Learning from Games for Generative Purposes

Adam Summerville

November 2, 2017

Introduction

PCG has been a part of computer games for the majority of their existence (Beneath Apple Manor - 1978). For most of this time, generation has required human designers to encode all of the rules governing the generation (achieved through many different formalisms: imperative code, declarative code, grammars, evolutionary geno/phenotypes, fitness criterions, search). This is a difficult endeavor for a number of reasons:

1. For a designer in process, design decisions are rarely accurately accessible via introspection (*?, ?*)

2. Performing a critical deep dive into a piece of work to access the design decisions is not something done lightly, making up entire fields of study (e.g. Digital Humanities)

3. Formalizing all design decisions is difficult, as humans often have hidden biases / foreseeing all possible outcomes of an algorithm is impossible in general (see decidability) and is difficult in specific (see the fields of automated testing/code verification)

This is not to say that it can not be done, as, obviously, people have done this. But it is to say that in general it is a hard, arduous endeavor that can still lead to feelings of "oatmealishness" (*?, ?*), i.e. content that while different in countless different ways, still feels the same.

However, the design decisions to generate a piece of content do not need to be exhaustively encoded, they can be learned. The design of an artifact is latent within the artifact itself. The goal of this work is to learn that latent design, so as to be able to generate perceptually similar artifacts, and to do so from observation, with minimal human supervision.

Research Questions and Contributions

To this end I propose a body of work that will look to answer the following questions:

RQ1: How can a machine learn the important mechanical properties of the content?

Currently, techniques such as Generative Adversarial Networks (GANs) or Variational AutoEncoders (VAEs) are popular techniques for image generation. Certainly, it would be possible to supply images of game levels (these exist in abundance on the internet **CITE VGATLAS**), but visual signifiers are only one portion of the semiotics of a game. The mechanical

properties of the game entities are at least equally as important (and certainly visuals can deceive, as two visually identical entities might have different properties). While image based techniques might be able to generate visually acceptable levels, without a way of encoding these properties they are going to lack the expressivity of a generator which has access to such properties. Previous machine learned approaches have either operated on the visual level or with designer specified mechanical properties. I aim to answer the question of how to learn the mechanical properties in a way that enables down stream use (e.g., for a generator).

RQ2: How should content be represented such that it can be efficiently learned for generation?

Most uses of machine learned generation have focused on two types of data: text and images. Both have standard structures and associated machine learning techniques (e.g. text as a sequence handled by Long Short-Term Memory Recurrent Neural Networks and images as RGB Tensors handled via GANs or VAEs). Game levels have some features similar to both, but are often composed of multiple pieces of arbitrary shape and size. Behaviors and rules have no commonly agreed upon format. I aim to answer the question of how to best represent game content for the sake of efficient training and generation.

RQ3: How do we assess whether a learned generator has learned to generate the desired content?

Evaluation of PCG for games has largely focused on examining a few hand-picked generated artifacts or on visually analyzing a histogram of the expressive range of a generator. The former is prone to cherry picking, leaving one to wonder what the expressive capabilities of the system actually are, while the latter is rarely tied back to an intuitive understanding (e.g. Is it good or bad to have a wide histogram? What does it mean for certain portions of the space to be under or over represented? Are the dimensions shown actually worthwhile?). For a classical PCG system these evaluations are perhaps unsatisfying, but given that authors do not necessarily make any claims about the generative space of the systems, they are not mission-critical. On the other hand, machine learned PCG systems make the claim that they are accurately learning the design space of the original training data, so it is necessary to show that the generators are able to generate similar content while not memorizing the training data.

RQ4: Can higher level properties of games be learned?

Most work on machine-learning of game content has focused on learning game level content. What little other work exists has still learned fairly low-level mechanics (i.e., hybrid automata behaviors and state transition rules). Despite the ability for games to convey great meaning, and furthermore, despite the presence of game generators that intend to generate games with meaning, there has been very little research into understanding the intended (or unintended) meaning of a game. Can a machine learning system learn to infer the meaning of a game? Can this be utilized for the generation of games targeting a specific meaning?

The primary contributions of this research are three-fold:

1. The development of techniques to learn the properties of game content automatically from observation

2. The development of formalisms for representing rules, entities, and levels in a manner amenable to the training of machine learning systems and then generating new content

3. The development of techniques for evaluating machine learned generators

Which represent contributions to the fields of machine learning, procedural content generation, and automated game design research.

We will address these questions and contributions as they relate to three different modes of procedural content generation:

- Level generation
- Behavior generation
- Game generation

with each mode of content being addressed separately. The rest of the document will detail these three modes, the related work by others for that mode, my prior work, and my proposed work for that mode.

Procedural Generation of Levels

Level generation has been the most popular avenue of procedural content generation for games, since PCG's inception. The vast majority of work has been centered around human authored rules and constraints and only recently has machine learning been applied to the task of learning from levels so as to generate levels with similar properties. My work on level generation addresses:

- **RQ1** – The use of *Mappy* to learn mechanical properties from observation which in turn can be used by *Kamek* and *Agahnim*
- **RQ2** – Explorations in representation for *Kamek* and *Agahnim* to find what representations allow for efficient learning
- **RQ3** – The development of new techniques for comparing generators to their exemplar data

Related Work

The procedural generation of level content has been popular in commercial, hobbyist, and academic circles with recent years seeing increasing amounts of interest in procedural generation. A game jam devoted to procedural content, PROCJAM, had 99 games submitted in 2015 and nearly double, 173, in 2016 (***?, ?***). While PCG has focused on a number of different game aspects such as characters (***?***), narrative (*?*), levels have been a major area of focus.

Level Generation - Human Design Approaches

Platformer levels have represented one of the most fecund areas of research, particularly levels for *Super Mario Bros.* As the progenitor of the platformer genre, as well as one of the most popular games of all time, *SMB* has been the most common game for level generation. A large number of different approaches have been taken for *SMB* level generation.

Many approaches have utilized hand authored level pieces and rule systems to generate levels. The Probabilistic Multi-Pass (ProMP) generator by Ben Weber (***?***) creates a base level and then iterates through it adding new components on each pass. Components have a probability

for being added, but the probabilities are tuned based on other constraints (e.g. if the maximum number of gaps has been reached, the probability for a new gap is set to 0). The Occupancy-Regulated Extension (ORE) approach of Mawhorter (*?*) places hand-authored pieces to generate whole levels. Each piece was annotated with anchor points which represent places where other pieces may be placed. The system by Baumgarten (*?*) uses linear discriminant analysis to provide a low-dimensional embedding of player skill and then uses that value to determine which hand-crafted level pieces to place to generate a complete level.

Hopper by Takahashi and Smith (*?*) attempts to generate levels that imitate the style of *Super Mario World*. Hopper uses rules to probabilistically place pieces, the weights of which are manually tuned based on inferred player type and skill level. Tanagra (*?*) is only Mario adjacent, creating levels for a platformer with similar components to that of Super Mario. Tanagra uses a rhythm-based framework to generate "beats" of player action in varying rhythms. These beats may take the form of gaps the player must jump over or an enemy the player must defeat. Tanagra uses reactive planning to choose beats, create the corresponding level segments, react to user input, and generate playability constraints. The playability constraints are then solved in a second pass by a constraint solver which attempts to prevent unplayable levels from being created. Tanagra can work solely as a generator, or it can work in conjunction with a human designer as a co-creative assistant. The human can design level segments which Tanagra will incorporate in its generation, or the human can author at a more abstract beat level which Tanagra can then reify.

The approach of Shimizu and Hashiyama (*?*) utilized offline interactive evolutionary computation to generate candidate level pieces and then placed those based on player modeling that attempted to classify how likely players were to break blocks, collect coins, and defeat enemies. The approach of Sorenson and Pasquier (*?*) uses evolutionary computation with an objective that rewards alternating periods of high and low challenge, with the goal of keeping player's engaged but not overwhelmed.

While Mario-like platformers have been the dominant testbed for procedural level generation, there has been some interest in generating "dungeons" for games like *The Legend of Zelda*. While platformers are 2 dimensional games viewed from the side, Zelda-likes have a top-down perspective and the player generally has much more freedom to explore the entirety of the level.

The system of Valls-Vargas, et al. (*?*) performs an exhaustive breadth-first search to find all possible mission spaces for an input set of plot points. Once all possible missions have been constructed, they are evaluated for fitness based on how closely the mission matches the desired properties set forth by the designer. This approach will find the best solution, as defined by the evaluation criteria, but can only handle missions of a limited size due to the exhaustive nature of the search.

Grammar based methods are not guaranteed to find an optimal solution, but their limited expansion rules direct the search to more desirable regions of the possibility space much more efficiently than an breadth first search. Joris Dormans's system (*?*) uses grammars for both the mission and level spaces. The mission space is constructed by a graph grammar where nodes in the graph make up player actions such as "Defeat Boss" or "Unlock Door". From a starting

seed, production rules are applied pseudo-randomly until all non-terminal nodes have been converted to terminal nodes.Once the mission space of the level has been constructed, a shape grammar is applied on the mission graph. The first node of the mission graph is taken and the system looks for a shape grammar rule that satisfies the mission node symbol, finds a location where the rule could be applied and applies it. A similar grammar based system was developed by van der Linden, et al. (*?*) that uses grammars for both mission and level generation as well. The advantage to van der Linden's work is that multiple mission actions can occur at the same location, e.g. kill a dragon and get dragon's treasure could both occur at the same location. The grammar based methods have the advantage of being intuitive, but unless the grammar is heavily constrained it can be hard to guarantee that the levels produced match designer criteria while retaining uniqueness.

Genetic algorithm solutions have the advantage of being able to find a wide range of optimal solutions; However, they come with no guarantees that the optima found are global or that they will terminate in any amount of time. The key factors that determine the type of level created by a genetic system are the genotype and phenotype representations and the evaluation metric. The system developed by Sorenson and Pasquier (*?*) has direct genotype to phenotype mapping as elements in the phenotype equate to the placement of elements in the level. Their evaluation metric seeks to keep interactions along the optimal path varied. To keep genetic diversity while also ensuring the validity of produced levels, they used a Feasible-Infeasible Two-Population split. By considering only the positioning of game elements, their system considers the mission space tangentially, by assuming that a good level space will lead to a good mission space. Valtchanov and Brown (*?*) created a system that uses a tree based genotype to handle room to room placement. Their fitness evaluation rewards novelty and room interconnectedness, but only considers player experience by guaranteeing the maximum optimal path length. Hartsook, et al. (*?*) used a similar tree based genome, but their fitness function considers the distance between critical nodes, length of sidepaths, average number of sidepaths per non-critical node, total number of sidepaths, total number of nodes, and environmental transitional probabilities.

While the genetic algorithms can produce a variety of levels, they have no guarantees about content produced. At the opposite end of the spectrum are constraint based solvers that provably produce levels that meet the designer's desires. In these systems, the designer sets up a system of logical properties and relations and the solver finds solutions that match the designer's criteria. Horswill and Foged (*?*) developed a constraint solver that uses constraint propagation to place items within a level to meet a desired player experience, but it requires a level annotated with the desired player path.

The above approaches all rely on human design intuition at some level. Some stitch together human-authored snippets (Baumgarten's, ORE, ProMP), while others use human authored rules to generate level geometry (Tanagra, Hopper), while others utilize human design intuition to score evolved content (Sorenson and Pasquier, Shimizu and Hashiyama), and some rely on human design intuition about playability (Horswill and Foget). More recently there has been interest in generators that do not require humans to author at the rule or design level and instead utilize machine learning techniques to generate levels.

Level Generation -Machine Learned Approaches

Dahlskog et al. trained n-gram models on the levels of the original *Super Mario Bros.* game, and used these models to generate new levels (*?*). As n-gram models are fundamentally one-dimensional, these levels needed to be converted to strings in order for n-grams to be applicable. This was done through dividing the levels into vertical "slices," where most slices recur many times throughout the level (*?*). This representational trick is dependent on there being a large amount of redundancy in the level design, something that is true in many games. Models were trained using various levels of n, and it was observed that while $n = 0$ create essentially random structures and $n = 1$ create barely playable levels, $n = 2$ and $n = 3$ create rather well-shaped levels.

In (*?*) Snodgrass and Ontañón present an approach to level generation using multi-dimensional Markov chains (MdMCs) (*?*). An MdMC differs from a standard Markov chain in that it allows for dependencies in multiple directions and from multiple states, whereas a standard Markov chain only allows for dependence on the previous state alone. In their work, Snodgrass and Ontañón represent video game levels as 2-D arrays of tiles representing features in the levels. For example, in *Super Mario Bros.* they use tiles representing the ground, enemies, and *?*-blocks, etc. These tile types are used as the states in the MdMC. That is, the type of the next tile is dependent upon the types of surrounding tiles, given the network structure of the MdMC (i.e., the states that the current state's value depends on).

In order to generate levels, first the model must be trained. They train an MdMC by building a probability table according to the frequency of the tiles in training data, given the network structure of the MdMC, the set of training levels, and the set of tile types. A new level is then sampled one tile at a time by probabilistically choosing the next tile based upon the types of the previous tiles and the learned probability table.

In addition to their standard MdMC approach, Snodgrass and Ontañón have explored hierarchical (*?*) and constrained (*?*) extensions to MdMCs in order to capture higher level structures and ensure usability of the sampled levels, respectively. They have also developed a Markov random field approach (MRF) (*?*) that performed better than the standard MdMC model in *Kid Icarus*, a domain where platform placement is pivotal to playability.

Guzdial and Riedl utilized gameplay video of individuals playing through *Super Mario Bros.* to generate new levels. They accomplished this by parsing each *Super Mario Bros.* gameplay video frame-by-frame with OpenCV (*?*) and a fan-authored spritesheet, a collection of each image that appears in the game (referred to as sprites). Individual parsed frames could then combine to form chunks of level geometry, which served as the input to the model construction process. In total Guzdial and Riedl made use of nine gameplay videos for their work with *Super Mario Bros.*, roughly 4 hours of gameplay in total.

Guzdial and Riedl's model structure was adapted from (*?*), a graph structure meant to encode styles of shapes and their probabilistic relationships. The shapes in this case refer to collections of identical sprites tiled over space in different configurations. For further details please see (*?*), but it can be understood as a learned shape grammar, identifying individual shapes and

probabilistic rules on how to combine them. To train this model Guzdial and Riedl used a hierarchical clustering approach utilizing K-means clustering with automatic K estimation. First the chunks of level geometry were clustered to derive styles of level chunks, then the shapes within that chunk were clustered again to derive styles of shapes, and lastly the styles of shapes were clustered to determine how they could be combined to form novel chunks of level. After this point generating a new level requires generating novel level chunks in sequences derived from the gameplay videos. A similar graph approach was used by Londoño and Missura (***?***), but learned a graph grammar instead. Their work is unique in platformer level generation in that they used semantic connections such as tile X is reachable from tile Y instead of just physical connections.

Shaker and Abou-Zleikha (***?***) use Non-Negative Matrix Factorization (NNMF) to learn patterns from 5 unique non-ML-based unique generators of *Super Mario Bros.* levels (Notch, Parameterized, Grammatical Evolution, Launchpad, and Hopper), to create a wider variety of content than any of the single methods. To their knowledge, this is the first time that NNMF has been used to generate content. Because NNMF requires only non-negative values in its matrix decomposition, the basis representations are localized and purely additive, and they are therefore more intuitive to analyze than PCA.

Their system begins by generating 1000 levels of length 200 with each of the $G = 5$ unique generators. These 5000 levels in the training set are represented as vectors recording the locations or amount of $T = 5$ content types at each level column: Platforms, Hills, Gaps, Items, and Enemies. These vectors are combined into T matrices, one for each level content type. The algorithm then uses a multiplicative update NNMF algorithm (***?***) to factor the data matrices into T approximate "part matrices" which represent level patterns and T coefficient matrices corresponding to weights for each pattern that appear in the training set. The part matrices can be examined to see what types of patterns appear globally and uniquely in different generators, and multiplying these part matrices by novel coefficient vectors can be used to generate new levels.

Prior Work

My work in level generation has been on two systems now given the names *Kamek* and *Agahnim*.

Agahnim

Agahnim is level generator designed to generate levels at two different scales, the room-to-room topology of a level and also the in-room tile level construction. The room-to-room topology is generated using a Bayesian Network (BN) and the in-room tile level utilizes dimensionality reduction via Principal Component Analysis. To train the BN we annotated existing ARPG levels to be able to extract the relevant features needed. To do this, we used level images from

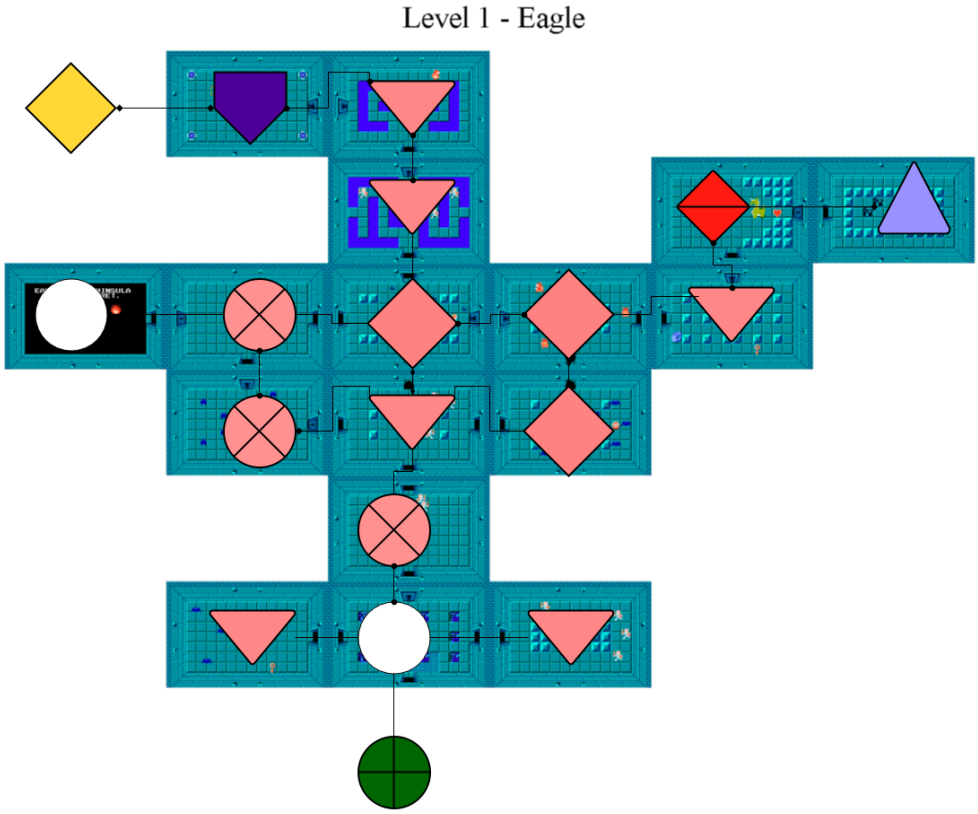


Figure 1: Annotations for the first level from *The Legend of Zelda*. Green Circle with Plus is the start point. Light Blue Upwards Triangle is the end point. Pink nodes (Diamond, Downwards Triangle, and Circle with X) contain enemies. Downward Triangle nodes contain keys. Diamond nodes contain items. Purple Pentagon nodes contain puzzles. The Red Diamond with horizontal line node contains the boss. The Yellow Diamond off the map shows that it warps to another room

three different ARPGs from the Legend of Zelda series (Fig. 1). The Legend of Zelda is the progenitor of the genre as well as the genre's most popular series, and, furthermore, is the most prolific series of ARPGs with 17 titles over the course of 28 years. We have used the levels from *The Legend of Zelda*, *The Legend of Zelda: Link to the Past*, and *The Legend of Zelda: Link's Awakening*. We manually annotated 38 levels from the three games and held out 4 levels for our test set. The held out set contained one level from *The Legend of Zelda: Link to the Past* and *The Legend of Zelda: Link's Awakening* each, and two levels from *The Legend of Zelda*. The 34 levels that the model was trained on were composed of 1031 rooms in total, which was the final size of our training set. To annotate these levels, we used images of the levels that show the physical structure of the levels as well as the placement of enemies, items, puzzles, and traps, allowing us to see the full structure of the levels (*?, ?, ?, ?*). We then annotate the images to turn them into the graph topology that makes up the *physical space* of the level, where each room

is represented as a node in the graph. Nodes (rooms) are annotated by what types of objects it contains: *Start*, *Enemies*, *Puzzles*, *Beneficial Items*, *Keys*, *Big Key*, *Key Item*, *Boss Enemy*, *End*. A room can contain any number of these elements, although in practice only contain up to 4 of these types of objects. The connections between rooms are annotated with one of the following directed link types: *Door*, *Bombable Wall*, *Locked Door*, *Soft-Locked Door*, *Big Key Locked Door*, *Key Item Locked Door*, *One Way Door*, *Can See The Other Room*. The *Can See The Other Room* feature is obviously not a standard door but encapsulates an important concept in ARPGs, mainly that the player is often made aware of where they need to go by getting glimpses of another room, but there is no passable edge between their current room and that room. A sample annotated level can be seen in figure 1.

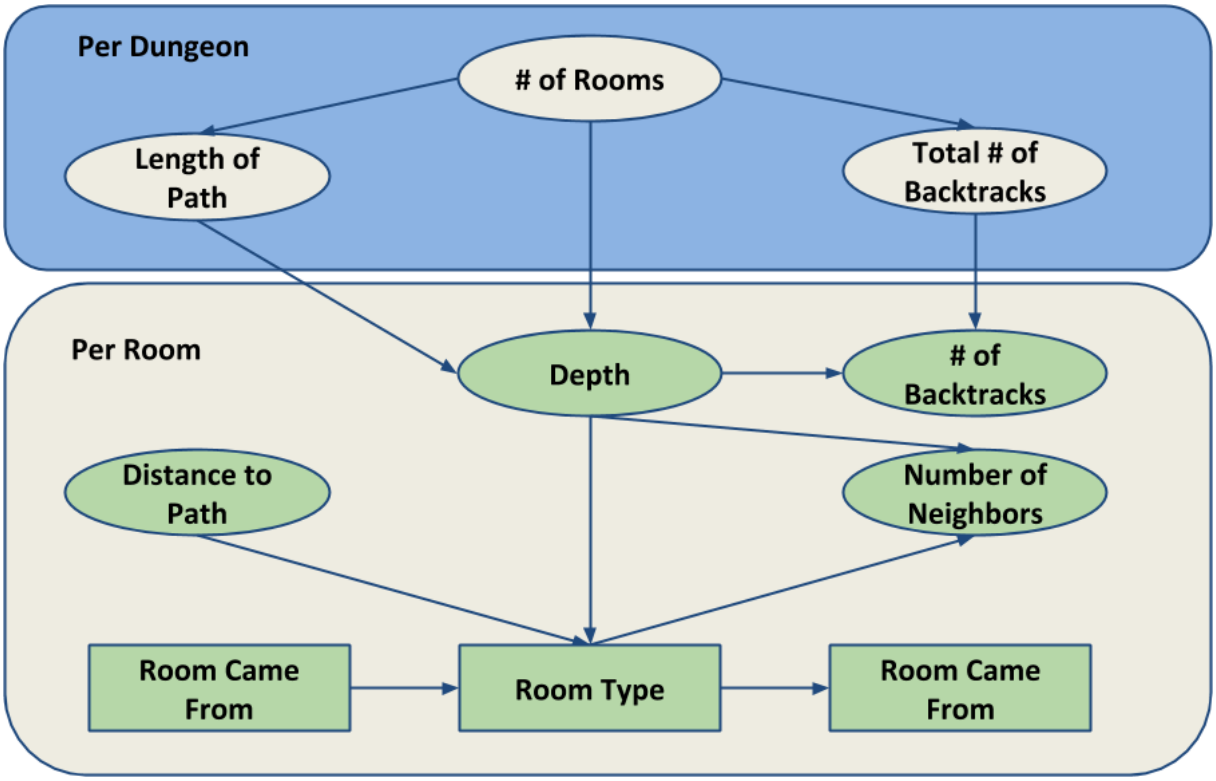


Figure 2

The trained BN captured a variety of high level features, such as number of rooms in the level and length of optimal path through the level, along with low level features such as room-to-room connections and room types. At generation time a designer can set whatever they want (or rather *observe* in the Bayesian parlance) and the system will work with that and infer the rest. e.g. a designer could observe the size of the level, the general make up of rooms, etc.. The system could even be used in a mixed initiative manner with a designer laying out a set of rooms and allowing the system to fill in the rest. The learned network was composed of the following features:

- *Number of Rooms in Level*
- *Number of Doors in Level*
- *Number of Path Crossings in Level*
- *Length of Optimal Path*
- *Distance of Room to Optimal Path*
- *Path Depth of Room in Level*
- *Distance From Entrance*
- *Room Type*
- *Previous Room Type*
- *Door Type from Previous Room*
- *Next Room Type*
- *Door Type to Next Room*

Once the global level parameters have either been observed or inferred, the dungeon is built up from a seed room. The initial entry point to every dungeon is a special *Start* room. *Start* rooms only ever have 1 neighbor, and it is this first neighbor room that is inferred. The room type, the incoming door type, and the number of neighbors are all inferred, given the prior room type (initially the *Start* room), the traversed door type (initially a regular door), its depth in the dungeon (initially one), and all of the inferred or specified global parameters. During this process there is a simultaneous grid embedding process that happens in parallel. The *Start* room is always placed at $(0, -1)$ and its child room is always placed at $(0, 0)$. Each new child room is placed at an open neighboring place on the grid, and added to a list of rooms to be inferred. Once all of the specified number of rooms have been placed, a second cleaning process guarantees that the level is playable. dfigure

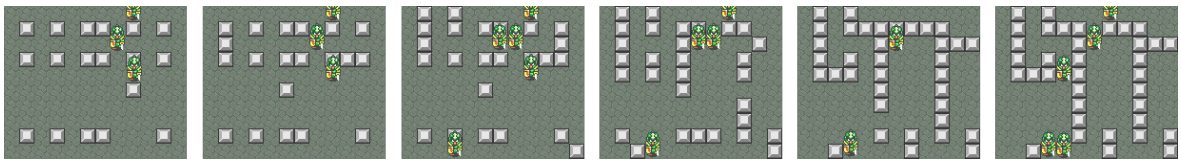


Figure 3: Interpolation between two human authored rooms. The rooms on the ends are human authored while the rooms in between represents steps between them of 20%.

For a level to be playable, all of the rooms must be accessible, the number of key locked doors must equal the number of placed keys, all keys must be used to complete the level, all

keys must be in front of the doors they lock, and there must be a special item room, a boss room, and an end room. To resolve any of these constraints, we utilize the same machinery that tested the validity of the algorithm, the log-likelihood of a specific event occurring. For example, if there was no end room placed, we simply iterate over all rooms, find the one that has the highest likelihood of being an end room, and change it to an end room. We do this for the boss and special item rooms, in addition to the key locked doors. Once these have been taken care of, we then find the optimal path through the dungeon in order to guarantee that the optimal path through the dungeon uses all of the placed keys. If the level is not completable, then we know that the player is unable to reach a required key. This can be resolved in one of two ways, **1)** Move a key that has not been reached to a room that has been reached, or **2)** Move a locked door that has been reached to a door that has not been reached. The choice of how to resolve the violation is chosen at random. On the other hand, if the level is completable, but not all keys are used, then a key-locked door is placed unnecessarily. To resolve this, an unseen key locked door is moved to a seen unlocked door. No matter what constraint violation is being resolved, it is handled with the same machinery seen above, e.g. if a key needs to be moved, then the existing key room that is least likely to be a key room is removed, and the room it is moved to is the one most likely to be a key room.

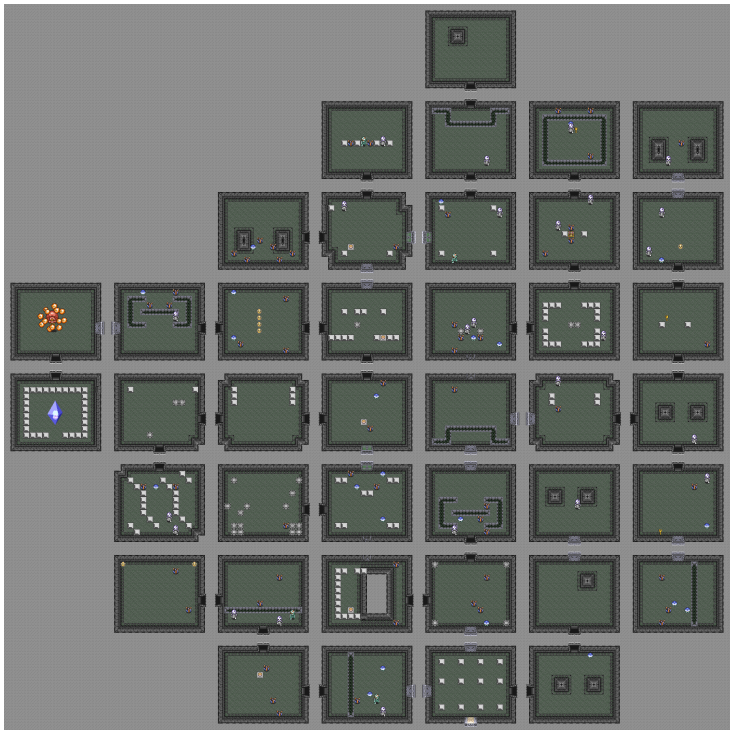


Figure 4: A generated level with the most likely high level parameters.

With the level topology constructed we need to generate the rooms at the tile level to have a playable level. To generate rooms we first performed Principal Component Analaysis (PCA) on

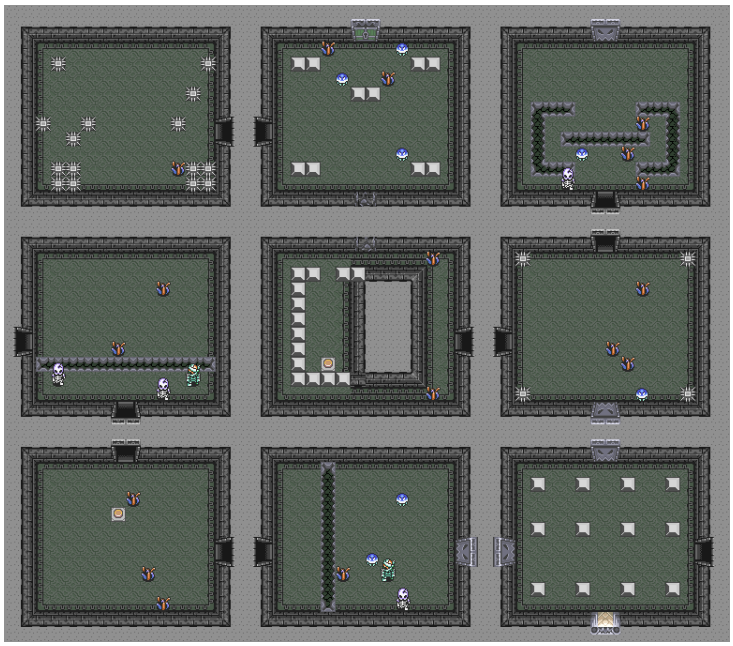


Figure 5: A zoomed in subset, showing the entry point (bottom right corner) and 8 other generated rooms..

a tile level grid representation of the rooms. The dataset for this work was 488 rooms from the three aforementioned games. However, to increase the size of the sample space we also used all mirrorings (up-down, left-right, both) to quadruple the size of our dataset. Each room can be thought of as a tensor of width by height by tile type, and this tensor is then raveled t make a 1-dimensional vector of size width $\times$ height $\times$ tile type. All 1952 rooms are converted to this vector format and stacked together to form a 1952 by width $\times$ height $\times$ tile type. PCA is then applied to this matrix to find a lower dimensionality representation, 20 dimensions in the case covering 95% of the variance within the data. To generate a room, existing vectors in this 20 dimensional space are chosen based on the room types (e.g. if a room containing enemies must be generated, 2 vectors corresponding to enemy rooms are chosen) and then randomly interpolated between. The results of this interpolation can be seen in figure 3. A generated level can be seen in figure 4, along with a zoomed subset in figure 5.

Kamek

Kamek is a tile-based level generator that, to this point, has been trained on human annotated levels, as in the work of Snodgrass and Ontañón and Dahlskog et al.

Kamek uses Long Short-Term Memory Recurrent Neural Networks (LSTM RNNs) to generate levels as a sequence of tiles. LSTMs are a neural network topology first proposed by Hochreiter and Shmidhuber (*?*) for the purposes of eliminating the vanishing gradient prob-

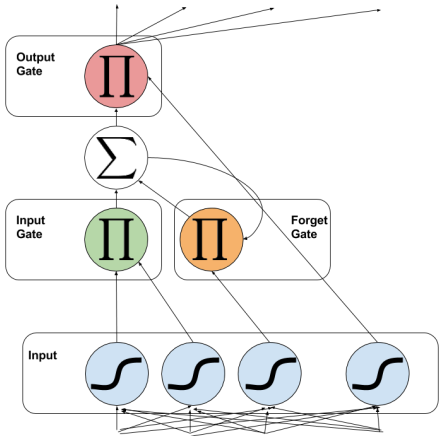


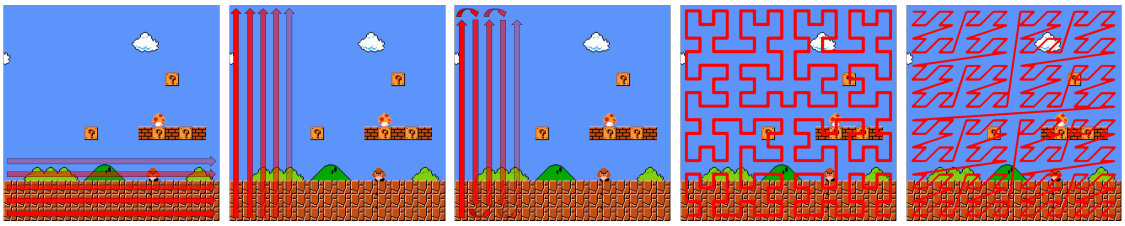
Figure 6: Graphical depiction of an LSTM block.

lem. LSTMs work to solve that problem by introducing additional nodes that act as a memory mechanism, telling the network when to remember and when to forget. A standard LSTM architecture can be seen in figure 6. At the bottom are nodes that operate as in a standard neural network, i.e. inputs come in, are multiplied by weights, summed, and that is passed through some sort of non-linear function (most commonly a sigmoid function such as the hyperbolic tangent) as signified by the S-shaped function. The nodes with $\sum$ simply sum their inputs with no non-linear activation, and the nodes with $\prod$ simply take the product of their inputs. With that in mind, the left-most node on the bottom can be thought of as the *input* to an LSTM block (although for all purposes it is interchangeable with the node second from the left). The node second from the left is typically thought of as the *input gate*, since it is multiplied with the input, allowing the input through when it is close to 1, and not allowing the input in when it is close to 0. Similarly, the right-most node on the bottom acts as a corresponding *output gate*, determining when the value of the LSTM block should be output to higher layers. The node with the $\sum$ acts as the *memory*, summing linearly and as such not decaying through time. It feeds back on itself by being multiplied with the second from the right node, which acts as the *forget gate*, telling the memory layer when it should drop whatever it was storing.

These LSTM blocks can be composed in multiple layers with multiple LSTM blocks per layer, and for this work we used 3 internal layers, each consisting of 512 LSTM blocks. The input layer to the network consists of a One-Hot encoding where each tile has a unique binary flag which is set to 1 if the tile is selected and all others are 0. The final LSTM layer goes to a SoftMax layer, which acts as a Categorical probability distribution for the One-Hot encoding.

LSTM's require a 1-dimensional sequence of inputs, but levels in *Super Mario Bros.* are 2-dimensional. Converting the 2-dimensional grid to a 1-dimensional sequence requires some form of space-filling curve and we considered 5:

- *Horizontal* - The ordering progresses from left-to-right and then bottom-to-top.



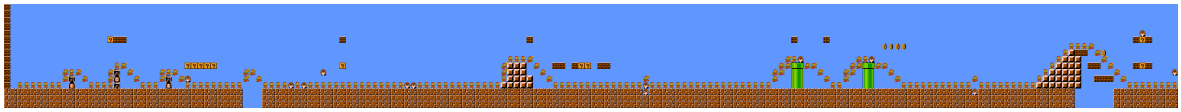
(a) Horizontal (b) Vertical (c) Snaking (d) Hilbert curve (e) Lebesgue curve

Figure 7: The 1-dimensional orderings considered.

- *Vertical* - The ordering goes from bottom-to-top and then left-to-right
- *Snaking* - The ordering goes from either top-to-bottom or bottom-to-top and when it progresses to the next column (left-to-right) it then reverses (e.g. column 1 is top-to-bottom and then column 2 is bottom-to-top)
- *Hilbert curve* - The Hilbert curve is a fractal space-filling curve (**?**). For this work we used a curve with fractal dimension of 4 (i.e. 16×16). When a curve ends (if the origin at bottom-left is $< 0, 0 >$ the end point is $< 15, 0 >$) it progresses into another curve (e.g. if the first curve began at the bottom left, the next curve would begin at $< 16, 0 >$)
- *Lebesgue curve* - The Lebesgue curve is another fractal space-filling curve (**?**). Again, we used a curve with a fractal dimension of 4 (i.e. 16×16)

which can be seen in figure 7. We also included meta-information in the inputs beyond just the level geometry. We also considered "depth" information, i.e. how deep into the level the geometry was. As has been alluded to in some of the human-authored rules, Mario levels often have a progression and flow, starting with low-intensity safe areas, progressing through alternating segments of high and low intensity with a final, end-of-level structure that the player must climb. By including depth information we were able to help the network learn these long-term structures. We also included path information from a simulated A* agent, causing the generator to learn not just how to generate levels, but to also generate exemplar player paths through the level. Since all training levels had solid, unbroken paths, the generated also had such paths, thereby drastically increasing the percentage of playable levels. Each network was trained on 15 levels from *Super Mario Bros.* and 24 levels from the *Japanese Super Mario Bros. 2*for a total of 39 levels. Each network then generated 4000 levels. These levels were then evaluated for whether they were completable. We also wanted each generator to produce unique levels, and not just memorize from the training set. We found that the **Vertical-Depth-Paths** and **Snaking-Depth-Paths** generators performed the best with 93.30% and 96.63% completability rates for the generated level. In comparison, the highest reported playability results for Snodgrass and Ontañón were 66% (**?**) and the highest reported for a human-authored system is 94% for the

(a) Level 1-1 from *Super Mario Bros.*



(b) The generated level from the **Snaking-Depth-Paths** generator most similar to level 1-1 from *Super Mario Bros.*

(c) Level 1-3 from *Super Mario Bros.*

(d) The generated level from the **Snaking-Depth-Paths** generator most similar to level 1-3 from *Super Mario Bros.*

Figure 8: Two levels generated that are most similar to two levels from the original *Super Mario Bros.*

ORE system of Mawhorter (*?*). Examples of generated levels can be seen in figure **??**. The levels shown were chosen because they were the closest to two original levels across a number of metrics. Each level has a number of different metrics computed for it. The metrics are then normalized such that they have a mean of 0 and variance of 1. The l^2 distance is calculated and the closest point is chosen. The metrics considered are:

- C - The percentage of the levels that are completable by the simulated agent
- e - The percentage of the level taken up by empty space
- n - The negative space of the level, i.e. the percentage of empty space that is actually reachable by the player
- d - The percentage of the level taken up by "interesting" tiles, i.e. tiles that are not simply solid or empty
- p - The percentage of the level taken up by the optimal path through the level
- l - The length of the level in number of tiles in width
- L - The leniency of the level, which is defined as the number of enemies plus the number of gaps minus the number of rewards

- R^2 - The linearity of the level, i.e. how close the level can be fit to a line
- j - The number of jumps in the level, i.e. the number of times the optimal path jumped
- j_i - The number of meaningful jumps in the level. A meaningful jump is a jump that was induced either via the presence of an enemy or the presence of a gap.

This method of selecting generated content works not just as a methodology for choosing how to showcase generated content (removing problems of cherry-picking, random selection, and showing that the most similar generated content is not merely regurgitated), but is also a way for a designer to exert control on the generated content. A designer does not have to encode the rules that they are looking for, they need merely find exemplars.

An extension of this work replaced the simulated A* agents with paths provided by actual players. These paths were then able to bias the generation even though the coarse level geometry remained the same across the players. Due to the incorporation of player paths in the generation process, the generated levels generated paths informed by the player's actions with corresponding impacts on the generated levels. E.g. a player that interacts with enemies more is more likely to have enemies along their generated path, which in turn means that enemies are more likely to be in the generated level. We found that the players on the extremes (e.g. those who interact with every interactable object or those who interact with none) had the most biased levels, tailored to those play styles, while "average" players in between those extremes had levels most similar to the original *Super Mario Bros.* levels.

Mappy

Mappy is a system that observes play-throughs of NES games and produces maps, designated as graphs of "rooms" (a room being a tile represented level/sub-level). *Mappy* is designed to work on games where an *avatar* moves around a large *world* broken up into smaller *rooms*. This covers significant aspects of a broad class of games including platformers, action-adventure games, and role-playing games. We based this view of the world on these games' usual composition of four *operational logics* (***?, ?***): *collision logics*, which describe spaces made up of distinct objects which can touch each other and possibly block each other's movement; *linking logics*, which define larger conceptual spaces including connected rooms and the transit between them; *camera logics*, which account for the fact that the visible part of the world is a window onto a larger contiguous world; and *control logics*, which map e.g. button inputs to in-game actions.

Operational logics combine abstract processes (collision detection and restitution, the movement of the player between discrete spaces, the selective drawing of a sub-region of the whole level, or conditional control of the player character) with strategies for communicating these processes to players (tiles and sprites, scrolling or screen-fading to change rooms, continuous smooth scrolling, and ignoring input during cutscene-like segments such as switching rooms). We find that operational logics provide useful inspiration for knowledge representation and inductive bias; they help structure intuitive observations about how games function in a way

that is amenable to automation. The following sections expand on the leverage we get from operational logics as a knowledge representation.

In its current form, *Mappy* takes as input a playthrough of a game and the game program, then runs an NES emulator on that program and observes the system's state over time. *Mappy* watches a portion of the screen for changes; this screen rectangle is currently given in advance, but it could be determined automatically in the future. At each timestep, *Mappy* determines what tiles are visible on the screen, whether the screen is scrolling and if so by how much, and whether the player currently has control over the game (through speculative execution of inputs). *Mappy* accumulates a map of the current room as the game is played: when *Mappy* sees a new part of a room, it adds those tiles to that room's map. If a tile in a room changes, *Mappy* also notes that the tile has changed, storing a history of each coordinate's contents over time. This is important for capturing e.g. breakable blocks in *Super Mario Bros.* or collapsing bridges. *Mappy* also watches for cases when the player might be moving between rooms and starts on a new map when the move is complete. Finally, *Mappy* analyzes the rooms it has seen and suggests cases where two witnessed rooms might actually be the same room so that a human may choose whether to merge them together.

NES Pragmatics

Mappy works on NES games because that platform's hardware explicitly defines and supports the rendering of grid-aligned tiled maps (drawn at an offset by hardware scrolling features) and pixel-positioned sprites. The NES implements this with a separate graphics processor (the *Picture Processing Unit* or PPU) that has its own dedicated memory defining tilemaps, sprite positions (and other data), color palettes, and the 8×8 patterns which are eventually rasterized on-screen. During emulation, *Mappy* can directly read the PPU memory to access all these different types of data; we briefly describe the technical details below (referring the interested reader to (***?***)).

Although the PPU has the memory space to track 64 hardware sprites at once, there are two important limitations that games had to contend with: first, each sprite is 8×8 pixels whereas game objects are often larger; and second, the PPU cannot draw more than eight hardware sprites on the same *scanline* (screen Y position). This means that sprites are generally used only on objects that *must* be positioned at arbitrary locations on the screen.

Static geometry, including background and foreground tiles, are not built of sprites but are instead defined in the *nametables*, four rectangular 32×30 grids of tile indices; these four nametables are themselves conceptually laid out in a square. Since the PPU only has enough RAM for two nametables, individual games define ways to mirror the two real nametables onto the four virtual nametables (some even provide extra RAM to populate all four nametables with distinct tiles). On each frame, one nametable is selected as a reference point; when a tile to be drawn is outside of this nametable (due to scrolling) the addressing wraps around to the appropriate adjacent nametable. Note that many game levels are much wider than 64 tiles—the game map as a whole never exists in its player-visible form in memory, but is decompressed

on the fly and loaded in slices into the off-screen parts of the nametables as the player moves around the stage.

Mappy remembers all the tiles that are drawn on the visible part of the screen, filling out a larger map with the observed tiles and updating that map as the tiles change. A *Mappy* map at this stage is a dictionary whose keys are a tuple of spatial coordinates (with the origin initially placed at the top-left of the first screen of the level) and the time points at which those coordinates were observed, and whose values are *tile keys*. A tile key combines the internal index used by the game to reference the tile with the specific palette and other data necessary to render it properly (from the attribute table and other regions of NES memory). After *Mappy* has determined that the player has left the room (see Sec.), the map is offset so that the top-left corner of its bounding rectangle is the origin and all coordinates within the map are positive; this is rasterized and output as an image. We thereby construct the level as it is seen from the perspective of (tile-based) collision logics: the (mostly) static geometry and its (semantically significant) visual appearance over time.

We learn the full history of every tile, rather than committing to its initial or final appearance, for four main reasons. First, during scrolling, stale tiles are regularly replaced with fresh ones, and in some games this can even happen at the edges of the screen causing visible glitching. Second, we often fade into or out of rooms (or perform some other animation), and just taking the first- or last- seen tile could lead to unusable maps. Third, many tiles animate during play (for example, ocean background tiles or glittering treasures). Finally, the player can interact with many tiles: switches can be flipped, blocks can be broken, walls can be bombed, and so on. So we must store all the versions of a tile to admit applications like learning tile animations or interactions. For rasterization and visualization, we generally pick the tile's appearance 25% of the way into its observed lifespan, but this is an arbitrary choice and the generated maps are mainly for human viewing. A more principled choice might be to take the most common form the tile took during its lifespan.

While in general the nametables are used for terrain and the hardware sprites are used for game characters, there are some exceptions. Large enemies that do not animate much are often built from background tiles (as in some *Mega Man* bosses and *Dragon Quest* enemies); moving platforms act as terrain but generally must be implemented as sprites. Objects like movable blocks in *Zelda* or breakable bricks in *Super Mario Bros.* are tiles most of the time, but temporarily turn into sprites when interacted with so that they can animate smoothly off of the tile grid. *Mappy* does not account for these special cases yet.

Because some important level objects are sprites and not tiles, we also hope to learn the initial placements of dynamic game objects in the larger map. *Mappy* identifies abstract game objects by observing hardware sprites over time using the sprite tracker described in (***?***). This system uses information-theoretic measures to merge adjacent hardware sprites into larger game objects and maintains object identity across time using maximum-weight matchings of bipartite graphs (object identity and positioning in 2D space are natural conclusions to draw from collision logics). For *Mappy*, we take the first-witnessed position of each object, register those coordinates relative to level scrolling (explained in the next section), and render its constituent

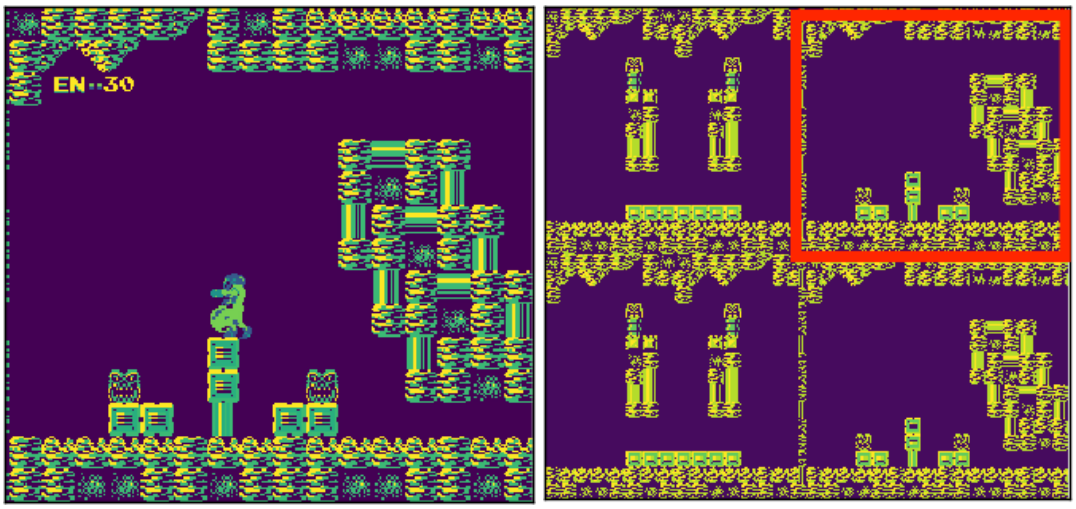


Figure 9: Visible screen registered with PPU nametables. Note vertical mirroring and horizontal wrapping.

sprites into our level maps to capture, for example, that a mushroom pops out of a particular question-mark block.

Scrolling

Although the PPU features hardware scrolling, and (some) of this information persists in the PPU's hardware registers, capturing screen scrolling information is surprisingly subtle. Games can alter the hardware scrolling registers essentially at any time during rendering, to achieve for example split screens or static menus over scrolling levels (the NES does not support layered rendering, unlike the Super NES). *Super Mario Bros.* and its sequels draw the top part of the screen containing status and score information without scrolling, and then turn scrolling on for subsequent scanlines. *Super Mario Bros. 3* puts status information on the *bottom* of the screen as well, so only a small window of the larger screen scrolls. These are concrete examples of camera logics, where a portion of the screen is dedicated to a viewport backed by the illusion of a moving camera. As mentioned above, we register the visible part of the level in a larger tilemap, under the assumption that a rectangular viewport will view a rectangular region of a potentially larger space.

We could have obtained pixel-precise scrolling information by instrumenting the emulator to trace when hardware scrolling state changes, but we wanted to see how far we could get without such interventions to remain as general as possible. We deploy two techniques, each with their own strengths and weaknesses: a perceptual algorithm based on registering each frame's visual output with the previous frame's and a hybrid approach which registers only the current frame's visual output (converted to grayscale) with the PPU's four nametables to determine which rectangular sub-region of the larger tilemap is being shown (see Fig. 9). The

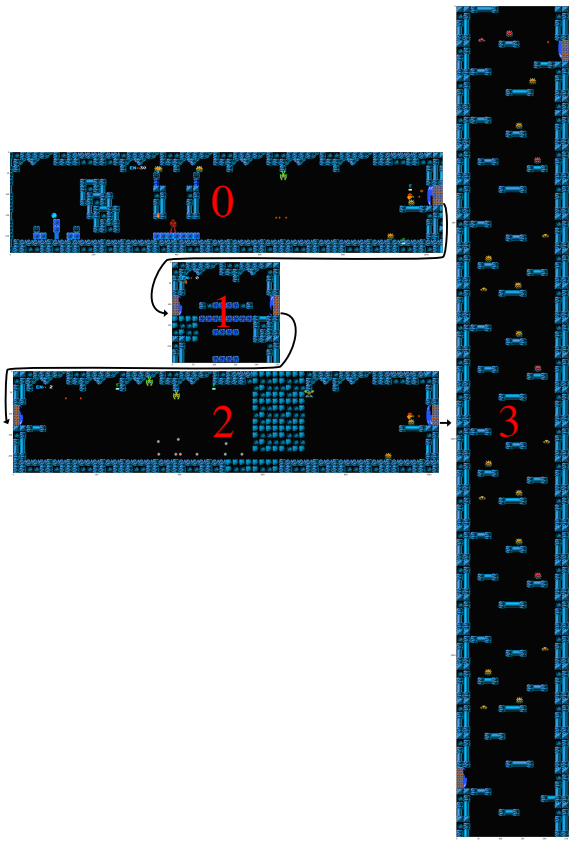


Figure 10: The first four rooms from *Metroid*. Note that we only observed a small portion of room 1, which is actually another tall vertical corridor.

former technique can break down with animated backgrounds (for example, waterfalls), while the latter will fail if the perceived scrolling is done mainly by sprites rather than background tiles, as in certain boss fights in *Mega Man 2*—this would also be an issue if we tracked hardware scrolling with the instrumentation described above. In either case, once *Mappy* has precise scrolling information it can convert coordinate spaces from the subset of tiles drawn on the screen into the frame of reference of the larger map it is assembling.

Linked Rooms

In this work we want to learn not only one large tilemap, but the graph structure by which smaller rooms are linked together (game worlds are not in general planar or even Euclidean). To do this, we need to determine when the player leaves one contiguous space for another. We consider two main ways in which linking logics communicate room changes to players:

- Smoothly scrolling between connected rooms

- Teleporting between rooms

The first type of transition is the most common type in *The Legend of Zelda* and *Metroid* (see Fig. 10). In these games, when traversing between most rooms the player loses control for a period of time while the screen rapidly but smoothly scrolls completely into the new room. After the player regains control, they are in a new room. To test for this type of transition we must know for each frame whether the screen is scrolling and whether the player has control; we already know about scrolling, so we use the savestate features of the emulator to determine whether the player has control.

The central question of player control is: "Would the world have been different if the player had done something else?" Because we know the full input sequence we can look a few moves ahead to see how the world will evolve according to the playthrough; we automatically take a screenshot of that state for reference. Next, we simulate seven possible futures (one for each button besides "start") three frames ahead and compare a screenshot taken in each of those eventualities aGhirahimgainst the reference state. If these actions produce different outcomes than the reference, then the player must have control at the initial frame.

In many games, some animations enacted by the player implicitly remove player control for some period of time (e.g. the fixed length jumps in *Castlevania*), so we have a configurable parameter for how long control must be taken away before counting as a complete loss of control. Since most room transitions take at least one or two seconds, and most in-game animations remove control for less time than that, this allows for a clean separation of the two causes for losing control. Of course, it is conceivable that the player does not have control but is not entering a new room, so we stipulate that the screen must also be scrolling while control is lost (and, indeed, that it must have scrolled by at least half the scroll window width/height). This accounts for freeze frame animations such as when Mario acquires a mushroom and grows or the fanfare that plays when Samus acquires a new item, which show a loss of control but the screen stays stationary.

The second category of spatial transition above places the player in a new room that has little or no visual relation to the previous room, perhaps from descending a staircase or going down a pipe. We treat these by looking at the overall appearance of the game screen, and if it changes too drastically within a short timeframe we assume that the player has probably teleported to a new room. This is complicated by game levels that incorporate drastic sudden changes to the visible portion of the tilemap (such as the "dark storm" level in *Ninja Gaiden* or Bright Man's stage in *MegaMan 4*), which yield false positives where *Mappy* thinks that it has gone to a different room. Given the optional room merging discussed below (and the possibility of stronger heuristics which we leave for future work), we do not believe that this is a fatal flaw.

Learning Mechanical Property Groupings

Learning mechanical properties of entities, such as "solidity" or "A harms B" is an important first step towards learning full rulesets and behaviors. These properties are borne out via their

mechanical interactions. I.e. an object might show the property of solidity due to the fact that other objects are unable to inter-penetrate it, or an object might be seen as harmful if it causes the deletion of the player or decreases a resource that the player sees as beneficial (e.g. the player's health or score). Some of the above approaches (Guzdial and Riedl) operate strictly on a visual level and only learn to place images with no concept of the mechanical properties, but I believe that the mechanical properties are key for generating sound, sensible levels. Some approaches, like those of Snodgrass and Ontañón and Dahlskog et al. require a human to semantically compress the visual representations down to mechanically important categories.

In earlier work, I used different machine learning approaches to learn the mechanical properties based on the effects that object interactions cause in the game using gameplay trace data.

The mechanical groupings that we learned (e.g. 'Solid', 'Enemy', etc.) rely on the cause and effect relationships that the objects in the games have, i.e. their causal affordances, that the different objects share. To learn the groupings, we must first find the causal affordances of objects and then cluster on those affordances. To learn these causal properties we must first specify what effects are, which effects we care about, and also the causes that we consider. The Operational Logics (OL) framework of Mateas and Wardrip-Fruin (***?***) presents an attractive method for determining what types of interactions we should care about. OLs are "the fundamental abstract operations – with effective interpretations available to both authors and players – that determine the state evolution and underwrite the gameplay" (*?*). For instance, graphical logics are those that deal with the simulation of movement and collision detection, and for the purposes of learning causal relationships in platformers are the core unit of analysis that we care bout when determining effects. Though a wide range of graphical logic effects can occur, we consider the following subset:

- $\triangle$ **velocity** - A sudden change in an object's velocity
- **Change of Visual Representation** - A visual representation of an object changing to a different visual representation
- **Addition of an Object** - An object not previously present is added to the screen
- **Deletion of an Object** - An object previously present is removed from the screen

While most are self-explanatory, the change of visual representation is the most open ended. An example of this type of effect is: "Mario changes into Fire Mario upon collecting (colliding with) a Fire Flower." The semantics of Mario changing from Learning mechanical properties of entities, such as "solidity" or "A harms B" is an important first step towards learning full rulesets and behaviors. These properties are borne out via their mechanical interactions. I.e. an object might show the property of solidity due to the fact that other objects are unable to inter-penetrate it, or an object might be seen as harmful if it causes the deletion of the player or decreases a resource that the player sees as beneficial (e.g. the player's health or score). Some of the above approaches (Guzdial and Riedl) operate strictly on a visual level and only learn to

place images with no concept of the mechanical properties, but I believe that the mechanical properties are key for generating sound, sensible levels. Some approaches, like those of Snodgrass and Ontañón and Dahlskog et al. require a human to semantically compress the visual representations down to mechanically important categories.

In earlier work, I used different machine learning approaches to learn the mechanical properties based on the effects that object interactions cause in the game using gameplay trace data.

The mechanical groupings that we learned (e.g. 'Solid', 'Enemy', etc.) rely on the cause and effect relationships that the objects in the games have, i.e. their causal affordances, that the different objects share. To learn the groupings, we must first find the causal affordances of objects and then cluster on those affordances. To learn these causal properties we must first specify what effects are, which effects we care about, and also the causes that we consider. The Operational Logics (OL) framework of Mateas and Wardrip-Fruin (*?*) presents an attractive method for determining what types of interactions we should care about. OLs are "the fundamental abstract operations – with effective interpretations available to both authors and players – that determine the state evolution and underwrite the gameplay" (*?*). For instance, graphical logics are those that deal with the simulation of movement and collision detection, and for the purposes of learning causal relationships in platformers are the core unit of analysis that we care bout when determining effects. Though a wide range of graphical logic effects can occur, we consider the following subset:

- $\triangle$ **velocity** - A sudden change in an object's velocity
- **Change of Visual Representation** - A visual representation of an object changing to a different visual representation
- **Addition of an Object** - An object not previously present is added to the screen
- **Deletion of an Object** - An object previously present is removed from the screen

While most are self-explanatory, the change of visual representation is the most open ended. An example of this type of effect is: "Mario changes into Fire Mario upon collecting (colliding with) a Fire Flower." The semantics of Mario changing from a small red/brown object to a larger white/orange object upon collision with the Fire Flower object are only readily apparent via play, but even without these semantics it is easy to understand that some change in the underlying simulation has occurred.

We used four different approaches:

- Pointwise Mutual Information
- Bayes Nets
- RESCAL
- The Infinite Relational Model

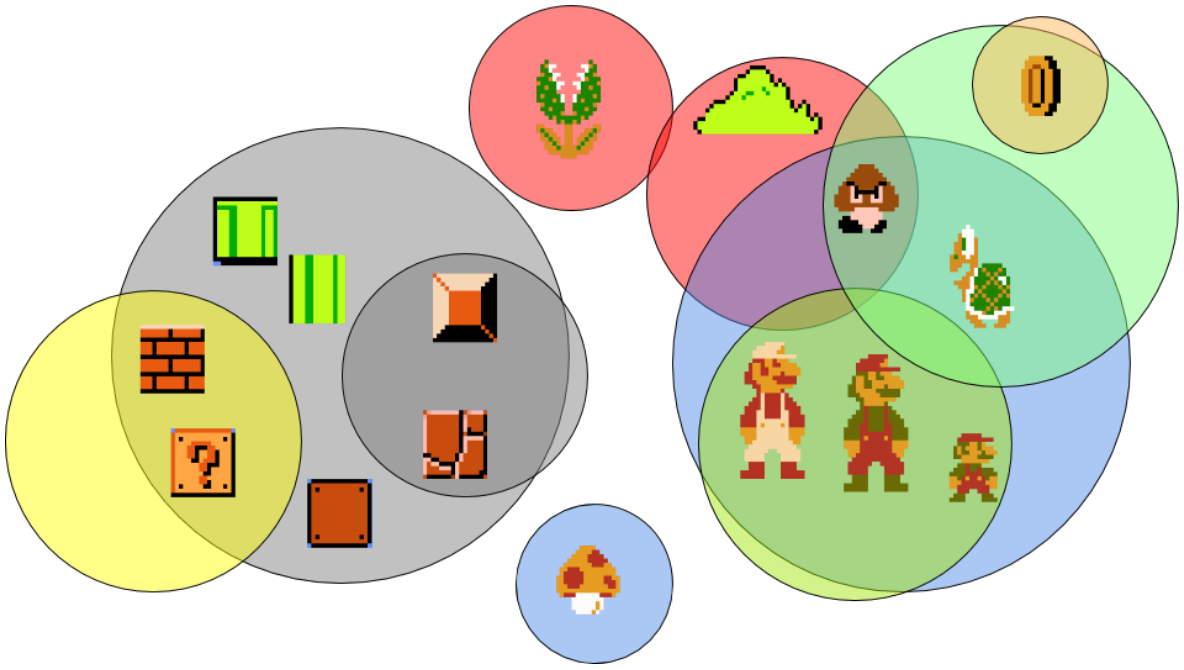


Figure 11: Clusters found by **PMI - Generalized**. *Grey* is solid, *Yellow* produces coins, *Blue* moves, *Green* can be deleted from the top, *Orange* can be deleted from the sides and bottom, *Yellow-Green* can hit ?-Blocks, *Red* harms the character

to find important causal relationships, and cluster them based on their shared properties. We found that a hybrid approach utilizing Pointwise Mutual Information to first cluster together the most highly causal interactions and then using the Infinite Relational Model to fill in gaps in the coverage had the best overall results with 15 discovered clusters, full coverage of all in-game entities, and an average cluster entropy of 0.07. The clusters found by Pointwise Mutual Information can be seen in figure **??**. In general the most important mechanical properties have been discovered, with important distinctions being found, e.g. all of the solid tiles (the gray cluster) are found with the fully inert tiles (the smaller gray cluster) being found different than those that have other properties (the breakable block and question mark blocks being able to be changed by collision with Mario) or some oddities (pirannha plants passing through only the pipes, or the solid brown block being the end-state of breakable blocks and question mark blocks).a small red/brown object to a larger white/orange object upon collision with the Fire Flower object are only readily apparent via play, but even without these semantics it is easy to understand that some change in the underlying simulation has occurred.

Remaining Work

There are two remaining pieces of work that I intend to look at for level generation:

- Integration of *Mappy* with *Kamek* and *Agahnim*
- Refinements on *Kamek*
- Producing the next iteration of *Agahnim*

Mappy Integration

The integration of *Mappy* will require more than just converting a *Mappy* output map into a form understandable by *Kamek* or *Agahnim*. Currently, *Mappy* does not learn any mechanical properties, and instead just keeps track of all seen visual representations of tiles. I believe that while the visual signifiers are important, that the mechanical properties of the game entities/background tiles is of higher importance. *Mappy* will first need to be integrated with my earlier work on learning mechanical property clusters.

Kamek Refinements

Beyond integration of *Mappy*, I would like to encode the levels in a lower dimensional space. Recently, there has been a lot of focus on Generative Adversarial Networks (GANs) (***?***), Variational Autoencoders (***?***), and sequence-to-sequence models (***?***) (which I used for *Mystical Tutor*) which all operate with a fixed dimensional latent "encoding" of the content they generate. Some, such as the GAN, attempt to learn this by producing output in an attempt to confuse a critic network as to what is generated and what is an actual representative. Others, like the VAE and sequence-to-sequence models learn an encoded latent representation that is then decoded to reproduce the input. The advantage to approaches such as these, is that they tend to learn densely informative latent representations, that can be utilized for meaningful modifications. For instance, some work has shown that head angle, lighting angle, and facial gesture (***?***) can be learned as dimensions in the latent space, allowing users to seamlessly change images by modifying these latent parameters. Similarly, interpolating in the lower dimensional space is capable of providing conceptually blended artifacts in a way not possible by averaging pixel values between images (***?***). Toward this end, I would like to learn a level representation that is capable of being utilized in these manners, with the goals being the discovery of important latent features (e.g., difficulty, etc.) that a user can modify and a meaningful generative space to be sampled from. This meaningful space is an important component for future work on *Agahnim*, which I will now discuss.

Agahnim Refinements

The earlier iteration of *Agahnim* relied heavily on human knowledge (e.g., annotation of room types, knowledge of the players' paths through the dungeons, viability constraints, etc.). It also used a machine learning formulation that was ill-suited for generalization (e.g., it relied on

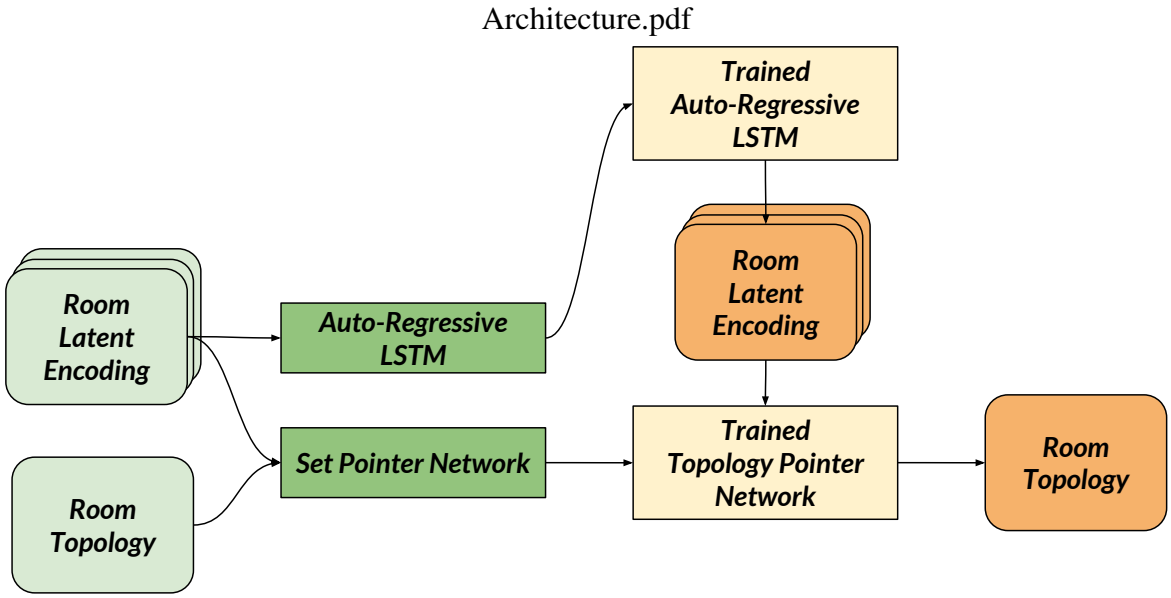


Figure 12: The proposed *Agahnim* architecture. *Agahnim* will be composed of two parts, an auto-regressive LSTM that will learn from and generate sets of rooms and a pointer network that will learn from and generate graph topologies from sets of rooms.

sparsely filled tables, so there were many situations that were outside of the training contexts). Toward this end, I propose a new architecture for *Agahnim*, illustrated in figure 12.

Agahnim will be composed of two components, an auto-regressive LSTM (the same network topoology found in *Kamek*) and a pointer network (***?***). The input to *Agahnim* will be a graph of latent room encodings (as learned and supplied by *Kamek*). The LSTM portion will train on sets of latent room encodings, so as to be able to generate a set of latent room encodings (which can then be decoded by a coupled *Kamek* generator). The pointer network will take in a set of room encodings and produce a set of edges between the input rooms, so as to turn the set of room encoding nodes into a graph.

Evaluation

The goal of these systems are to generate levels with similar properties as the training set, so I propose two measures for evaluating the generators:

1. Comparing the expressive volume of the generator to that of the exemplar set
2. Computing the e-distance of the generator's generative space to that of the exemplar set

The expressive volume is a concept that borrows from that of the *expressive range* of a generator. The *expressive range* (***?***) of a generator has typically been thought of as a visualization

of the metric space covered by the generated content. Most commonly, this has been visualized as a heatmap in 2 dimensions (linearity and leniency, classically (***?***, *?*)) although some have done up to 8 dimensions (2 at a time) (***?***). However, in the literature it is common to refer to the "width" of the expressive range, but this has yet to be done in anything beyond a qualitative visual assessment (***?***).

Volume is the measure we care about, as width is problematic as it is a linear dimension. For instance, a generator that always produced perfectly linear levels that had a very wide range in leniency would still be unlikely to be thought of as very expressive, given that it is completely lacking in 1 dimension. To this end, we will consider the $n-$dimensional volume (e.g., area in 2D, standard volume in 3D, etc.) of the generated metric space to be the *size* of the expressive range.

To calculate this volume, we use Kernel Density Estimation (KDE) as calculated by the "ks" R package (***?***). KDE determines a non-parametric function of the density of a sampled space, similar to the binning process of a histogram, but typically smooth given the use of a Gaussian kernel. Figure **??** shows the calculated Linearity and Leniency for the 1000 levels generated by MdMC Most-to-least 29 Level Generator as grey circles. The density estimate is visualized by black contour lines. We then threshold this density estimate for points greater than 0, which we take to be the boundaries of the expressive range. We then form an $n-$dimensional grid, and count the number of bins that lie within the expressive $n-$volume, multiplying the count with the volume of a single bin.

By comparing the volume of generators to that of the exemplar set, we can see how closely the generators are matching the target distributions.

Furthermore, we can directly compare the expressive space of two generators via a statistical measure. Most statistical tests are simply univariatek tests and we wanted to test how close the multi-dimensional joint probability distributions are for the metric space of two generators; however, the e-distance proposed by Szekely and Rizzo (***?***) compares samples as defined by

$e(C_i, C_j) = \frac{n_i n_j}{n_i + n_j}[2M_{ij} - M_{ii} - M_{jj}]$

where

$M_{ij} = \frac{1}{n_i n_j} \sum_{p=1}^{n_i} \sum_{q=1}^{n_j} \|X_{ip} - X_{jq}|$

and n_i, n_j are the sizes of the two samples and X_{ip} is the p-th observation from the i-th sample. This statistic follows an F-distribution which can provide p-values. From this we can tell whether the levels from a generator are believed to have come from the same distribution as the original levels, or, in comparing multiple genators, to determine which generators' distributions are closest to the distribution of the original levels.

Procedural Generation of Behaviors

Procedural generation is not just limited to levels, but can also be used to generate entity behaviors. I intend to show that behaviors can be learned from observation, which can be used to inform automated generation of entity controllers and behaviors. My work on behavior generation addresses:

- **RQ1** – The use of *CHARDA* to identify behaviors directly from observation
- **RQ2** – Explorations in representation for *CALE* showing which representations can be learned and generated efficiently

Related Work

There has not been much work in procedurally generating behaviors, but Siu, *et al.* (***?***) turn to behavior generation via the lens of program synthesis. The goal of program synthesis is to produce a program that satisfies a user's intent, either via pairs of input and desired behavior (***?***) or realizing a complete specification (***?, ?, ?***) Their work removes requirements on the desired shape of the design, with the only desired result being a well-formed program that describes the behavior of a Mega Man style boss. They do this by using a generative grammar with type-aware expansion where a given expression of type τ is either:

1. A literal of type τ .
2. A function call (with zero or more sub-expressions for arguments) that returns τ .
3. A reference to an existing in-scope variable of type τ .

ANGELINA$_1$ created "arcade-style" games similar to the aforementioned GVG-AI approach. Also similarly, it took an EC approach to the generation of non-player entity behaviors (chosen from a set of 5 potential behaviors), entity interaction behaviors, termination rules. The main difference is that *ANGELINA*$_1$ co-evolved levels to work with the evolved rules.

ANGELINA$_2$ turned from "arcade-style" game generation to generation of "Metroidvania-style" games (i.e. platformers that have lock-and-key progressions and non-linear player paths). To generate Metroidvania-style games, powerups are generated that increase the abilities of

the player, namely by increasing their jump height, allowing them to pass through previously obstructions, or decreasing the effect of gravity. Similar to *ANGELINA*$_1$ levels were co-evolved with powerups, with overall fitness rewarding a set of powerups and level design that allowed for a player to slowly progress through the level, with each new powerup increasing the amount of space the player can traverse.

While there has been little work on behavior generation, there does exist work on learning automata. Depending on the desired properties of the representation and the observed properties of the system, different approaches are suitable. For deterministic discrete state-space system there exists a body of work to learn Finite State Machines from observation.

Learnlib (***?***) is a library that learns Mealy machines via active learning. A Mealy machine is a deterministic finite automata that differentiates input and output alphabets. The active learning process iterates between exploration and testing phases. In the exploration phase, a sequence of system inputs are entered and the states reached are observed. In the testing phase, the hypothesized system is compared to the actual system and divergence in behavior is tracked so as to provide an input sequence for the next exploration phase.

For systems that have a mixture of real valued behaviors (e.g. position changing in real time) that change depending on discrete modes (e.g. an aircraft maneuvers differently while turning than it does while traveling at a constant heading) Hybrid Automata (HA) represent a natural formalism. Given the desirable properties of HAs, and the ready availability of tools for dealing with them, many researchers have explored automatically recovering these high-level models from real-world system behaviors. HyBUTLA (***?***) aimed to learn a complete automaton from observational data. HyBUTLA seems able to learn only acyclic hybrid automata, since it works by constructing a prefix acceptor tree of the modes for each observation episode and then merges compatible modes from the bottom up. Moreover, HyBUTLA assumes that the segmentation is given in advance and that all transitions happen due to individual discrete events, presumably from a relatively small set.

Santana *et al.* (***?***) learned Probabilistic Hybrid Automata (PHA) from observation using Expectation-Maximization. At each stage of the EM algorithm a Support Vector Machine was trained to predict the probability of transitioning to a new mode. Their work requires a priori knowledge about the number of modes.

Ly and Lipson used Evolutionary Computation (EC) to perform symbolic regression (***?***) to find common behaviors across different modes, as well as to find guard transitions between modes. Adding more parameters in the symbolic regression step will always increase model accuracy, so they used the Akaike Information Criterion (AIC) to balance model accuracy with model complexity. This work makes minimal assumptions about the types of behaviors that can be found in a given mode, but does require a priori knowledge about the number of modes that will be found in the dataset. Moreover, since their work assigns individual datapoints, not intervals, to a mode, their approach can only model stationary processes.

Other work has sought to learn non-HA descriptions of dynamical systems' behavior. Several approaches have sought to learn models that describe dynamical systems' behavior. Hidden Markov Models (***?***) learn probabilistic state transitions between a hidden state and the observed

data. The Infinite HMM (*?*) extends this to an unbounded number of states which assumes a Chinese Restaurant Process governs the state space. These approaches do not characterize guard *conditions*, but instead learn the *probability* of taking state transitions at each instant. Some techniques have leveraged certain inductive biases from biological or other domains to obtain good results. Kukreja *et al.* (*?*) found models of switched mode systems given prior knowledge about the locations of the switchpoints, and Bridewell *et al.* (*?*) used exhaustive search over model structures to best explain observed data without assuming fixed switchpoints.

Prior Work

With the ultimate goal of entity behavior generation, I worked on a system, CHARDA, to learn Hybrid Automata (HA) representations of the game mechanics present for different game entities directly from play traces. CHARDA recovers the distinct dynamic modes of the HA, learns a model for each mode from a given set of templates, and postulates *causal* guard conditions which trigger transitions between modes. CHARDA first learns which mode is active at each point in the trace via a dynamic programming implementation. At each sub-interval, we consider all possible mode templates, systems of equations that could possibly explain the observed phenomena, and determine the best possible model for that interval for a given definition of "best". In this work, we considered two criteria, Bayesian Information Criterion and Minimum Description Length, which are similar in form and function, but have different theoretical underpinnings. Both reward model accuracy and penalize model complexity, with the goal being to find a simple, well founded model. With each sub-interval considered, we can then construct the optimal assignment over all sub-intervals with a dynamic programming scheme. From this assignment, we then merge sub-intervals according to the aforementioned criterion, i.e. if merging two intervals into one shared mode improves the balance of accuracy and complexity.

With a segmentation of the trace into these modes, we then learn guard transitions between modes, i.e. under what conditions do we transition from mode A to mode B. e.g. In *SMB* Mario transitions from being on the ground to jumping when the A button is pressed. We do this by again using PMI to find which explanatory variables are most likely causing these transitions (e.g. we might observe many different B button presses or collisions with background objects with no change in mode, but the A button press is always found). The HAs learned by CHARDA can be seen in Figure 14, in comparison to the true HAs in 13. The Mario trace used for this work was 3772 frames in length, 63 seconds. The learned HAs are over-approximations of the true HA. Whereas the true HA has 3 separate jump modes based on the state of $\dot{x}$ at the time of transition, the learned HAs have only one such jump whose parameters are averages of the parameters of the true modes. Following from learning just one jump, CHARDA learns only a single falling mode. MDL does learn that releasing the **A** button while ascending leads to a different set of dynamics, but it considers this a change in gravity as opposed to a reset in velocity.

On Ground $\dot{y} = 0$ — Caused by Mario colliding with something solid from above

Jump(1,2,3) Three jumps with parameters:

- $\dot{y} := 4, \ddot{y} = -\frac{1}{8}$
- $\dot{y} := 4, \ddot{y} = -\frac{31}{256}$
- $\dot{y} := 5, \ddot{y} = -\frac{5}{32}$

Entered from **On Ground** when the **A button** is pressed and $|\dot{x}| < 1$, $1 \leq |\dot{x}| < 2.5$, or $2.5 < |\dot{x}|$, respectively

Release(1,2,3) $\dot{y} := min(\dot{y}, 3)$ — Entered from the respective **Jump** when the **A** button is released; $\ddot{y}$ same as respective **Jump**.

Fall(1,2,3) Falling at one of three rates: $\ddot{y} = -\frac{7}{16}$, $-\frac{3}{8}$, or $-\frac{9}{16}$; entered from the respective **Jump** or **Release** mode when the apex is reached ($\dot{y} \leq 0$)

Terminal Velocity(1,2,3) $\dot{y} = -4$ - Entered from **Fall** when $\dot{y} \leq -4$. The initial timestep in the **Terminal Velocity** state is actually $\dot{y} = -4 + \dot{y} - \lfloor \dot{y} \rfloor$ before being set to -4.

Bump(1,2,3) $\dot{y} := 0$ — Entered from a **Jump** or **Release** when Mario collides with something hard and solid from below; $\ddot{y}$ same as respective **Jump** or **Release**

SoftBump(1,2,3) $\dot{y} := 1 + \dot{y} - \lfloor \dot{y} \rfloor$ — Entered from a **Jump** or **Release** when Mario collides with something soft and solid from below; $\ddot{y}$ same as respective **Jump** or **Release**

Bounce(1,2,3) $\dot{y} := 4, \ddot{y} := a$ — Entered when Mario collides with an enemy from above; a is given by the respective **Jump**, **Release**, **Fall**, or **Terminal Velocity** state

Figure 13: The true HA for Mario's jump in *Super Mario Bros.* $:=$ represents the setting of a value on transition into the given mode, while $=$ represents a flow rate while within that mode.

MDL produces the more faithful model of the true behavior, but is overzealous in its merging of the distinct jump mode chains into a single jump mode chain. As such, it only recovers 7 of the 22 modes; however, abstracting away the differences between the jump chains it learns 7 of 8 modes, only missing the distinction between hard bump and soft bump. As an abstraction, this seems similar to what a human analyst might decide. Any implementation of this algorithm could certainly track which modes were merged and offer to un-merge them in case an analyst believes the elided distinctions are actually important.

On Ground $\dot{y} = 0$ — Caused by Mario colliding with something solid from above

Jump $\dot{y} := [3.97, 4.10], \ddot{y} = [-0.140, -0.131]$ — Entered from **On Ground** when the **A button** is pressed

Release $\dot{y} := [2.10, 2.54], \ddot{y} = [-0.430, -0.384]$ — Entered from **Jump** when the **A button** is released

Fall $\dot{y} := 0, \ddot{y} = [-0.373, -0.359]$ — Entered from **Jump** or **Release** when the apex is reached

Bump $\dot{y} := [-1.85, -1.27], \ddot{y} = [-0.324, -0.238]$ — Entered from **Jump** when something solid is collided with from below

Bounce $\dot{y} := [3.51, 3.82], \ddot{y} = [-0.410, -0.378]$ — Entered from **Jump** when an enemy is collided with from above

Terminal Velocity $\dot{y} = [-4.15, -4.06]$ — Entered from **Jump** or **Fall**

(a) HA with MDL as the penalty.

On Ground $\dot{y} = 0$ — Caused by Mario colliding with something solid from above

Jump $\dot{y} := [4.19, 4.42], \ddot{y} = [-0.195, -0.181]$ — Entered from **On Ground** when the **A button** is pressed

Fall $\dot{y} := 0, \ddot{y} = [-0.356, -0.338]$ — Entered from **Jump** when the apex is reached

Bump $\dot{y} := [-2.37, -1.67], \ddot{y} = [-0.289, -0.188]$ — Entered from **Jump** when something solid is collided with from below

Bounce $\dot{y} := [3.52, 3.88], \ddot{y} = [-0.424, -0.391]$ — Entered from **Jump** when an enemy is collided with from above

Terminal Velocity $\dot{y} = [-4.16, -4.05]$ — Entered from **Jump** when the threshold of -4 is reached.

(b) HA with BIC used as the penalty

Figure 14: Learned Mario HAs. Parameters as 95% confidence intervals.

Remaining Work

While CHARDA is able to learn automata from observation it is unable to generate. Toward this end, I propose CALE —Creating Automata from Learned Examples. The goal for CALE

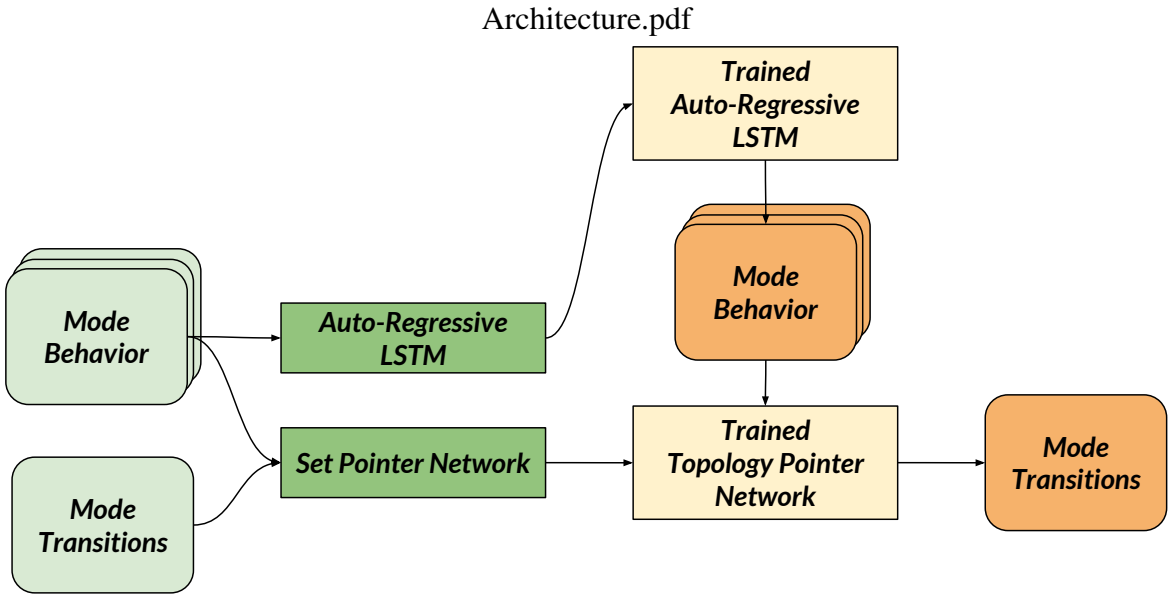


Figure 15: The proposed *CALE* architecture. *CALE* will be composed of two parts, an auto-regressive LSTM that will learn from and generate sets of mode behaviors and a pointer network that will learn from and generate transitions between modes.

is to produce automata after being trained on automata extracted by CHARDA. The proposed architecture of CALE (see figure 15) is very closely related to the architecture of *Agahnim*; however, instead of latent encodings, the modes are structures containing: $\dot{x}$, $\ddot{x}$, $\dot{y}$, and $\ddot{y}$ — the reset velocities and accelerations triggered by entering the state. Similarly, instead of just the presence of a link between two nodes (rooms for *Agahnim*, modes for CALE), CALE will need to generate the conditions under which a mode transition should occur. These conditions are:

- Collision with an entity (which is from a given mechanical grouping) from a certain direction
- $\dot{x}$ or $\dot{y}$ passes a threshold
- A given button is pressed, held, or released
- A within-mode timer has elapsed

Evaluation

It is very much an open question as to how to evaluate generated behaviors. Similar to generated levels, a notion of the expressive volume of the generative space of behaviors can perhaps be evaluated. Simple metrics such as number of modes, number of transitions, number of types

of transitions can be easily captured, but these do little to tell us what an automata is *about*. Perhaps the best metrics are found in situ, via testing similar to fuzzing. Where fuzzing is classically about testing software for improper behavior under random input data, here it would be about placing an automata in different situations with different input (which it may or may not respond to) and recording its positions. The original training automata will also be placed in the same situations, and their positions will also be recorded. These traces can then be compared to eachother, to see how similar generated behaviors are to the original behaviors.

Procedural Generation of Games

Finally, there is the goal of automatic game generation. Instead of being concerned with a single type of content found within a game, here the goal is to generate all aspects of a game, so that they act in concert with eachother. My work on game generation addresses:

- **RQ2** – The development of a new game specification language, *Cygnus*, that allows for games to be "read" procedurally
- **RQ3** – The bi-directional nature of *Gemini* allows it to verify that it has generated games that match its input specifications
- **RQ4** – The analysis path of *Gemini* shows a way for procedural readings of games to be performed by a machine, and *Leda* demonstrates a method to learn interpretative rules, widening the scope of analyses produced by *Gemini*

Related Work

Zook and Riedl (***?***) used Answer Set Programming to generate mechanics in the form of Planning Domain Description Language (PDDL) rules. PDDL rules take the form of a set of preconditions and postconditions. For instance, a spell that checks for the enemy being alive and reduces enemy health by 1 on the two next turns is:

$\langle DamageOverTime,$
$\{\langle Absolute, 0, GreaterThan(Health(Enemy), 0)\rangle\},$
$\{\langle Relative, 1, Update(Health(Enemy), -1)\rangle,$
$\langle Relative, 2, Update(Health(Enemy), -1)\rangle\}\rangle$

Answer Set Programming found mechanics that satisfied a number of constraints for playability (the player must be able to reach a goal location given some number of timesteps) and design heuristics (the set of generated mechanics should be small, and a minimal number of entities should be referenced by a given mechanic).

Nelson and Mateas (***?***) used ConceptNet (***?***) and WordNet (***?***) to generate *WarioWare* style minigames. ConceptNet and WordNet are graph structured knowledge bases that represent the semantics and relationships between different words. For instance, ConceptNet has information of the form `Duck` ***CapableOf*** `Shot`. Nelson and Mateas had 5 game templates and given input

entities and relationships filled in the slots in the templates, e.g. an *Avoid* game found entities that fit the relationship "Avoider-Noun Avoids-Verb being Attack-Verbed by Attacker-Noun".

Game-O-Matic by Treanor *et al.* (***?***) similarly required a human user to input the entities and relationships between them, and generated a game that intended satisfy those relationships. Instead of 5 template games, there were a set of standard verbs (e.g. Attacks, Avoids, Eats, etc.) which could then be mapped to different mechanics (e.g. Attacks might result in an entity that shoots bullets at the attacked or that chases after the attacked and harms them on touch) via micro-rhetorics. After choosing a micro-rhetoric for each relationship, the generator then used preset game recipes (e.g. a Pacman-like or Space Invaders-like) to take the loose pool of mechanics and for a cohesive game out of them.

Nielsen *et al.* (***?***)Towards generating arcade game rules with VGDL) used the General Video Game Artificial Intelligence (GVG-AI) framework to generate "arcade-style" games, i.e. action games that take place on a single screen that emphasize achieving a high score. They generated new game via two methods, mutation of existing games and random generation of new games. GVG-AI games are defined by a list of entities ($SpriteSet$), the rules for interactions between the entities ($InteractionSet$), termination rules for the game ($TerminationSet$), and a level. For a mutated game, one of 13 human authored games was taken as the initial input and the $InteractionSet$ was mutated with a 25% chance of mutation per rule. The random games were generated by randomly creating between $3 - -8$ entities, $3 - -10$ interactions, and 2 terminations (a win and a lose), with minimal constraints on the types of sprites/rules allowed. Games were then rated via simulated play with 6 different agents, with fitness being determined by a weighted sum of the difference in score between an "intelligent" (e.g. an Monte Carlo Tree Search variant) controller and an "unintelligent" (e.g. a controller that does nothing), whether a game can be won quickly, and whether a game can be both won and lost. From the initial set of 45 mutated and 9 randomly generated games, an EC algorithm was employed to evolve new games over 10 generations maximizing their fitness. While a few interesting games were discovered, the authors were generally unhappy with the results stating "Overall, the VGDL game generation process was not able to create any game of reasonably high quality, especially in comparison to the human-designed arcade games."

The *ANGELINA*$_{1-5}$ series of automatic game generators by Cook *et al.* (***?***) took a variety of approaches to the generation of games. The aforementioned *ANGELINA*$_{1-2}$ focused on behavior generation within a fixed game rule setting.

ANGELINA$_3$ is a direct follow on to *ANGELINA*$_2$ that takes the same genre, Metroidvanias, but introduces facets that tie in to real world goings on. *ANGELINA*$_3$ themes a platformer using a news article as the basis and finds images, sounds, and music that try to convey the message of the given news article.

ANGELINA$_4$ is an evolutionary system that generates mechanics via code reflection and modification, in addition to the generation of playable levels, given the generated mechanics. *ANGELINA*$_4$ takes as input a code base that represents the base definitions of a game, i.e. all of the entities in the game and their respective fields. The generation process occurs in two phases, first an EC algorithm is run to generate mechanics, where fields are taken at random

and a type specific modifier is chosen at random (e.g. inverting a boolean field or modifying a numerical value). These mechanics are evaluated via simulation on a test level. Once the mechanics have been chosen, an EC algorithm is run to find levels that are playable given said mechanics and that utilize the generated mechanics (which exist alongside more standard platformer mechanics).

Finally, *ANGELINA*$_5$ moves away from 2D games and instead creates first-person 3D games, focused on movement through a space. Similar to *ANGELINA*$_3$, *ANGELINA*$_5$ takes in a them and sources sounds, music, 3D models, and textures to convey the desired theme.

Moving away from videogames, Cameron Browne's *Ludi* (***?***) generates 2 player board games. *Ludi* uses EC to generate games and then undergoes a multi-step evaluation. Similar to the work of Nielsen *et al.*, simulated players play the games and their playtraces are evaluated for different aesthetic criteria, such as whether the game is balanced (player order does not matter) or the game has drama (a player can come back from a bad position to win the game). While a wide variety of rulesets are possible with the *Ludi* game formalism, the EC tended to favor N-in-a-row goals more heavily than other types of rules, perhaps speaking to an issue in evaluation.

Prior Work

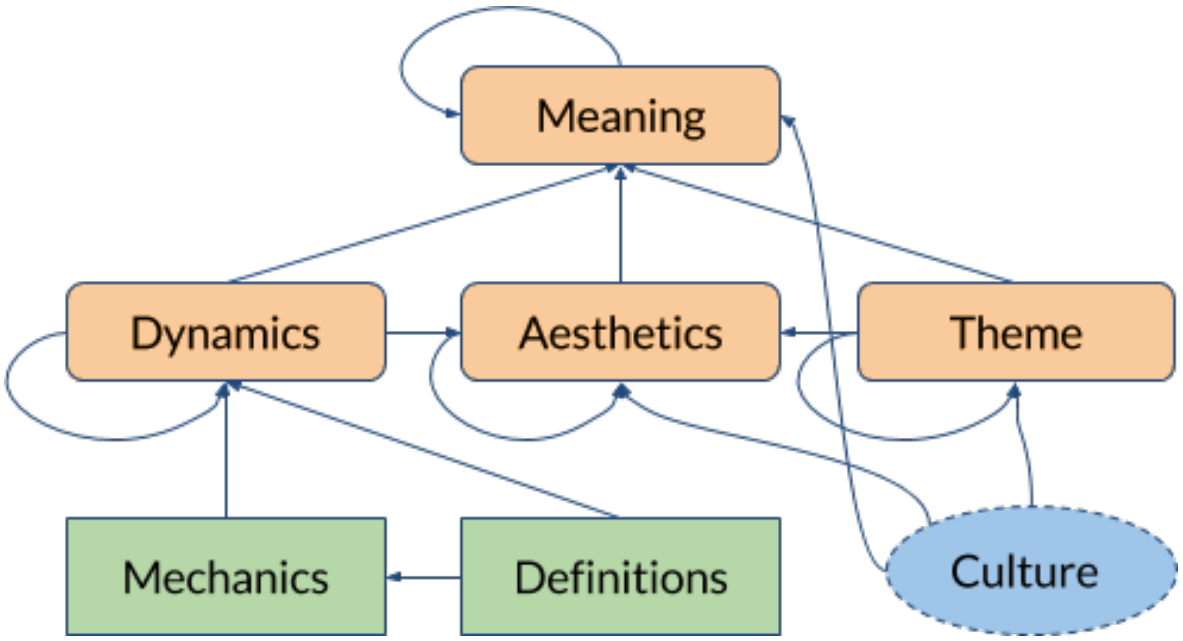


Figure 16: The structure of a proceduralist reading of a game. Ground facts of the game are green, interpretations are orange, and the cultural knowledge that informs the reading is blue.

As a part of a larger project designed to procedurally generate narratives paired with match-

ing thematically relevant minigames, I have produced a *Gemini*, a game generator focused on generating games that afford specific proceduralist readings, a semi-formal approach for interpreting the meaning of a game based on its underlying processes and interactions in conjunction with aesthetic and cultural cues, offer a novel, systematic approach to game understanding. In proceduralist readings, a game is defined as a set of definitions (entities, resources, etc.) and a set of mechanics. From these ground facts, the dynamics of play, i.e. what the player will actually do while playing (e.g. if the player has the ability to jump and there exist enemies, the player will jump to avoid them), as well as the aesthetics of play, i.e. how the game actually feels given the dynamics and theming. These interpretations can be built up to provide high-level readings of the game (e.g. the game of *Kaboom* feels hopeless, since difficulty is always increasing and there is no possibility of winning). This structure can be seen in figure 16.

In earlier work, the generator portion of the program was turned off, and instead was run only in the proceduralist reading direction, i.e. given a game definition it produces a set of readings for that game.

One example carefully studied by Treanor et al. is *The Free Culture Game*, in which "new ideas" are represented as floating particles that must be herded towards producers in the creative commons to keep them inspired (creating new ideas) and away from the *vectorialist*, who takes ideas out of the creative commons to commodify them for consumers. The player exerts an indirect force via the mouse cursor on new ideas. Several proceduralist readings are extrapolated from these mechanics together with the game's interpretive affordances, such as the colors selected for producers versus consumers (green versus grey) and the robotic and malicious audio-visual character of the vectorialist. An annotated screenshot can be seen in figure 17.

For example, they read the following meanings from the game:

1. The player must navigate the cursor between the vectorialist and new ideas to prevent commodification.

2. The vectorialist is an evil adversary who does not care about the happiness of people.

They derive these meanings from a number of implicit rules, which they call *dynamics*. We read these as equivalent to (***?***)'s *constitutive mechanics*, though we will use the term *dynamics* here to avoid confusion. For example, the first inference shown on the path to deriving the first reading is:

*Because producers need new ideas to collide with them in order not to turn into consumers, the player's goal is to maintain as many producers as possible, and the player can exert a force on the new ideas, the player will push new ideas towards producers. (**R**eading 1)*

This derivation can be made more explicit by identifying the base assumptions that can be directly observed about the game (e.g. **G**oals and **M**echanics), then building the argument in a tree structure:

Base assumptions:

(G) The goal is to maintain producers.

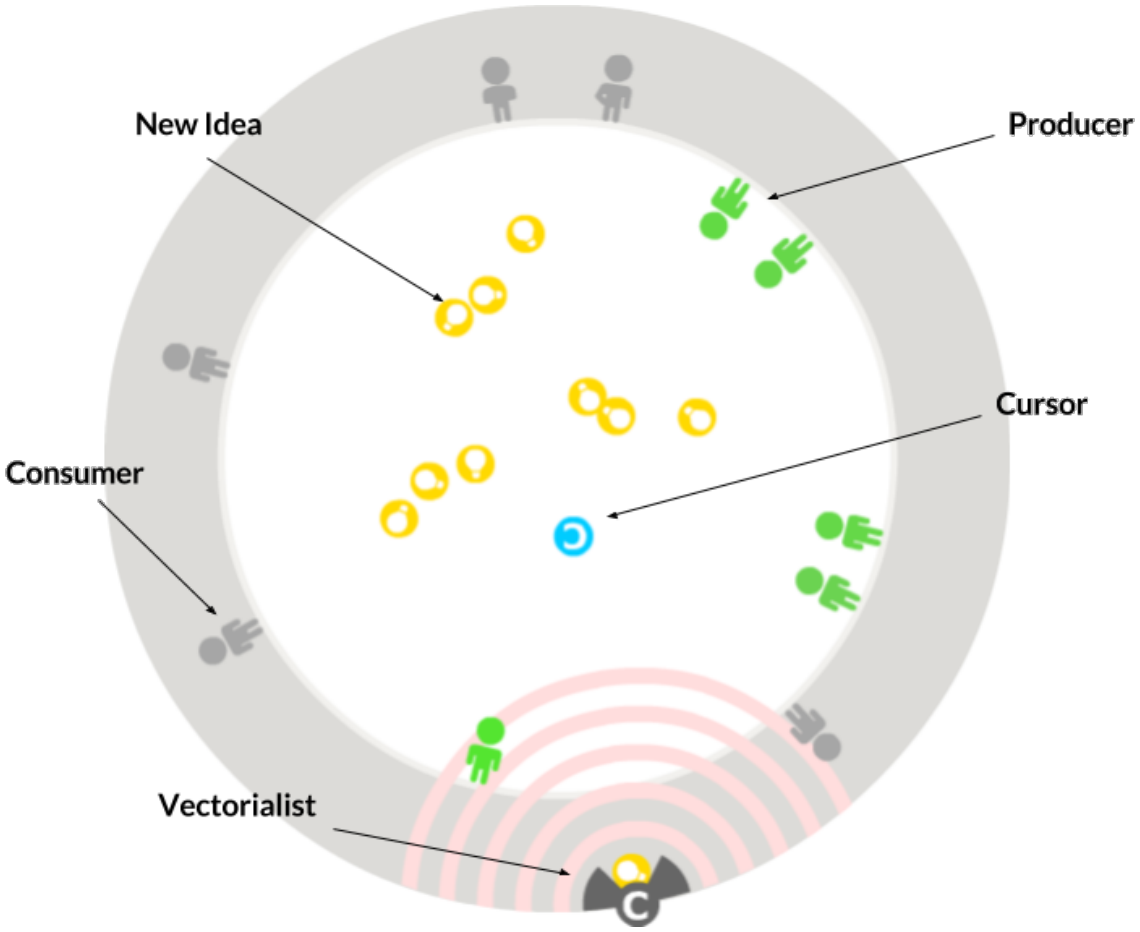


Figure 17: A screenshot of the *Free Culture Game*.

(M8) When the ideas absorbed by a producer go below a threshold, the producers convert into consumers.

(M6) When a producer collides with a new idea, the ideas absorbed increases.

(M2) The cursor pushes new ideas.

Meaning derivation :

By	
(G), **(M8)**	**(R1)**. The goal is to maintain ideas absorbed.
(M6), **(R1)**	**(R2)**. The player wants the producer and new ideas to collide.
(M2), **(R2)**	**(R3)**. The player will use the cursor to push new ideas toward the producers.

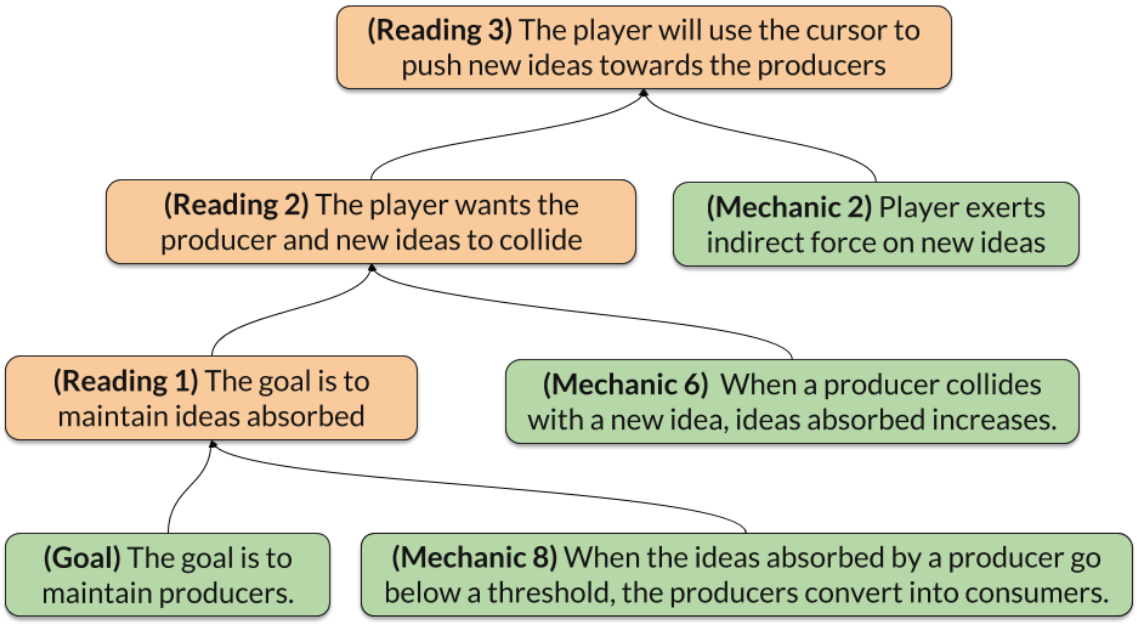


Figure 18: A color-coded example meaning derivation of an aspect of the aesthetics of the *Free Culture Game*. *Green* nodes represent concrete pieces of game **M**echanics and stated **G**oals that are used as the building blocks for inference chains. *Orange* nodes represent **R**eadings, i.e. dynamics such as **(R1)** are derived from these mechanics and in turn used to derive (also *Orange*) aesthetics such as **(R2)** and **(R3)**.

Characterizing game analysis as a process of constructing these reasoning structures constitutes an initial step towards a computable formalism. To operationalize this process, we need to understand the implicit inference rules that govern *why* each proof step is valid. For example, to state that **(R1)** follows from **(G)** and **(M8)** makes use of the reasoning that we do as humans about the relationship between goals and circumstances that defeat those goals.

Some other game reasoning principles that we must codify to fully formalize the *Free Culture Game* meaning derivation include:

- If the player's goal is G and action A accomplishes G, the player will do action A.
- Pushing an entity E with an entity P causes E to move away from P.
- An entity moving away from something might move toward another entity.
- When E moves toward X, E and X might collide.

This kind of reasoning is powerfully general in that we can expect to apply it to many games without having to encode game-specific interpretation knowledge. It includes knowledge about player goals and control, causal relationships between game events, and knowledge about

state change phenomena such as spatial and physical relationships, increase and decrease of resources, the passage of time, and win and lose conditions. In other words, this kind of *game literacy* must be made explicit in order to perform proceduralist readings.

We describe game mechanics as a collection of named *outcomes* that have preconditions and results, similar to linear logic-based game specification in Ceptre (***?***) or the sensors and actors of Kodu (***?***). To standardize our game descriptions in such a way as to describe a broad range of genres, but apply the same reasoning principles to all specifications, we developed the Cygnus game specification language, which includes the aforementioned outcomes as well as notions of entity, resource, timer, and player controls. Using this formalism, which we call Cygnus, we can describe the mechanics of *The Free Culture Game* as well as other classic arcade-style games.

A non-exhaustive list of possible preconditions includes:

- Comparisons of resources: e.g. $R_1 >= 0$
- Collision detection: $overlaps(E_1, E_2)$
- Geometric proximity: $near(E_1, E_2)$
- Timers elapsing: $timer\_elapsed(T_1)$
- Player input: $button\_press(mouse, held)$

A non-exhaustive list of possible results includes:

- Resource modification: $R_1 += 2$
- Entity movement: e.g. $move(E_1, north)$; $move\_toward(E_1, E_2)$
- Entity creation/deletion: e.g. $delete(E_1)$
- Game mode changes: $mode\_change(game\_loss)$

To carry out automated inquiry about a game, the author creates a specification of the game's mechanics in Cygnus, then runs the answer set solver on it in conjunction with the reasoning principles, resulting in a stable model (collection of logical facts) representing derived knowledge about the game. Here we give one example in depth to illustrate the approach, and we later summarize further efforts for breadth.[1]

[1]All code is available at: `https://github.com/LudoNarrative/ClimateChange/tree/master/GameGenerator/Justifications`

The Free Culture Game

First, we describe our encoding of the *Free Culture Game*'s mechanics in Cygnus (which we directly embed as ASP predicates). One such mechanic is *The vectorialist pulls in new ideas*, which we describe as an outcome with a single precondition, *the vectorialist is near a new idea*, and a single effect, *the new idea moves toward the vectorialist*. In ASP predicate notation, we assign this outcome the name `pull_idea` with the following syntax:

```
precondition(
  near(vectorialist, new_idea),
  pull_idea).
result(pull_idea,
  move_toward(new_idea, vectorialist)).
```

The `pull_idea` token is simply an identifier to connect the precondition to the result, while the `near(-)` and `move_toward(-,-)` predicates designate specific meanings as preconditions and results in the Cygnus language.

We encode the rest of the game in a similar fashion; we will describe them in-line with informal descriptions of preconditions and results. The interested reader may refer to Figure 19 for the encoding of these rules as ASP predicates. For example, the following three mechanics establish the relationships between producers, new ideas, and the player:

- (Mechanic 1): "Producers make new ideas." Precondition: a slow repeating timer goes off. Result: a new idea is added to the game in a random location.
- (Mechanic 2): "The cursor exerts force on ideas." Precondition: the cursor is near a new idea. Result: the new idea moves away from the cursor.
- (Mechanic 6): "Collision between a new idea and a producer increases ideas absorbed for that producer." Precondition: a new idea and a producer overlap. Result: the producer's ideas absorbed resource increases, which is reflected by the producer shifting in color from grey to green.

In addition to specifying mechanics, we may also supply facts about the initial state of the game and the game's goal, such as *The player's goal is to prevent producers from converting into consumers.* See Figure 20 for these auxilliary clauses.

We pass our reasoning principles and game specification to an answer set solver, specifically Clingo (***?***). The solver determines a set of facts (an answer set) that is consistent and complete with respect to all axioms and rules provided. This will include all facts derivable from the reasoning principles about the specified game mechanics.

The answer set generated when given the game specification for the Free Culture Game includes the following facts:

```
%% (Mechanic 1): Producers make new ideas
precondition(slow_timeout, gen_idea).
result(gen_idea, add(new_idea)).

%% (Mechanic 2): The cursor exerts force on ideas.
precondition(near(cursor, new_idea), push_idea(cursor)).
result(push_idea(Entity), move_away(new_idea, Entity))
  :- physicsLogic(Entity, pushing).

%% (Mechanic 3): The vectorialist moves toward groups of new ideas.
precondition(far(vectorialist, new_idea), scan).
result(scan, move_toward(vectorialist, new_idea)).

%% (Mechanic 4): The vectorialist pulls in new ideas.
precondition(near(vectorialist, new_idea), pull_idea(vectorialist)).
result(pull_idea(Entity), move_toward(new_idea, Entity))
  :- physicsLogic(Entity, pulling).

%% (Mechanic 5): Collision between new ideas and vectorialist
%%                kills new idea & increases old ideas.
precondition(collide(new_idea, vectorialist), commodify).
result(commodify, delete(new_idea)).
result(commodify, increase(old_ideas, med)).

%% (Mechanic 6): Collision between new idea & producer increases
%                ideasAbsorbed.
precondition(collide(new_idea, producer), learn).
result(learn, increase(ideasAbsorbed, mid)).
result(learn, set_color(producer,green)).

%% (Mechanic 7): ideasAbsorbed decreases with time.
precondition(tick, forget).
result(tick, decrease(ideasAbsorbed, low)).
result(tick, set_color(producer,gray)).

%% (Mechanic 8): If ideasAbsorbed goes to 0, producer turns into consumer.
precondition(le(ideasAbsorbed, 0), convert_producer).
result(convert_producer, delete(producer)).
result(convert_producer, add(consumer)).

%% (Mechanic 9) (only referred to later than 1st example):
%%  If ideasConsumed goes to 0, consumer turns into producer.
precondition(le(ideasConsumed, 0), convert_consumer).
result(convert_consumer, delete(consumer)).
result(convert_consumer, add(producer)).

%% (Mechanic 10) (only referred to later than 1st example):
%% Vectorialist provides consumers with old_ideas
precondition(gt(old_ideas, 0), feed_consumer).
precondition(near(vectorialist, consumer), feed_consumer).
result(feed_consumer, decrease(old_ideas, low)).
result(feed_consumer, increase(ideasConsumed, low)).
precondition(tick, forget_old).
result(forget_old, decrease(ideasConsumed, low)).
```


Figure 19: Formal specification of the mechanics of the *Free Culture Game* in Cygnus.

```
%% Initializations
initialize(set_to(ideasAbsorbed,10)).
initialize(set_sprite(producer, person)).
initialize(set_color(producer,green)).
initialize(set_sprite(consumer, person)).
initialize(set_color(consumer,gray)).
initialize(set_sprite(vectorialist,
  evil_robot)).
initialize(set_color(vectorialist,black)).

%% (Goal) The goal is to prevent producers
%% turning into consumers.
goal(prevent(convert_producer)).
```

Figure 20: Auxilliary clauses for the *Free Culture Game*.

The goal is to maintain `ideasAbsorbed`. The outcome `forget` affects `ideasAbsorbed` negatively, and the outcome `learn` affects it positively, meaning that `learn` is a favorable outcome. The player has agency over the cursor and uses the cursor to influence the outcome `push_idea(cursor)` which in turn enables the outcome `learn` by pushing ideas towards the producers. As such `learn` is a favorable outcome that the player has indirect control over via pushing ideas with the cursor, so the player will place the cursor near `new_ideas` to try to enact the `learn` outcome. A graphical representation of these inferred properties along with the rules that were required to derive these can be seen in figure 21.

However, this is for simply reading a game. To generate a game, we instead fix a reading, e.g. the game must represent sharing, and generate a game that affords that reading. The same rules that produce the above reasoning chains are used, but a host of additional rules are required for generation. The pipeline for game generation is:

1. Generate partial Cygnus game definitions using ASP
2. Determine scalar values found in rules (e.g. boolean comparisons or modification of other scalar values) using EC
3. Compile complete Cygnus game definitions to Phaser (a javascript game library)

Given the ability to analyze a set of game rules and mechanics to derive a reading, the declarative nature of ASP allows us to easily invert the process to take a fixed set of readings and generate a game that meets the desired reading. Choice rules are a construct in ASP that allow for the solver to non-deterministically choose facts such that they are consistent with all other facts and constraint rules. The simplest example in our code is:

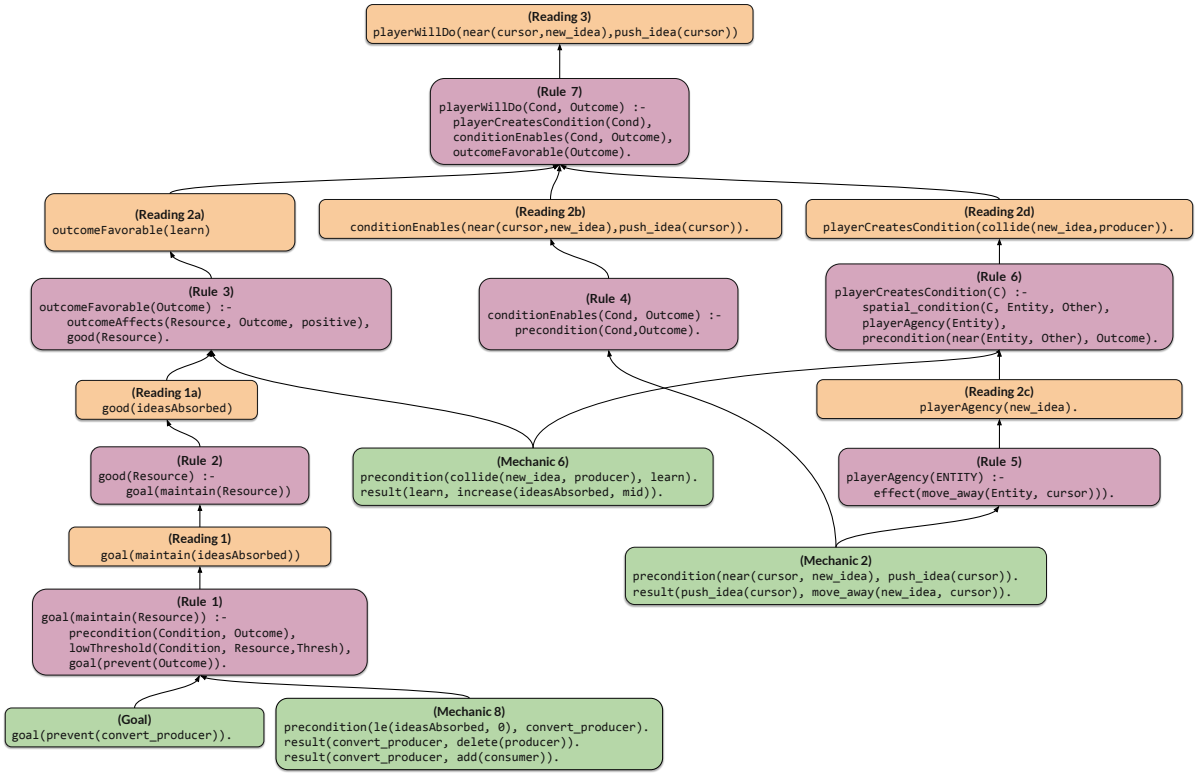


Figure 21: A full reasoning chain, analagous to the prose interpretation found in figure **??**. The rules invoked in the reasoning are chain are found in purple. As can be seen, numerous rules and sub-facts might need to be invoked to build up to larger pieces of knowledge, such as the chain building up to **Reading 2a** "The outcome labeled 'learn' is favorable" which builds off of 3 separate game definitions and invokes **Rules 1**, **2**, and **3**.

$\{$`max_entity(M)`$\}$ `:-`
`M = min_entities..max_entities.`
which chooses an integer M to be the term of the predicate `max_entity`, such that
`min_entities` $\leq M \leq$ `max_entities`

In general, the choice rules that make up the generation can be thought of under a few broad classes:

- **Game Definitions** — General facts about the game, such as the number of entity types, the number of resources, the visuals associated with each entity type, or the number of timers found in the game
- **Rule Definitions** — As previously mentioned, a rule is a set of preconditions and results. *Gemini* is able to arbitrarily choose any combination of conditions, such as player input, resource comparisons, entity-to-entity interaction, or a timer elapsing, and pair them with

any combination of actions, such as the creation or deletion of an entity, the modification of a resource, or the movement of an entity.

- **Mechanic Choices** — While *de novo* mechanic generation is a core aspect of *Gemini*, we have also hand encoded mechanics, such as an entity following the cursor, an entity chasing another entity, or an entity fleeing from another entity.

The inclusion of hard-coded mechanics could feel at odds with our goal of game generation, but we found from playtesting that many of the generated games contained control schemes that, while able to be played, induced too much confusion and placed too much of a cognitive load on players to be useful for a generated experience (e.g., a game generated where clicking on an entity causes it to move up, pressing the mouse button while not clicking on it causes it to move right, and pressing the up key causes it to rotate to the cursor is technically controllable but is unpleasant for a human to play). However, while the base definitions of the mechanics are hard-coded, *Gemini* is able to modify any rule with the addition of preconditions and results, which leads to the generation of novel mechanics that are not hard-coded. In essence, the hard-coded rules act as a biasing of the generator towards games that have interpretable movement models with little other effect on the generative space.

There are 163 game design common-sense constraints placed on the generated games to ensure readability and playability. By readability, we mean ensuring that they are designed for human players. For instance, games become very difficult to parse if there are a large number of preconditions for a given rule (e.g., the player can only move right when a red circle and blue square are touching, resource R is above 3, resource S is between 2 and 7, they are holding the mouse button, and pressing the right arrow key simultaneously), so we limit the number of simultaneous preconditions to 3. Others perform static analysis on the rules to ensure that all outcomes are achievable. For instance, if there is a rule predicated on a resource being greater than a threshold, there must also be a way for that resource to increase. Furthermore, to eliminate dead lock, the way to increase that resource must not be predicated on the aforementioned rule. While most of these rules are always hard constraints, there are 19 that are able to be ignored if the user specifies that they can be ignored. For instance, a general rule is that if a resource is initialized, it must be used in at least one precondition and it must be modified by at least one rule; However, points are a common occurrence in games and they often are modified by the game, but no rules are conditioned on them, so if the user wants a point system in their game they can disable the constraint.

In addition to the common-sense design rules, *Gemini* can also accept a design intent, the readings and specifications with which the user wants the generated game to conform. These can be as simple as specifying the number of different types of entities that can appear, or more complex specifications such as:

```
:- not reading(hand_eye_coordination).
:- not reading(maintain,resource(r(1))).
label(resource(r(1)),concentration) :-
  reading(good,resource(r(1))).
```

```
label(resource(r(1)),stress) :-
   reading(bad,resource(r(1))).
```
or more simply, "The game should be read as a hand eye coordination game that requires the player to maintain resource $r(1)$. If $r(1)$ is determined to be good, it should be labeled as concentration, and if $r(1)$ is determined to be bad, it should be labeled as stress." (We note that in AnsProlog headless rules are read as constraints that forbid the body from being true, hence the `not` in the first two rules).

After generating the rules and definitions of the game, it must be made playable. The second phase takes in the *Cygnus* game definition and compiles it to JavaScript for use in with the Phaser game engine. The compiler operates by converting the input into the Rensa (***?***) format, a Prolog-like relation specification. The compiler operates by constructing Rensa relations from *Cygnus* code, e.g. `resource(value1)` $\rightarrow$ $\langle$ `value1` $instance\_of$ `resource` $\rangle$. The Rensa relations are then converted to JavaScript code in a context sensitive manner, owing to the intricacies of the Phaser engine. For example, code resulting from clicking on an entity is handled in a callback function, as is code resulting from entity-to-entity collision, but when a result is predicated on both clicking and entity-to-entity collision the callbacks must be nested such that the collision checking code is called from within the click callback.

It is very important that *Gemini* make things playable, for a number of reasons. Perhaps the most important reason from a non-research perspective is that *Gemini* was created in service to a larger creative project so generating just rules or VGDL (to then be interpreted by a cumbersome framework) was not a possibility. However, the most important reason is that the debugging process for rule writing requires playability. A game might seem plausible when just reading the rules, but only when played is it shown to be untenable. E.g, a game where the player controls an entity chasing another a fleeing entity seems reasonable and challenging; however, in the absence of any other rules, this will inevitably lead to the chased entities getting stuck in the corners of the screen. Had we only read the rules, we would have missed this interaction rising from the dynamics.

Remaining Work

A key authorial challenge with *Gemini* is that of the construction of interpretative rules. Often, it can be much easier to identify that a specific type of interaction/behavior/interpretation is occurring than to specify *why* that is. Towards this end, I have begun work on *Leda*, an inductive logic programming system designed to learn these rules.

In creating *Leda*, we wanted an approach that satisfied:

1. Could learn arbitrary clauses (both functors and literals)
2. Required only the base background knowledge and positive examples
3. Could utilize language constructs from AnsProlog

Other approaches fail these requirements for a variety of reasons: FOIL is inappropriate due to (1), GOLEM is inappropriate due to (3), Metagol is inappropriate due to (2) and (3), etc. This led to the creation of *Leda*. We note that *Leda* is very much a work in progress, so while fully utilizing the language constructs specific to AnsProlog is a desired end goal, currently *Leda* only utilizes the constructs of AnsProlog shared with Prolog (e.g., aggregations such as `X {p(Z)} Y` remain unused).

Leda takes in a list of games G and a corresponding list of positive examples for those games E. Games take the form of Cygnus (***?***) programs. An example rule from a Cygnus representation of *Pong* is show below:

```
precondition(control_event(
   player_input(up_arrow,held)),move_up).
result(move_up,
   moves(paddle_player,north,low)).
```

This rule defines the set of preconditions under which it fires (e.g., the control event coming from the player input of the Up arrowing being held down) and the results (e.g., the entity labeled as the `paddle_player` moves north by a low amount). All of the definitions and rules describing a game are typically on the order of 40-50 lines of Cygnus code (e.g., *Pong* can be represented in 37 lines and *Kaboom!* in 40).

Leda operates by reading in each game and determining a graph structure for how the lines of code relate to each other. This is done by determining the terms found in each line and associating lines that have the same terms. For instance, in this example the first line has the terms:

{`up_arrow, held,move_up`}

and the second:

{`move_up,paddle_player,north,low`}

So the association between these lines is that both contain the term `paddle_player`. This is important, as it tells us how we can expect to use these lines in our rules. For instance, we know that a rule might take the form of:

```
Rule :- precondition(control_event(
                player_input(A,B)),C),
        result(C,moves(D,E,F)).
```

i.e. we use the only related term to determine the implicit type structure as opposed to explicitly defining the types of terms.

Given the graphical structure of how the clauses relate to each other, we know perform a breadth-first search over the rule space. The starting point of this search are the positive examples given. We will keep with the *Pong* motivating example and say that our positive example is `player_controls(paddle_player)`. Given this, we know that the neighbors of `player_controls(paddle_player)` should include the term `paddle_player`. In this case, the clauses with `paddle_player` are:

```
entity(paddle_player).
initialize(paddle_player,1,
```

```
   location(middle,left)).
initialize(set_sprite(paddle_player,
   rectangle)).
initialize(set_color(paddle_player,
   white)).
moves(paddle_player,north,low)).
moves(paddle_player,south,low)).
precondition(overlaps(ball,paddle_player),
   player_hit).
```

Given the bias that better rules are those that are more compact, we visit the neighbors in ascending order number of predicates. With a set of clauses to be used in the rule, we then enumerate all possible rules given those clauses. This combinatorial space is made of all combinations of:

- Concrete and variable terms admitted by the clauses
- Subsumation of functors into variables
- Safe negations

Again, given the two clauses:

```
precondition(control_event(
   player_input(up_arrow,held)),move_up).
result(move_up,
   moves(paddle_player,north,low)).
```

The abstractification process for the rule finds the set of all concrete terms found in the clauses (e.g., {`up_arrow, held, move_up, paddle_player, north, low`}) and finds the possible combinations of variables and concrete atoms (2^6 in this case). As with the bias that fewer terms are more preferred, we assume that better rules will use a smaller number of concrete term. For instance, in the case of *Pong*, the rule:

```
player_controls(paddle_player) :-
   entity(paddle_player).
```

would have full coverage of the positive examples in *Pong* but would do a poor job of covering any other game (except for say, *Breakout*, maybe). As such it is considered after the rule:

```
player_controls(X) :-
   entity(X).
```

Given all possible combinations of concrete and variable terms, we then consider whether a functor can safely subsume its terms and be considered. Continuing with the motivating example of:

```
precondition(control_event(
   player_input(up_arrow,held)),move_up).
result(move_up,
```

```
   moves(paddle_player,north,low)).
```
we see the functors:
```
control_event(player_input(up_arrow,held))
player_input(up_arrow,held)
moves(paddle_player,north,low)
```
A functor can safely be subsumed if it does not orphan any of the clauses considered. In this case, each could safely be subsumed, as the only term that ties the clauses together is `move_up`. For instance, the subsumption of
```
control_event(player_input(up_arrow,held))
```
leads to the rule:
```
precondition(X),move_up).
result(move_up,
   moves(paddle_player,north,low)).
```
However, if we were to consider the clauses:
```
result(move_up,
   moves(paddle_player,north,low))
precondition(overlaps(ball,paddle_player),
```
we see that neither `moves(paddle_player,north,low)` or `overlaps(ball,paddle_player)` could be subsumed in this case, as they both contain the linking term `paddle_player`.

Finally, we wish to be able to create rules with more expressivity than just $P \leftarrow A \wedge B \wedge ...$, and wish to consider rules that allow for the negation of clauses, i.e. $P \leftarrow A \wedge B \wedge \neg C \wedge \neg D \wedge ....$ However, we need to ensure that negated clauses remain safe constructs, i.e. they do not contain any variables that are found in any other clause. For instance the rule:
```
P(X) :-
   Q(X),
   not R(Y).
```
is unsafe as `Y` is not contained elsewhere, but
```
P(X) :-
   Q(X,Y),
   not R(Y).
```
is safe because `Y` is found in `Q(X,Y)`. As such, we only allow the negation of a clause if all variables in the clause are contained in at least one positive clause.

After finding all possible combinations of rules involving the abstractification of concrete terms, functor subsumption, and clause negation, we need to test the rules. At each point in the search we test the rule on each pair of game and positive examples found in that game. If a rule produces false positives it is ignored, and the rule of minimal complexity that covers the most positive examples per game is kept. This minimal complexity is determined by a lexicographic sorting, first sorting by number of concrete terms contained and then sorting by the number of terms found in the rule. If at any point a rule covers all positive examples in a game, then that rule is kept and the game is discarded. The search continues until all games are removed or a maximal depth is reached.

Evaluation

As with behavior generation, yet still even moreso, the evaluation of generated games is still very much an open question. For *Leda*, the evaluation can take the form of "How many of a desired set of rules can be found?" and "How usable are the learned rules for down-stream generation?" Unlike the work on behavior and level generation, the generated games from *Gemini* are not trying to match the distributions of some target set, but instead are trying to match certain design goals. This means that it is likely going to require user evaluations to determine whether the desired properties are found in the generated games.

Timeline

1. **Fall 2017** - Proposal, apply for faculty positions, work on *Leda*, *Mappy* $\rightarrow$ *Kamek* & *Agahnim*, work on CALE, work on *Agahnim*
2. **Winter 2017** - work on *Leda*, *Mappy* $\rightarrow$ *Kamek* & *Agahnim*, work on CALE, work on *Agahnim*, dissertation writing
3. **Spring 2017** - dissertation writing
4. **Summer 2017** - disseration writing/defense