my implementation, the domain of U is composed entirely of singletons, meaning that no lock requires two keys to open. Additionally, for every key save the last one (the one that allows the player to fly), the relation of U is one to one, one singleton key maps to one lock. Thus, each zone in the zone sequence corresponds to the lock that blocks it. The first zone necessarily requires no keys, and the last zone requires the fly key. The rest are randomly selected amongst the remaining possible locks that are present in the game world.

The hardest problem with zoning is how to decide what zone an area should have given the connections from and leading to that area. If two areas already have zones, there is one rule to follow for placing locks on the connections in between the areas: If the zones the areas are different, then the lock protecting the succeeding area must be require a key that is not required by the preceding zone.

Note that there are no conditions that make locks weaker. Having stronger locks protecting a weaker zone creates a barrier or an obstacle. It may not be good to have too many of these, but they are valid in the space. For instance, an area may be protected by a wall, which later is found to have a gate requiring a weaker key. These are what I have called barriers.

When the zones are not known, but the locks and connections are, the rule becomes more complicated. The set of allowable zones may be algorithmically determined by iterating through each of the connections for the area. The initial set of allowable zones is equal to all possible zones. For each connection, there are four conditions:

1. Connection is incoming, and the neighboring zone follows the lock on the connection: Each possible zone must be a successor of the neighboring zone, or equal to the neighboring zone.

2. Connection is incoming, and the neighboring zone does not follow the lock on the connection: Each possible zone must be a predecessor of the neighbor, or equal to it.

3. Connection is outgoing, and the neighbor zone follows the lock: Each possible zone must both succeed the neighbor, and either just follow the lock on the connection, or not follow the lock at all. The possible zone may also be equal to the neighbor zone.

4. Connection is outgoing, and the neighbor zone follows the lock: The possible zones must either precede (or equal) the neighbor zone, or just follow the lock on the connection, or not follow the lock at all.

5. If none of these connections are met, make no changes to the current allowed set of zones.

A zone *follows* a lock if the net keys needed to enter the zone unlock the lock. A zone *just follows* a lock if the zone follows the lock, and none of its parents do.

7.4 Evaluation

Evaluators allow the generation problem to become a selection problem. The purpose of the evaluators is to rank and score possible configurations, excluding ones that are invalid. Evaluators are responsible for encouraging and enforcing the essential properties of key and lock puzzles as described before. Since several facets of simulation work independently, there are categories of evaluators, applying specifically to keys, zones, and tensions.

In my implementation, evaluators are objects and there may be an arbitrary number of them, which is beneficial for many reasons. The best reason for abstracting evaluators is that different types of key and lock behaviors may be encouraged or discouraged. If one wishes to have a game experience that is very sensitive to tension curves or the pacing of keys and locks, this may be done by adjusting parameters on existing evaluators, or excluding some evaluators and including others. The second best reason for abstracting evaluators is that temporary evaluators may be attached to the program for debugging purposes.

There are 12 distinct evaluators used in the implementation. Most of these are meant to enforce the key and lock properties, but there are several others that were added to improve the game flow of the simulation. The purpose of these is to evaluate incomplete networks, and produce scores that will lead to an optimum. The individual evaluators are discussed later.

Because generation occurs in small incremental steps, the simulator does not have the capacity to plan far ahead. Generally, evaluators have competing interests, and work against each other in order to find a "sweet spot" that is optimal amongst all of them. Because the simulator has a limited view of the whole game model, it is necessary to force the evaluators to recognize what will potentially lead to ideal results, not just for the current configuration, but for the whole in the future.

To get the total score for a configuration, the following formula is used:

$$score = \prod_{\substack{evaluator \in \\ relevant\, evaluators}} evaluator(incomplete\, network)^{emphasis}$$

Here, the incomplete network is the current network with the new configuration attached. A common practice in AI weighted selection algorithms is to take the sums of the evaluators to produce the final weight. There are several reasons why this is not the case here. Firstly, when this is done, the results are inclusive. Generally, evaluators always return values in between 0 and 1. This means that the best possible score an evaluator can give a configuration is 1, which has the effect of leaving the weight unchanged. Every value less than 1 makes the configuration that much less likely. In a situation in which they were being summed, a value of zero would leave a score unchanged, making it more inclusive.

The second reason why multiplication is used is that it is invariant under new evaluators. If a new evaluator is introduced that gives everything a score of .5, the multiplied scores would remain proportionally equivalent. If the scores were added, this new evaluator would unbalance the existing scores significantly. With multiplication, many specialized evaluators are introduced that may encourage or discourage specific features in a network.

The evaluators used are as follows:

1. **EncourageNextKey**

   This evaluator scores networks based on the proportion of the number of keys placed versus the number of areas. The proportion is not direct, but the number of areas is raised to a power specified as a parameter. Without this evaluator, maps would be built that had many more areas than were appropriate.

2. **EncourageNextZone**

   This is a special evaluator that ranks maps very highly if they have placed the next zone. There are only two return values, so the return is basically a boolean condition. This evaluator is present so that networks will begin showing new zones sooner than later, rather than leaving many areas of a single zone.

3. **Gateway**

   The Gateway evaluator attempts to make sure that gateways are present connecting new areas. For each new area in the network, the evaluator tests whether the area is accessible with the keys that should be required to access it. The configuration receives a low score if it has many inaccessible areas.

4. **KeyDistances**

   Property 3 (keys are encountered regularly) from earlier is manifested in this evaluator. The evaluator measures the distance in the network from the starting area to each of the next keys that are present. If the distances are too close or too far away, the network is failed (the evaluator returns 0), otherwise it is scored well if the distances are near some given number. The evaluator does this with the path from one key to the next one that is placed, and skips keys that are not yet placed.

   Originally this evaluator was intended to encourage the next key, but it turned out to be ill suited, so that logic was moved into EncourageNextKey.

5. **KeyEffect**

   This evaluator partially reflects property 4 (keys are rewards). The goal of the evaluator is to make sure that all keys have some effect on tension in the network. The evaluator scores well networks where the keys have an effect on tension is close to some given value, otherwise the network receives a lower value (but not zero). This resembles the manner in which the tension effect was measured in the section on the model.

6. **KeyReuse**

   KeyReuse represents property 5 (old keys are still useful). The evaluator checks areas in zones that require a key to enter, and score the network well if there are several locks in which that key must be used to move within those zones. In these cases, the key is not strictly necessary, because it is already required to enter the zone, but it is good to have reuse anyway.

7. **LockPrecedence**

   Property 2 (locks precede keys) is represented here. This evaluator enforces that locks requiring a key must occur before the key is placed. This evaluator fails networks if they do not meet a certain minimum number, and rank highly those networks in which the number of preceding locks is closer to an optimum number. This evaluator competes with EncourageNextKey, and they should be set to work to a good balance.

8. **NewAreas**

   NewAreas enforces property 6 (3 to 10 new areas are opened up with each key). It is similar to LockPrecedence, but its emphasis is on making sure that there are places to use a new key, whether it is placed or not. The current implementation of this evaluator is essentially a binary condition, and scores well those networks which have enough new areas, and scores poorly those that do not.

9. **SeparateZones**

   This evaluator performs a similar function to EncourageNextZone, encouraging not just the existence of one new area of the type in the next zone, but giving a better score if there are more of the areas present. The goal of this is to produce enough new areas that are needed by LockPrecedence and NewAreas.

10. **Stuckness**

    This evaluator attempts to enforce the requirement that it should never be possible for the player to get stuck. Due to efficiency, the current implementation does not perform this for every area, but it does make sure that it is possible to get to each new key.

11. **TensionFitting**

    This evaluator is the second one that represents property 4 (keys are rewards). Specifically, it encourages the tension leading from one key to another to form the sorts of spikes that make good tension curves. The curves should grow and have a spike and then a dip at each key, and grow to a final climax. The evaluator performs a curve fitting test on the values of the tension in moving from one location to another, while taking into account the appropriate keys influence on the area tensions.

12. **ZoneDensity**

    This is an unusual evaluator that was introduced to attempt to balance the maps. The evaluator tests the number of areas belonging to each zone, and scores the network based on the proportion of the number of areas for each zone. Ideally, this evaluator wants every zone to have the same number of areas belonging to it. A perfect balance does not generally happen, but it discourages lopsidedness.

Chapter 8

Conclusion

The original goal of this project was to develop a method for simulating the logic behind key and lock puzzles, as an effort to address the larger problem of procedural gameplay. Procedural gameplay is a large and amorphous research subject, but ultimately it hinges on abstracting and simulating the ways to engage the player. Procedural gameplay relates closely to procedural content, which is a method for generating assets to be instantiated within a game, though the operational rules and the underlying structure of the game do not vary.

Procedural gameplay is important for several reasons. The first is that as game worlds continue to expand, the worlds must be filled gameplay such as quests and missions to engage the player in the large world. The task of building these things is huge, and it is also substantially different from the task of building assets to be placed in the world. The second reason is that there is great ludological value in understanding the nature of gameplay, and the process of simulating it is useful for understanding the strengths and weaknesses of the model used.

The simulation of key and lock puzzles is useful as an experiment in procedural gameplay because of the ease in their computational representation, as well as their rich history in games. The specific games analyzed were the Zelda and Metroid titles, designed by Shigeru Miyamoto. While the simulator I built admittedly does not create games of the same caliber as Miyamoto's, the simulator is able to mimic many of the important design strategies used in his games. This was possible by analyzing the games, closely examining the patterns under which the keys and locks were used, and forming a list of properties that persist in the games.

The most important finding in this research was the interconnection between the simulator and the renderer. The model of games using key and lock puzzles may be represented very abstractly with areas and connections, but abstract areas may not always be representable within a game space. The renderer and the simulator must communicate in order to build the game model. Additionally, while many of the design strategies that occur in these games may be simulated, some other properties are simply impossible to simulate directly. These properties may be instead expressed within the renderer.

There are many small places in this work that are open for further expansion. These range from small details, such as creating key and lock systems in which multiple keys may be used to open single locks, or creating additional evaluators to improve management of zones in new areas, or improving the manner in which tension is handled and represented within the renderer. These expansions can definitely improve the content of the research

presented here, however, as it stands, the matter of generation of key and lock puzzles has been successful. The representation of keys and locks within the project is abstract enough to be applied towards other procedurally generated spaces, which would be one good direction to take this work. Possibly the most useful direction to continue with research would be in how to reconfigure existing game spaces to make them continually enjoyable, as this would have very practical application in future games.

Chapter 9

Screenshots and Results

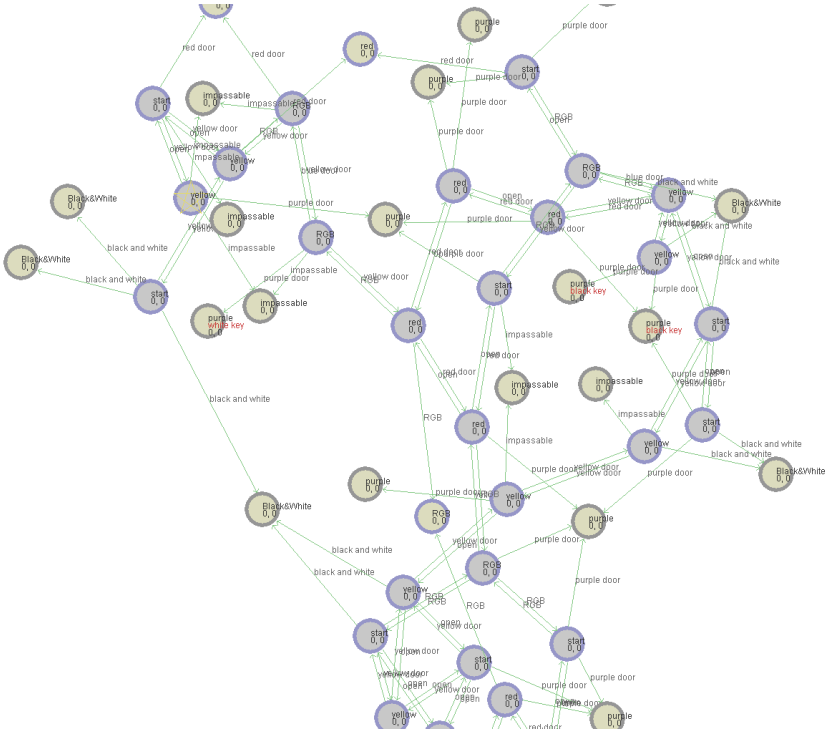


Figure 9.1: Early implementation of the model, before the properties were expressed. This is essentially a direct method of building a model, which will produce areas, but they neither valid nor renderable.

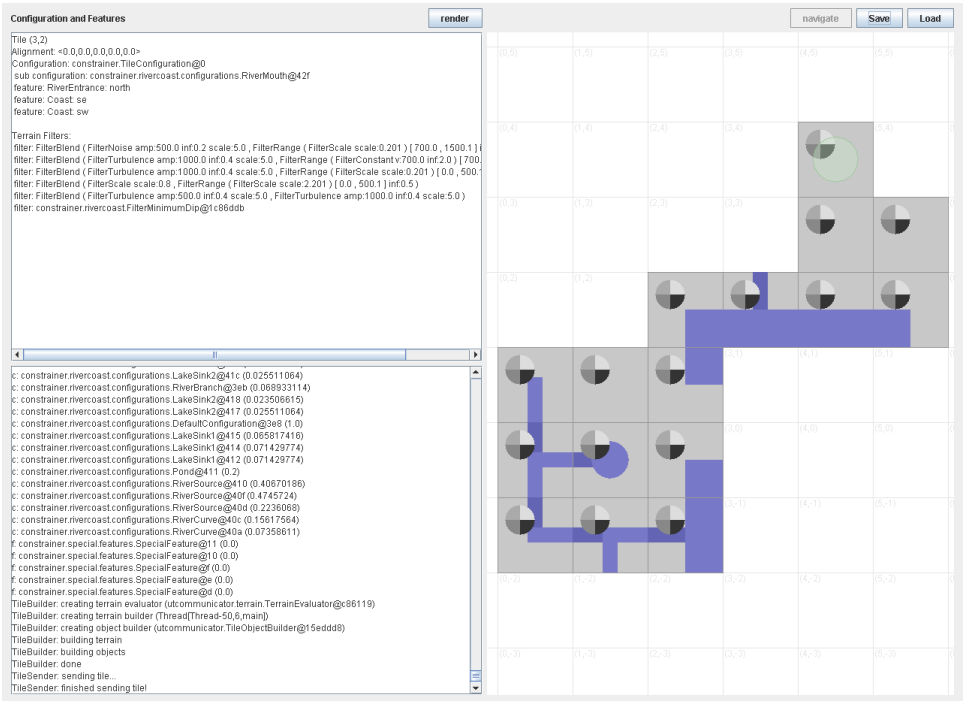


Figure 9.2: An early view of Charbitat, before the key and lock dynamics were applied. The gray circles in each of the tiles are used to represent the elemental alignment from the Charbitat game.

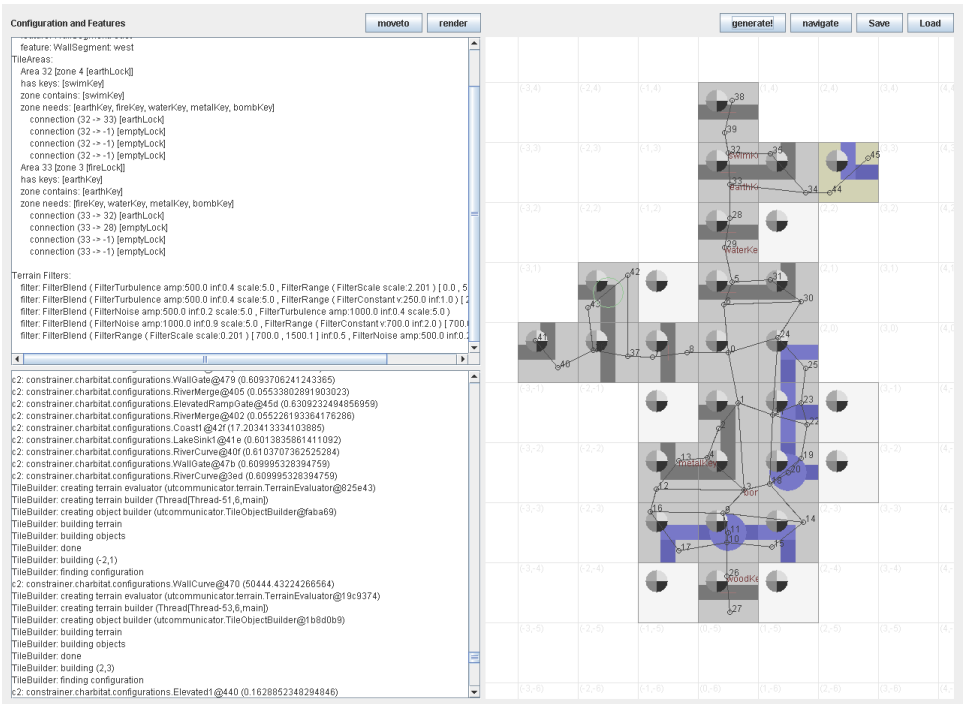


Figure 9.3: The renderer and simulator working together. Both the areas and the tiles can be seen here. The keys are visible in red.

Figure 9.4: This is a scene within the actual game. There is a key in the distance with a particle system emerging from it. Beyond that, to the right, there is a wall with a gate in it requiring that key.

Bibliography

[1] Greg Costikyan. I have no words and i must design. *Interactive Fantasy*, 2, 1994.

[2] Chris Crawford. *The Art of Computer Game Design*. Osborne/McGraw-Hill, 1982.

[3] Laura Ermi and Franz Mäyrä. Fundamental components of the gameplay experience: Analysing immersion. In *DIGRA*. DIGRA, May 2005.

[4] Syd Field. *Screenplay: The foundations of screenwriting*. Dell, 1984.

[5] Gonzalo Frasca. Ludologists love stories, too: Notes from a debate that never took place. In *Level Up*. Level Up, November 2003.

[6] Stefan M. Grünvogel. Formal models and game design. *Game Studies*, 5(1), 2005.

[7] Clara Fernández-Vara Brian Hochhalter Nolan Lichti José P. Zagal, Michael Mateas. In *Towards an Ontological Language for Game Analysis*. DIGRA, May 2005.

[8] Jesper Juul. *A Clash between Game and Narrative: A Thesis on Computer Games and Interactivity*. PhD thesis, University of Copenhagen, 2001.

[9] Jesper Juul. The game, the player, the world: Looking for a heart of gameness. In *Level Up*. Level Up, November 2003. Keynote presented at the Level Up conference in Utrecht, November 4th-6th 2003.

[10] Kejardon. The 14%/15% walkthrough for super metroid. http://db.gamefaqs.com/console/snes/file/super_metroid_15.txt, 2003.

[11] Rune Klevjer. Computer game aesthetics and media studies. In *15th Nordic Conference on Media and Communication*. 15th Nordic Conference on Media and Communication, 2001.

[12] Kevin Lynch. *The Image of the City*. MIT Press, 1960.

[13] Lev Manovich. Database as a symbolic form. *Convergence: London*, 1999.

[14] Micahel Mateas and Andrew Stern. A behavior language for story-based believable agents. *IEEE Intelligent Systems*, 2002.

[15] Michael Mateas. *Interactive Drama, Art, and Artificial Intelligence*. PhD thesis, Carnegie Mellon University, 2002.

[16] Janet Murray. *Hamlet on the Holodeck*. MIT Press, 1998.

[17] S Natkin and L Vega. Modeling for the analysis of the ordering of actions in computer games. In *Game On*, 2003.

[18] Katie Salen and Eric Zimmerman. *Rules of Play: Game Design Fundamentals*. MIT Press, 2003.

[19] Mark Saltzman. *Game Design: Secrets of the Sages (Second Edition)*. Bradygames, 2000.

[20] Joseph Weizenbaum. Eliza – a computer program for the study of natural language communication between man and machine. *Communications of the Association for Computing Machinery*, 1966.

Ludography

[M1] Nintendo. *Metroid*. Nintendo Entertainment System, 1986

[M2] Nintendo. *Super Metroid*. Super Nintendo Entertainment System, 1994

[M3] Nintendo. *Metroid Prime*. Nintendo Gamecube, 2002

[Z1] Nintendo. *The Legend of Zelda*. Nintendo Entertainment System, 1987

[Z2] Nintendo. *The Legend of Zelda: A Link to the Past*. Super Nintendo Entertainment System, 1992

[Z3] Nintendo. *The Legend of Zelda: The Ocarina of Time*. Nintendo 64, 1998

[Z4] Nintendo. *The Legend of Zelda: Majora's Mask*. Nintendo 64, 2000

[DL] Don Bluth, Cinematronics. *Dragon's Lair*. Laserdisk, 1983

[RO] Epyx, Inc *Rogue*. Amiga, Atari 8-bit, Commodore 64, DOS, 1986

[NH] http://www.nethack.org/ *Nethack*. Multiplatform, 2003 (ongoing)

[MY] Cyan Worlds, Inc. *Myst*. Multiplatform, 1995