# Finite Automata Extraction: Low-data World Models as Programs from Gameplay Video

Dave Goel
dgoel1@ualberta.ca
Department of Computing Science,
Alberta Machine Intelligence Institute
(Amii), University of Alberta
Edmonton, Alberta, Canada

Matthew Guzdial
guzdial@ualberta.ca
Department of Computing Science,
Alberta Machine Intelligence Institute
(Amii), University of Alberta
Edmonton, Alberta, Canada

Anurag Sarkar
sarkar4@ualberta.ca
Department of Computing Science,
Alberta Machine Intelligence Institute
(Amii), University of Alberta
Edmonton, Alberta, Canada

## Abstract

World models are often neural network-based models that attempt to approximate an entire video game. Existing world models are not practical for game developers due to their large scale, lack of accessibility, their prompt-based interfaces, and their nature as black box models where developers cannot access the code. In this paper, we propose an approach, Finite Automata Extraction (FAE), that learns a neuro-symbolic world model represented as programs in a novel domain-specific language (DSL): Retro Coder. The neuro-symbolic world model structure allows for low data learning, meaning we can learn these models from a single gameplay video. This model's structure means that developers can directly access code to change the behaviour of the model, and it's low data cost makes it more accessible. In a comparison to prior neural world model approaches, FAE learns a more precise model of the environment and it learns more general code than prior DSL-based approaches.

## CCS Concepts

• **Computing methodologies** → **Artificial intelligence**; **Computer vision**; **Machine learning**; **Semi-supervised learning settings**.

## Keywords

World Models, Video Games, Neuro-Symbolic, Machine Learning, Artificial Intelligence

**ACM Reference Format:**
Dave Goel, Matthew Guzdial, and Anurag Sarkar. 2026. Finite Automata Extraction: Low-data World Models as Programs from Gameplay Video. In *Foundations of Digital Games (FDG '26), August 10–13, 2026, Copenhagen, Denmark.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3815598.3815632

## 1 Introduction

Modern video games are primarily made using game engines[25]. Game developers are typically required to have technical depth and breadth in order to use these engines, needing knowledge of aspects such as programming languages, mathematics, physics concepts, AI implementations and so on. As a result, the barrier to entry to becoming a game developer can be quite high. With the arrival of the third AI summer, there has been a trend to develop tools that can lower this barrier [30]. Some industry researchers have recently proposed world models as a tool for game development [32].

World models are a compressed spatial and temporal learned representation of an environment [17]. They are predictive models that mimic an environment and approximate future states based on the actions of an agent. For example, creating a simplified, playable clone of Pac-Man in a neural network as in NVIDIA's GameGAN [32]. Researchers typically apply world model learning to reinforcement learning or other automated game playing approaches [17]. Recently, some industry researchers have moved towards motivating world model learning for game creation [23, 32]. They claim that world models can act as game engines, and a particular subset of the latent space/world model can be devoted to a single playable game. Thus game development would become as simple as searching through this latent space of possible games. However, there are some issues with this approach. These models do not have a game state or memory, and thus have a tendency to hallucinate or lose track of offscreen objects [29]. This could make it hard for developers to make a cohesive gameplay loop, since the mechanics in the environment would be unpredictable and unreliable. Since these playable world models are all neural networks, they are also a black box to the developer [4]. They cannot make changes to the behaviour of the model as they could with traditional game code. Training these models also requires a large amount of data and computation, resources which may not be available to the majority of developers. As a result, the only viable option for developers would be to use pretrained models, which further reduces their autonomy. Such models like Genie 3, can be accessed with a paid subscription [1]. Recently, OpenAI shut down SORA, a video generation model, raising concerns around long-term access.

If we had a world model learning approach that better suited the needs of game developers in terms of an editable code representation and only requiring low data, it could be a powerful tool for development. This paper presents Finite Automata Extraction (FAE): a novel neuro-symbolic approach to world model learning from a single gameplay video. A neuro-symbolic pipeline uses a combination of neural network architectures and symbolic approaches. Our pipeline learns a dictionary of sprites in a gameplay video using a neural network and then searches over Retro Coder, a novel Domain Specific Language (DSL) to extract symbolic programs for each of the learned sprites. A DSL is a custom simplified language

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*FDG '26, Copenhagen, Denmark*
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-2495-4/2026/08
https://doi.org/10.1145/3815598.3815632

created for a specific domain. This approach would give developers access to the underlying code behind the world model. This would remove the black box nature of a traditional world model, allowing developers to directly alter the mechanics of the game and remove unintended unpredictability. Our approach also uses substantially less data and is less computationally intensive than previous world model approaches. We argue that this would make it more accessible to more game developers. As an initial implementation, our pipeline targets simple grid based 2D games. In this paper we present the idea and initial evidence to support the feasibility of our pipeline.

We apply our approach on two domains, Pac-Man and River Raid and compare our results to GameGAN, a neural world model [18], and Game Engine Learning, a DSL-based world model [15]. In this paper, we present the following contributions:

- A novel neuro-symbolic approach to learn world models using gameplay video: Finite Automata Extraction.
- A novel Domain Specific Language: Retro Coder.
- The results of our evaluation comparing to existing neural and DSL-based world models.

## 2 Background

In this section we first cover the technical background knowledge needed to understand our work.

### 2.1 Program Synthesis

Program synthesis refers to the task of generating programs that satisfy user needs [13]. Commonly, developers define a search space of programs from which a desired program can be selected. Typically, this is done by selecting or writing a Domain Specific Language, as the search space in a custom DSL is usually much smaller than a Turing complete language. The developers then define a set of constraints or heuristic for the desired program [14]. For instance, they might give a series of input output pairs for the program. The goal of the search algorithm then would be to search for a program that satisfies most, if not all of these examples. Program synthesis has long been an area of interest for artificial intelligence research [20, 24, 33] with more recent efforts leveraging neural network-based approaches [2, 7, 10, 11].

### 2.2 MarioNette

MarioNette is a self-supervised neural network model that decomposes video frames into sprites, i.e distinct pieces of image data stored in a game[27]. It is a regularized Variational Auto-encoder that takes in video frames as input and reconstructs them. It does this by learning a sprite dictionary, a bounded set of neural sprites, i.e., sprite embedding learned by Marionette approximating true sprites. It uses this dictionary to reconstruct the frame. To do this, it employs a grid representation of the frame, where each grid cell contains a sprite from the sprite dictionary. The model has been found successful at this task for the domain of Super Mario Bros.. We leverage Marionette as the basis for the neural network in our approach. By learning sprites, we can avoid the requirements of prior DSL-based world models in having to provide all possible sprites as the developer [15].

## 3 Related Work

In this section we first cover prior work in program synthesis, given that Finite Automata Extraction (FAE) relies on it to learn appropriate programs in our Domain Specific Language, Retro Coder. We then cover prior work on world models.

### 3.1 Program Synthesis in Games

Within the specific context of games, program synthesis has been used for generating boss fights [5], solving grid-based games [26], generating strategies for RTS games [21] and generating goal-based minigames [6]. More commonly program synthesis is used for automated control. Gu et al. use program synthesis for representing policies for agents in the Overcooked environment [12]. Similarly Eberhardinger et al. synthesize programs to explain an agent's decision-making process in the Atari environment [9]. More recent work has also looked at defining a framework for leveraging large language models to assist with synthesizing programs for accomplishing a variety of in-game tasks [8]. The majority of this prior work does not focus on program synthesis for world model learning, we address exceptions in the next subsection.

### 3.2 World Models

A recent focus has been in training world models that enable real-time interaction via user inputs [22, 35]. While some of these works [1, 3, 23, 32, 36] are often presented as models capable of mimicking or generating games, they operate by generating video frames conditioned on user input in real-time and lack explicit knowledge of underlying game rules and mechanics. Due to their size, the majority of the recent world models cannot run in real-time on consumer hardware and are not open source. For these reasons we compare to only GameGAN in our experiments [18]. GameGAN uses a Generative Adveserial Network to predict the next frame, It takes in the previous frame and keyboard actions as its input.

Some prior work has looked at more explicitly modelling of game mechanics such as Game Engine Learning, which employed a Domain Specific Language of if-then rules in a program synthesis-based symbolic learning approach [15]. Our approach differs from this in two ways. Firstly, our pipeline includes learning the sprites, which is not the case with Game Engine learning. Secondly, we use a bottom up approach to learn the programs, as opposed to Game Engine Learning, which uses a top down approach. As a result, our approach learns rules that are more general. We compare against this approach as well in our experiments as the most similar example of purely symbolic prior work. There are other symbolic approaches, such as learning hybrid automata [28] but they depend on access to an emulator instead of purely gameplay video.

## 4 System Overview: Finite Automata Extraction

In this section, we cover our proposed neurosymbolic approach to learn programs from gameplay video: Finite Automata Extraction (FAE). We give an overview of our pipeline in Figure 1. We begin by splitting the gameplay video data into RGB frames. We then train a self-supervised model based on the Marionette architecture [27] on these frames to extract a neural sprite dictionary, a bounded set of sprites. After training the model, we extract a learned symbolic representation of the frames. We then learn a program for each

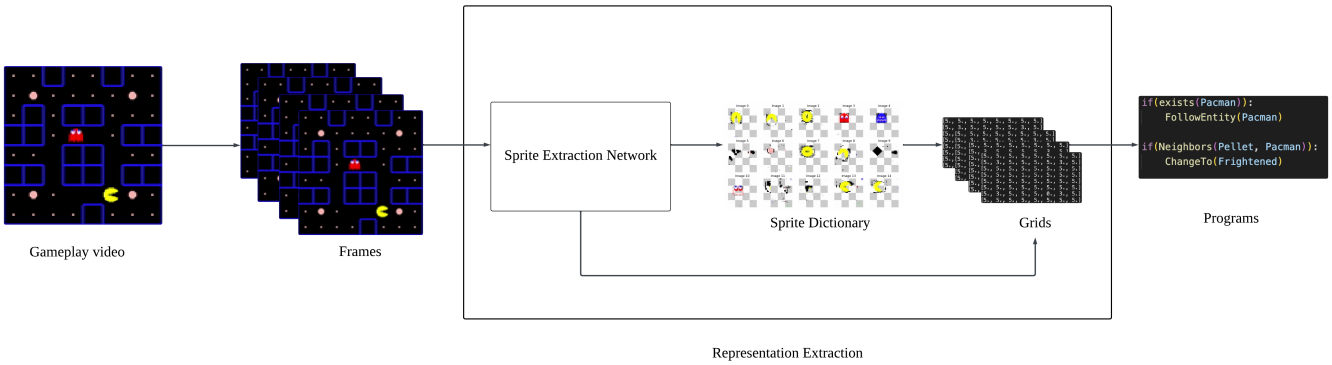


**Figure 1: System Overview of Finite Automate Extraction (FAE). We first extract frames from gameplay video, then learn a neural sprite dictionary and a symbolic representation of those frames, and then learn programs for each of the sprites in the sprite dictionary**

learned sprite to replicate their behaviour in the gameplay video. We learn each program by searching over Retro Coder, a novel domain-specific language.

## 4.1 Representation Extraction

In this paper, we use entities to refer to unique 2D game objects. Each entity has a Sprite-Class, which dictates it's appearence and behaviour. For example, Blinky, Pellets and Pac-Man would all be different Sprite-Classes. We require the set of all entities and the positional information of each entity in each frame for the later program learning step of FAE. We settled on a grid representation, where each grid is a matrix of symbolic values, where each value corresponds to a Sprite-Class. This enables us to handle collisions discretely. In order to obtain the grid representation, we chose to learn the sprite dictionary consisting of a maximum $l$ number of sprites using a variant of the Marionette architecture [27].

We modified the original Marionette network to add an option to specify the background image, which is helpful for domains with a constant background. We changed the model to subtract the background image from both the input and output to ensure the model optimized only the reconstruction of the changing sprites. The original Marionette has multiple different loss components, including one for frame reconstruction. We added an additional multiplicative weight ($\gamma$) to this loss to further bias the model towards better reconstructions, which incentivizes learned sprite quality. We set $\gamma = 10$ based on early experimentation.

After training, for all the RGB frames $\{F_1, ...F_n\}$, we extract their grid representations $\{G_1, ...G_n\}$. Marionette employs a latent representation of frames in terms of grids of sprite embeddings. Thus we could extract these grids via passing the frames through the model and using the softmax weights from the model to determine the sprite present in each grid cell. This results in a $w \times h$ grid where each cell references a learned sprite in the sprite dictionary. As shown in Figure 2, the sprite dictionary consists of repeated sprites. For instance, index 0,1,2,7,8,13 and 14 all have sprites that resemble Pac-Man. To resolve this, we group similar looking sprites that imply similar behaviour together into same Sprite-Class. We then modify the $w \times h$ grid so that each cell in the grid refers to the Sprite-Class of the learned sprite.

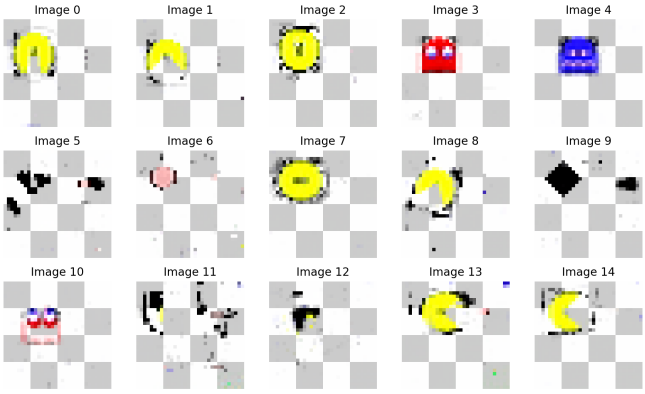


**Figure 2: Learned Sprite Dictionary**

## 4.2 Retro Coder

For this project we required a DSL that would support learning sprite behaviours. We thus defined our custom Domain Specific Language, Retro Coder. Retro Coder consists of condition functions (if) and action functions (then). Each if statement has one condition and one action. Since we designed FAE with 2D games in mind, we focused on common sprite behaviours and conditions for these behaviours. These consist of movement logic, entity transformation logic, and entity collision logic [34]. The condition functions are:

- $exists_in_map(sprite)$ checks if an entity of sprite-class $sprite$ exists in the frame
- $neighboring(sprite)$ checks if the entity of the program itself is next to an entity of sprite-class $sprite$,
- $neighbours(sprite1, sprite2)$ checks if an entity of sprite-class $sprite1$ is next to an entity of sprite-class $sprite2$. Since we are using a grid representation where one entity can exist

at one location in a moment of time, we treat neighbouring sprites as collisions.

The action functions are:

- $follow_entity(sprite)$ moves this entity toward the nearest entity of sprite-class $sprite$
- $follow_direction(direction)$ moves this entity in a direction (Up, down, left, right).
- $change_to_sprite(sprite)$ changes the entity's sprite-class to $sprite$
- $follow_target_location(x, y)$, which moves the entity towards to location $x, y$.

In the above we note that at times multiple entities of the same sprite-class. may appear on screen. For example, in Pac-man there are multiple pellets on screen at once.

## 4.3 Search Algorithm

```
programs ← [ ];
for sprite ∈ sprites do
    program ← [ ];
    for
     batch ∈ (listofIndicesWhereSpriteExists, batchSize)
     do
        distanceDecreased ← True;
        while distanceDecreased == True do
            distanceDecreased ← False;
            neighbours ← NeighbourPrograms();
            bestNeighbour ← program;
            for neighbour ∈ neighbours do
                if D(batch, neighbour) <
                 D(batch, bestNeighbour) then
                    bestNeighbour ← neighbour;
                    distanceDecreased ← True;
                end
            end
            if bestNeighbour ≠ [ ] then
                program ← bestNeighbour;
            end
        end
    end
    programs.append(program);
end
```

**Algorithm 1:** Pseudocode for Search Algorithm

This section covers the program synthesis step of FAE. The first step is that after we have our sprite-classes, we must categorize the sprite-classes into Learnable and Non-learnable sprites. This might be based on domain knowledge but can also be general at times. For example, the sprite-class that is controlled by the player would be Non-learnable as it does not have a ground truth program. We then learn a program for all of these hand-identified Learnable sprite-classes with our search algorithm, as seen in 1. We learn one program at a time, as this greatly reduces the search space. For a sprite-class $s$, we first scan $\{G_1, ...G_n\}$ to get a list of indices $\{I_1, ...I_s\}$ where $G_{I_i}$ contain $s$. We then go through the indices sequentially, for every $i$, compute the distance $D = \sum_{x=i}^{i+b} A.E(G_{I_{x+1}}, G_{I_x})$ where $A.E$ is the Absolute Error and $b$ is the batch size. We set $b = 3$ based on experimentation. We then search for neighbour programs to minimize $D$. The neighbour programs can either have an additional if-then statement, or remove an if-then statement from the current program. We consider all possible additions and all possible deletions in the $NeighbourPrograms()$ function.

# 5 Evaluation

Finite Automate Extraction allows us to process a single gameplay video of a 2D game and extract a learned set of sprites and a learned set of programs, one associated with each sprite. We evaluate our model from two perspectives: the quality of the learned programs, which assesses the backend of the model, and the quality of the world model as an approximation of a game, which assesses the front end. We test our model on two domains, Pac-Man and River Raid. Since this is an initial implementation, and thus primarily targeted at simple grid based 2D games, both of these meet the criteria while differing in terms of the types of games, with one focused on combat and the other on collecting.

The programs were first learned using a single training video of each game, after which we used a test video for evaluation. To generate the frames from the model, we used 2 approaches: Next Frame and Total Simulation. For Next Frame we use the ground truth frame at each time step $t$ to predict the next frame with our model at time step $t + 1$. For Total Simulation, we only use the ground truth frame at time step $t = 0$, after which the predicted frame at timestep $t$ is used to predict the frame at timestep $t+1$. Next Frame allows us to evaluate local coherency while Total Simulation evaluates global coherency of the learned programs.

## 5.1 Domains

Pac-Man is an arena-based game where the player navigates a level and collect pellets while avoiding ghosts that can kill Pac-Man if they make contact. The game normally features 4 ghosts, Blinky, Inky, Pinky and Clyde. We used a simplified version of the game developed with Unity by removing all the ghosts except Blinky. We also reduced the size of the map from 20x31 to 8x8. We recorded the gameplay videos for Pac-Man ourselves. We extracted frames at a rate of four frames per second based on the average human reaction time is 0.25s [31].

The width $w$ and height $h$ of the learned grid were set to 8, and the maximum number of sprites in the dictionary $l$ was set to 15. We trained the model for 1000 epochs.

River Raid is a scrolling shooter game where the player controls an airplane and has to shoot obstacles. While recording, we cropped the screen to remove background details that were not involved in the gameplay to better focus on the sprites and their behaviours. We used the Atari gym environment to record the gameplay videos for River Raid. We used the same hyper parameters from Pac-Man with the exception that $l$ was set to 16.

## 5.2 Baselines

For a neural world model baseline, we used GameGAN [18]. We believe that this model is a reasonable representation of current world models. Further the largest of modern world models like

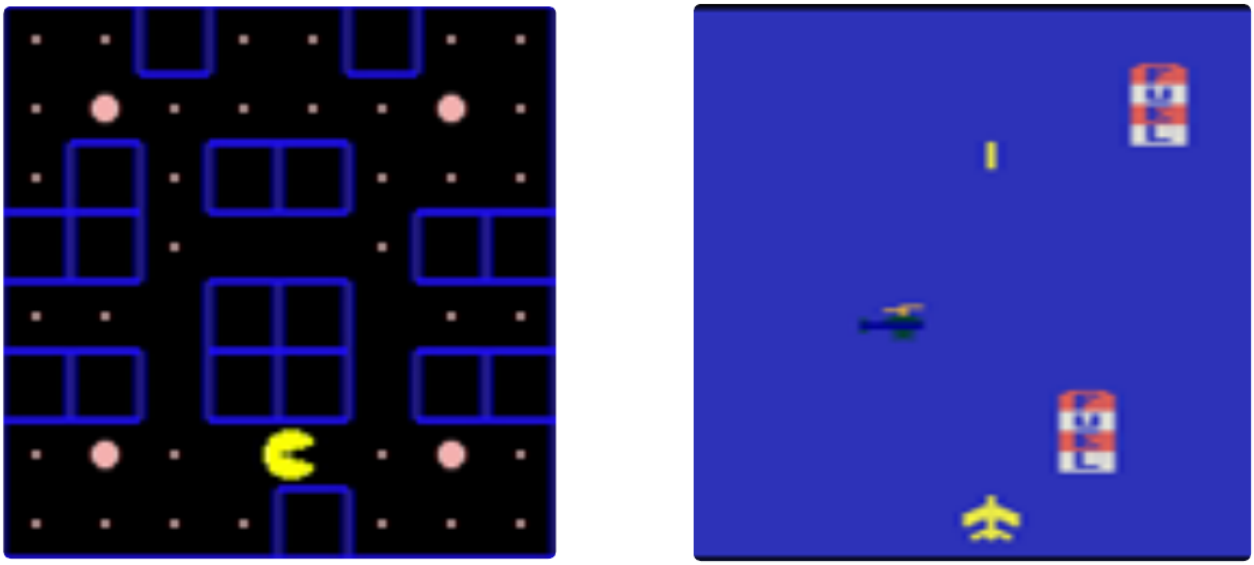

**Figure 3: Screenshots of gameplay of both domains, Pac-Man and River Raid.**

GameNGen and Genie 3 are not accessible to the public, and could not be retrained with our resources even if they were. We trained GameGAN on the same Pac-Man and River Raid data we used to train FAE. Notably this is far less data than GameGAN had in the original publication. Our goal was to see how well a neural world model could do with this limited amount of data.

We use Game Engine Learning [16] as a DSL-based world model baseline. Game Engine Learning is an existing symbolic world model that uses a top down approach, we wanted to see how well our algorithm performs again GEL. We again used the same training data that we used for FAE.

### 5.3 Metrics

For a reconstruction quality metric we used **FID**, which is a standard metric for video generation [19]. This allows us to capture a notion of the perceptual quality of the learned world models. In addition to using this established metric, we created another metric, **Prediction Error** $\sum_{i=0}^{l} MAE(T_{i+1} - T_i, M_{i+1} - T_i)$ to assess the distance between the ground truth set of frames $T_1, ...T_l\}$ and generated set of frames $M_1, ...M_l\}$. This allowed us to capture a notion of precision over the sprites instead of over pixels.

It is difficult to assess the quality of learned programs. We note that the above metrics do indirectly capture some notion of the functional quality of the learned programs, especially **Prediction Error**. Outside of this, we also report the number of conditions, as a way to roughly approximate the generalizability of the learned programs. We only report this for FAE and Game Engine Learning, as GameGAN learns no explicit programs.

## 6 Results

In this section we first cover the learned programs for the Pac-Man and River Raid domains. Part of the benefits of FAE is that we can easily extract the learned programs and present them in a readable format. This also allows us to discuss some particulars of the learned programs that may be non-obvious from the metrics-based analysis. We then cover our metric results comparing the performance of FAE against the baselines within both domains.

### 6.1 Learned Programs

Using our variant of Marionette, FAE was able to learn six distinct sprite-classes for Pac-Man: PACMAN, BLINKY, GHOST, EYES, PELLET, EMPTY. PACMAN indicates the player, BLINKY indicates the red ghost sprite, GHOST indicates the blue ghost sprite that BLINKY turns into when PACMAN eats a PELLET, EYES indicates the eye sprite a GHOST turns into when eaten, and EMPTY represents empty space. Notably these names are given only for reporting purposes and do not exist in the system. FAE represents these entities are indices in the learned sprite dictionary. We could not learn a sprite-class for the normal pellets due to their size, which is a limitation of Marionette. We include a human-readable version of the learned programs in Table 1 and Table 3 using these entity names. The DSL encodes If and Then statements implicitly, as such we prepended the "IF" and "THEN" statements. However, the simplicity of this translation and the associated programs themselves indicates a strong potential for explainability.

We also included the closest ground truth programs for the two domains in Table 2 and Table 4. Do note that there were certain behaviours that could not be captured in our DSL and therefore the programs. We cover these behaviours later in this section.

| Sprite | Learned Program |
|---|---|
| **BLINKY** | IF (exists in map(PACMAN))<br>THEN follow entity(PACMAN)<br>IF (neighboring entities(PACMAN, PELLET))<br>THEN change to entity(GHOST)<br>IF (is neighboring(GHOST))<br>THEN change to entity(PACMAN) |
| **GHOST** | IF (is neighboring(PACMAN))<br>THEN change to entity(PACMAN)<br>IF (is neighboring(PACMAN))<br>THEN change to entity(EMPTY) |
| **PELLET** | IF (is neighboring(PACMAN))<br>THEN change to entity(PACMAN) |
| **EYES** | IF (exists in map(PACMAN))<br>THEN change to entity(EMPTY) |

**Table 1: Programs Generated with Finite Automata Extractor for Pac-Man**

| Sprite | Learned Program |
|---|---|
| **BLINKY** | IF (exists in map(PACMAN))<br>THEN follow entity(PACMAN)<br>IF (neighboring entities(PACMAN, PELLET))<br>THEN change to entity(GHOST) |
| **GHOST** | IF (is neighboring(PACMAN))<br>THEN change to entity(EYES) |
| **PELLET** | IF (is neighboring(PACMAN))<br>THEN change to entity(PACMAN) |
| **EYES** | IF (exists in map(PACMAN))<br>THEN follow target location(4,4) |

**Table 2: Ground Truth Programs**

| Sprite | Learned Program |
|---|---|
| **PELLET** | IF (exists in map(PLANE))<br>THEN follow direction(UP)<br>IF (exists in position(4,1))<br>THEN change to entity(EMPTY)<br>IF (exists in position(4,3))<br>THEN change to entity(EMPTY)<br>IF (exists in position(3,1))<br>THEN change to entity(EMPTY) |
| **BLACKBOAT** | |
| **FUEL** | IF (exists in position(7,7))<br>THEN follow target(6,7)<br>IF (is neighboring(PELLET))<br>THEN change to entity(EMPTY) |
| **GREENBOAT** | IF (exists in map(PLANE))<br>THEN follow target(1,2) |

**Table 3: Programs Generated with Finite Automata Extractor for River Raid**

| Sprite | Learned Program |
|---|---|
| **PELLET** | IF (exists in map(PLANE))<br>THEN follow direction(UP)<br>IF (is neighboring(BLACKBOAT))<br>THEN change to entity(EMPTY)<br>IF (is neighboring(GREENBOAT))<br>THEN change to entity(EMPTY) |
| **BLACKBOAT** | IF (exists in map(PLANE))<br>THEN follow direction(DOWN)<br>IF (is neighboring(PELLET))<br>THEN change to entity(EMPTY) |
| **FUEL** | IF (exists in map(PLANE))<br>THEN follow direction(DOWN)<br>IF (is neighboring(PELLET))<br>THEN change to entity(EMPTY) |
| **GREENBOAT** | IF (exists in map(PLANE))<br>THEN follow direction(DOWN)<br>IF (is neighboring(PELLET))<br>THEN change to entity(EMPTY) |

**Table 4: Ground Truth Programs for River Raid**

| Method | FID(train) | FID(test) | Error(train) | Error(test) | Conditions |
|---|---|---|---|---|---|
| GameGAN | 10.17 | 10.46 | 35.31 | 40.64 | - |
| Game Engine Learning | 0.25 | 0.39 | 10.46 | 6.91 | 73±22.55 |
| Finite Automata Extraction | 0.23 | 0.28 | 10.60 | 7.06 | **1.75±0.83** |

**Table 5: Comparison of methods on FID, Predicted Error and average number of conditions for Pac-Man, with the Next Frame approach.**

For Pac-Man, we identified PACMAN and EMPTY as Non-learnable sprite-classes, PACMAN due to player control and BLANK due to a lack of dynamic information.

Even the ground truth program does not fully capture the correct behaviour for GHOST, indicating an issue with our DSL. In the actual game, GHOST moves in random directions. However, the

| Method | FID(train) | FID(test) | Error(train) | Error(test) | Conditions |
|---|---|---|---|---|---|
| GameGAN | 10.17 | 10.46 | 35.31 | 40.64 | - |
| Game Engine Learning | 0.19 | 0.23 | 10.87 | 7.16 | 73±22.55 |
| Finite Automata Extraction | 0.28 | 0.29 | 10.37 | 7.56 | **1.75±0.83** |

**Table 6: Comparison of methods on FID, Predicted Error and average number of conditions for Pac-Man, with the Total Simulation approach.**

| Method | FID(train) | FID(test) | Error(train) | Error(test) | Conditions |
|---|---|---|---|---|---|
| GameGAN | 58.62 | 58.81 | 143.62 | 144.2 | - |
| Game Engine Learning | 0.37 | 0.32 | 82.42 | 82.85 | 125.42±19.58 |
| Finite Automata Extraction | 0.29 | 0.25 | 82.60 | 83.03 | **2.00±1.87** |

**Table 7: Comparison of methods on FID, Predicted Error and average number of conditions for River Raid, with the Next Frame approach.**

| Method | FID(train) | FID(test) | Error(train) | Error(test) | Conditions |
|---|---|---|---|---|---|
| GameGAN | 58.62 | 58.81 | 143.62 | 144.2 | - |
| Game Engine Learning | 0.17 | 0.14 | 83.44 | 84.14 | 125.42±19.58 |
| Finite Automata Extraction | 0.14 | 0.13 | 83.58 | 84.04 | **2.00±1.87** |

**Table 8: Comparison of methods on FID, Predicted Error and average number of conditions for River raid, with the Total Simulation approach.**

current version of our DSL cannot capture random behaviour. As such this was not possible for FAE to learn. FAE successfully learned that BLINKY should always follow PACMAN, which can be seen in the first condition of BLINKY. It also predicts that BLINKY should change into GHOST when PACMAN collides or eats a PELLET. The third learned condition in BLINKY never gets satisfied in the game as the GHOST and BLINKY sprites are never together in a frame. This condition was learned due to the grids not being accurate all the time. Occasionally, MarioNette would hallucinate and return a grid with some sprites that do not actually exist in the true frame. The model learned that PELLET should change into PACMAN once it has been eaten by PACMAN. This may seem odd, but has the desired effect, a PELLET sprites should disappear when eaten by PACMAN, thus the PELLET turning into the PACMAN that ate it has the same effect. The model could not replicate the ground truth behaviour for EYES. This is because there were not enough frames in the training video to capture it's behaviour. The GHOST does learn about being eaten by PACMAN, however, instead of turning into EYES like in the original game, it just disappears. These results show that our system has learned two of the four programs correctly. It should be noted that these two sprite-classes occupy considerably more screen time than the other two. We think this is a reasonable performance from a single gameplay example and could meaningfully support many game development tasks. We provide an example in the Case Study section.

For River Raid, we also learned six sprite-classes, PLANE, FUEL, GREENBOAT, BLACKBOAT and PELLET. PLANE indicates the triangular sprite that the player controls, FUEL indicates the can sprite, GREENBOAT and BLACKBOAT represents the green and black boat sprites respectively. The translated programs for this domain can be seen in 3. Like Pac-Man, we identified PLANE and EMPTY as Non-learnable sprite-classes.

While the ground truth programs for the game are accurate, there is one mechanic that they cannot cover. In the original game, BLACKBOAT, FUEL and GREENBOAT get randomly spawned on the top row. However, our current DSL cannot capture this behaviour. The model successfully learned that PELLET should always be moving up. However, it also learned to change to EMPTY for arbitrary positions. The model did not learn any code for BLACKBOAT, which we believe was because the sprite-class appeared rarely in the training video, much like EYES in Pac-Man. FUEL is an interesting case since the size of the sprite is much bigger. As a result, it alternated between occupying two and three grid cells for each frame, depending on its position. This led to the algorithm not being able to learn it's movement behaviour. However, it did learn the collision behaviour of the original game as it changes to empty upon colliding with PELLET. The behaviour for GREENBOAT could not be replicated, which we believe was due to the same reason as BLACKBOAT. These results are less favourable than those of Pac-Man, and demonstrate the importance of DSL choices, since we assumed all sprites would occupy a single grid cell in our DSL.

## 6.2 Comparison with Baselines

In this subsection we discuss the results of our metrics-based comparison with our two baselines. We first go over the results of the metrics for Pac-Man. The results for the two variations can be seen in Tables 5 and 6. These show the FID, Predicted Error, and average number of conditions across the models. Since GameGan does

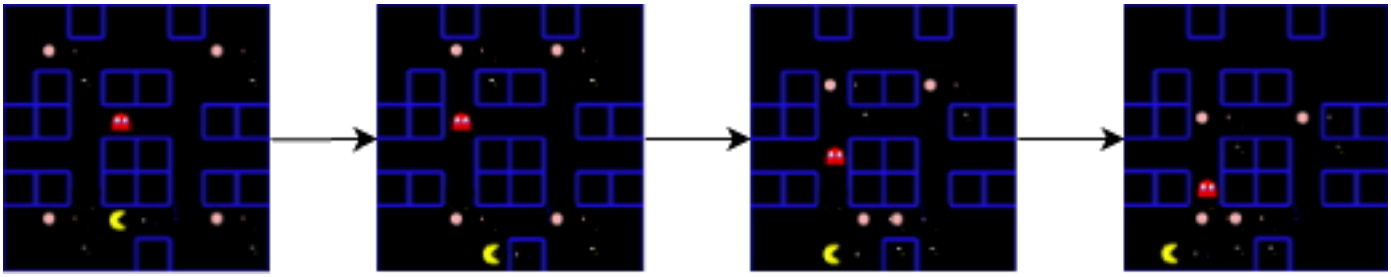

**Figure 4: Screenshots of gameplay of Pellet Chaser**

not produce any code, we cannot measure the average number of conditions. While running the Game Engine Learning System, the algorithm failed to converge due to not being able to make a perfect prediction and so we take the best performing program which follows the practices from prior work [16]. The average number of conditions were fewer for Finite Automata Extraction than Game Engine Learning (GEL). This shows that the code extracted by our system was more general as fewer conditions would lead to more cases of learned code running. The difference in the number of conditions is based on the way the different algorithms work, GEL starts with the maximum number of conditions then reduces whereas we add conditions as needed.

We trained GameGAN to until convergence. Due to its neural network nature and the low amount of data, it had the worst performance.

For the Next Frame approach, across the baselines, FAE had the lowest FID for both the train and test video, though the results are comparable with GEL. This is due to the fact that GameGAN was unable to learn a non-random model from the amount of available training data. Further, while GEL may appear to have similar performance this is due to it only predicting the passed in prior frame exactly. This indicates GEL did not learn transferrable information to this test video from the training video. However, predicting the prior frame is a strong baseline itself as established by prior work [16]. For the same reason, the Predictive Error for GEL and FAE is again comparable.

For the Total Simulation approach, GEL and FAE's results were again comparable, although GEL had a lower FID for both test and train. This is because GEL did not make any changes to the frame, so every frame output from the model consists of the same image. On the other hand, since FAE was not able to learn the behaviour of GHOST and EYES sprites correctly, the image sequence completely diverges as soon as one of the two sprites appear in the ground truth video. The Predicted Errors are inconclusive since they are lower for FAE for the train video and lower for GEL for the test video.

For River Raid, the FID and Predicted Error results for GameGAN were again the worst due to model's inability to learn a non-random model. The number of conditions were also again less for FAE than GameGAN. Notably, while the FID scores for GEL and FAE were again comparable, FAE had lower FID for both the approaches for both the train and test videos.

In the original River Raid game, fuel and boats are spawned offscreen randomly. This was not possible for our current DSL since we would need to model the area outside of the game screen, which is currently not possible in the current version of Retro Coder.

## 7 Case Study

In this section, we show a use case for how game developers could use our pipeline. We take the Pac-man and River Raid world model programs that we have learned, and then experiment with variations of them by altering the code and the sprites. We first make changes to the learned programs so that they resemble the ground truth programs, we assume the developer would have made changes to correct the issues from the learned programs, additionally we think these changes are small enough to be reasonable.

### 7.1 Example 1: Pellet Chaser

We changed the code in Pac-Man for PELLET so that it always chases PACMAN. We did this by changing only the if statement in PELLET. This changed the dynamics of the game quite a bit, as the pellets now all seek Pac-Man as seen in Figure 1. This changes the gameplay from collecting into just avoiding the ghost for long enough until the pellets can catch up with Pac-Man. While we don't think this is a particularly good game on its own, we do find it interesting, and it's the kind of simple thought experiment that FAE could afford to game developers.

| Sprite | Learned Program |
|---|---|
| **BLINKY** | IF (exists in map(PACMAN))<br>THEN follow entity(PACMAN)<br>IF (neighboring entities(PACMAN, PELLET))<br>THEN change to entity(GHOST) |
| **GHOST** | IF (is neighboring(PACMAN))<br>THEN change to entity(EYES) |
| **PELLET** | IF (exists in map(PACMAN))<br>THEN follow entity(PACMAN) |
| **EYES** | IF (exists in map(PACMAN))<br>THEN follow target location(4,4) |

**Table 9: Program for Pellet Chaser**

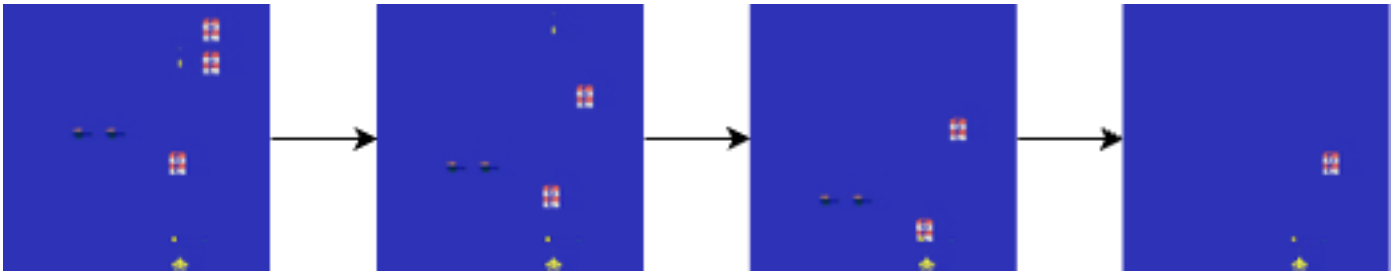

**Figure 5: Screenshots of gameplay of River Raid Power Up**

## 7.2 Example 2: River Raid Power Up

We changed the code in River Raid so that the boats get destroyed whenever PLANE touches FUEL. We did this by adding an if statement to both the boats, the condition being that if PLANE and FUEL are neighbouring each other, the boats change to EMPTY. While this maintains the core loop of the game, it changes the dynamics substantially as the player now has the option to collide with FUEL to destroy the obstacles essentially giving the player an area-of-effect (AOE) attack that did not exist in the original game. Again, this is likely not a better game than River Raid, but it allows for an interesting and fast exploration of a variation of the game.

| Sprite | Learned Program |
|---|---|
| **PELLET** | IF (exists in map(PLANE))<br>THEN follow direction(UP)<br>IF (is neighboring(BLACKBOAT))<br>THEN change to entity(EMPTY)<br>IF (is neighboring(GREENBOAT))<br>THEN change to entity(EMPTY) |
| **BLACKBOAT** | IF (exists in map(PLANE))<br>THEN follow direction(DOWN)<br>IF (is neighboring(PELLET))<br>THEN change to entity(EMPTY)<br>IF (neighboring entites(PLANE, FUEL))<br>THEN change to entity(EMPTY) |
| **FUEL** | IF (exists in map(PLANE))<br>THEN follow direction(DOWN)<br>IF (is neighboring(PELLET))<br>THEN change to entity(EMPTY) |
| **GREENBOAT** | IF (exists in map(PLANE))<br>THEN follow direction(DOWN)<br>IF (is neighboring(PELLET))<br>THEN change to entity(EMPTY)<br>IF (neighboring entites(PLANE, FUEL))<br>THEN change to entity(EMPTY) |

**Table 10: Program for Pellet Chaser**

## 8 Limitations

In this paper, we proposed a framework to learn code from a gameplay video. Our results indicate that we can learn more general programs than a prior DSL-based world model, and can outperform a neural network world model on this low amount of data. However, there are still some limitations to our existing approach. In our current set up, the grid representations that were extracted from the Marionette variant occasionally return inaccurate frames. For example, occasionally a sprite can be between two grid cells. In these instances, our grid representation hallucinated that there was the same sprite in both of the cells. This caused the model to learn conditions that were not true. Perhaps in the future, we can tweak the Marionette Architecture to take this into account. Another aspect that the current representation does not handle is the rotation of the sprites. The sprites currently are rendered in only one rotation. As noted in the case of River Raid, the grid based representation in its current state also does not handle sprites with different sizes well.

In its current state, Retro Coder also has some limitations. Since we learn programs for individual sprites, we cannot create a central manager or control program to handle sprite spawning. Additionally, the DSL does not take time into account, and can thus not make behaviours that are time dependent. For example, a certain sprite changing into a different sprite every $x$ seconds. The DSL also cannot replicate randomized behaviour.

## 9 Conclusion

In this paper, we proposed a neuro-symbolic approach to learn a world model as code from gameplay video: Finite Automata Extraction (FAE). We did this by training a model to convert the video frames into a symbolic grid representation. Additionally we defined a novel DSL, Retro Coder as the search space of the world model programs. FAE uses lower data compared to current neural network world models. We tested our pipeline in two domains, Pac-Man and River Raid. Our model was able to learn accurate code for some of the sprites and showed equivalent or better results compared to the existing baselines.

## Acknowledgments

This work was funded by the Canada CIFAR AI Chairs Program. We acknowledge the support of the Alberta Machine Intelligence Institute (Amii).

## References

[1] Philip J. Ball, Jakob Bauer, Frank Belletti, Bethanie Brownfield, Ariel Ephrat, Shlomi Fruchter, Agrim Gupta, Kristian Holsheimer, Aleksander Holynski, Jiri Hron, Christos Kaplanis, Marjorie Limont, Matt McGill, Yanko Oliveira, Jack Parker-Holder, Frank Perbet, Guy Scully, Jeremy Shar, Stephen Spencer, Omer Tov,

Ruben Villegas, Emma Wang, Jessica Yung, Cip Baetu, Jordi Berbel, David Bridson, Jake Bruce, Gavin Buttimore, Sarah Chakera, Bilva Chandra, Paul Collins, Alex Cullum, Bogdan Damoc, Vibha Dasagi, Maxime Gazeau, Charles Gbadamosi, Woohyun Han, Ed Hirst, Ashyana Kachra, Lucie Kerley, Kristian Kjems, Eva Knoepfel, Vika Koriakin, Jessica Lo, Cong Lu, Zeb Mehring, Alex Moufarek, Henna Nandwani, Valeria Oliveira, Fabio Pardo, Jane Park, Andrew Pierson, Ben Poole, Helen Ran, Tim Salimans, Manuel Sanchez, Igor Saprykin, Amy Shen, Sailesh Sidhwani, Duncan Smith, Joe Stanton, Hamish Tomlinson, Dimple Vijaykumar, Luyu Wang, Piers Wingfield, Nat Wong, Keyang Xu, Christopher Yew, Nick Young, Vadim Zubov, Douglas Eck, Dumitru Erhan, Koray Kavukcuoglu, Demis Hassabis, Zoubin Gharamani, Raia Hadsell, Aäron van den Oord, Inbar Mosseri, Adrian Bolton, Satinder Singh, and Tim Rocktäschel. 2025. Genie 3: A New Frontier for World Models. (2025).

[2] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *International Conference on Learning Representations*.

[3] Jake Bruce, Michael D Dennis, Ashley Edwards, Jack Parker-Holder, Yuge Shi, Edward Hughes, Matthew Lai, Aditi Mavalankar, Richie Steigerwald, Chris Apps, et al. 2024. Genie: Generative interactive environments. In *Forty-first International Conference on Machine Learning*.

[4] Vanessa Buhrmester, David Münch, and Michael Arens. 2019. Analysis of Explainers of Black Box Deep Neural Networks for Computer Vision: A Survey. arXiv:1911.12116 [cs.AI] https://arxiv.org/abs/1911.12116

[5] Eric Butler, Kristin Siu, and Alexander Zook. 2017. Program synthesis as a generative method. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*. 1–10.

[6] Guy Davidson, Graham Todd, Julian Togelius, Todd M Gureckis, and Brenden M Lake. 2025. Goals as reward-producing programs. *Nature Machine Intelligence* 7, 2 (2025), 205–220.

[7] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*. PMLR, 990–998.

[8] Manuel Eberhardinger, James Goodman, Alexander Dockhorn, Diego Perez-Liebana, Raluca D Gaina, Duygu Çakmak, Setareh Maghsudi, and Simon Lucas. 2024. From Code to Play: Benchmarking Program Search for Games Using Large Language Models. *arXiv preprint arXiv:2412.04057* (2024).

[9] Manuel Eberhardinger, Johannes Maucher, and Setareh Maghsudi. 2023. Learning of generalizable and interpretable knowledge in grid-based reinforcement learning environments. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, Vol. 19. 203–214.

[10] Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. 2018. Learning libraries of subroutines for neurally–guided bayesian program induction. *Advances in Neural Information Processing Systems* 31 (2018).

[11] Kevin Ellis, Lionel Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lore Anaya Pozo, Luke Hewitt, Armando Solar-Lezama, and Joshua B Tenenbaum. 2023. Dreamcoder: growing generalizable, interpretable knowledge with wake–sleep bayesian program learning. *Philosophical Transactions of the Royal Society A* 381, 2251 (2023), 20220050.

[12] Yin Gu, Qi Liu, Zhi Li, and Kai Zhang. 2024. KnowPC: Knowledge-Driven Programmatic Reinforcement Learning for Zero-shot Coordination. *arXiv preprint arXiv:2408.04336* (2024).

[13] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Found. Trends Program. Lang.* 4, 1–2 (July 2017), 1–119. doi:10.1561/2500000010

[14] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119.

[15] Matthew Guzdial, Boyang Li, and Mark O Riedl. 2017. Game Engine Learning from Video.. In *IJCAI*. 3707–3713.

[16] Matthew Guzdial, Boyang Li, and Mark O. Riedl. 2017. Game engine learning from video. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence* (Melbourne, Australia) *(IJCAI'17)*. AAAI Press, 3707–3713.

[17] David Ha and Jürgen Schmidhuber. 2018. World Models. (2018). doi:10.5281/ZENODO.1207631

[18] Seung Wook Kim, Yuhao Zhou, Jonah Philion, Antonio Torralba, and Sanja Fidler. 2020. Learning to Simulate Dynamic Environments with GameGAN. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[19] Yaofang Liu, Xiaodong Cun, Xuebo Liu, Xintao Wang, Yong Zhang, Haoxin Chen, Yang Liu, Tieyong Zeng, and Raymond Chan. 2024. EvalCrafter: Benchmarking and Evaluating Large Video Generation Models. 22139–22149. doi:10.1109/CVPR52733.2024.02090

[20] Zohar Manna and Richard Waldinger. 1975. Knowledge and reasoning in program synthesis. *Artificial intelligence* 6, 2 (1975), 175–208.

[21] Julian RH Marino, Rubens O Moraes, Tassiana C Oliveira, Claudio Toledo, and Levi HS Lelis. 2021. Programmatic strategies for real-time strategy games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 381–389.

[22] Willi Menapace, Stephane Lathuiliere, Sergey Tulyakov, Aliaksandr Siarohin, and Elisa Ricci. 2021. Playable video generation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 10061–10070.

[23] Jack Parker-Holder, Philip Ball, Jake Bruce, Vibhavari Dasagi, Kristian Holsheimer, Christos Kaplanis, Alexandre Moufarek, Guy Scully, Jeremy Shar, Jimmy Shi, Stephen Spencer, Jessica Yung, Michael Dennis, Sultan Kenjeyev, Shangbang Long, Vlad Mnih, Harris Chan, Maxime Gazeau, Bonnie Li, Fabio Pardo, Luyu Wang, Lei Zhang, Frederic Besse, Tim Harley, Anna Mitenkova, Jane Wang, Jeff Clune, Demis Hassabis, Raia Hadsell, Adrian Bolton, Satinder Singh, and Tim Rocktäschel. 2024. Genie 2: A Large-Scale Foundation World Model. (2024). https://deepmind.google/discover/blog/genie-2-a-large-scale-foundation-world-model/

[24] Amir Pnueli and Roni Rosner. 1989. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 179–190.

[25] Vinay Sharma, Sunil Kumawat, Ravi Jangid, Vaibhav Kumawat, Yash Bansal, Vishal Shrivastava, and Akhil Pandey. 2024. Game engines: A survey. 020060. doi:10.1063/5.0239665

[26] Tom Silver, Kelsey R Allen, Alex K Lew, Leslie Pack Kaelbling, and Josh Tenenbaum. 2020. Few-shot bayesian imitation learning with logical program policies. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 10251–10258.

[27] Dmitriy Smirnov, Michaël Gharbi, Matthew Fisher, Vitor Guizilini, Alexei A. Efros, and Justin Solomon. 2021. MarioNette: self-supervised sprite learning. In *Proceedings of the 35th International Conference on Neural Information Processing Systems (NIPS '21)*. Curran Associates Inc., Red Hook, NY, USA, Article 420, 12 pages.

[28] Adam Summerville, Joe Osborn, Christoffer Holmgård, and Daniel W Zhang. 2017. Mechanics automatically recognized via interactive observation: Jumping. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*. 1–10.

[29] Praneet Suresh, Jack Stanley, Sonia Joseph, Luca Scimeca, and Danilo Bzdok. 2025. From Noise to Narrative: Tracing the Origins of Hallucinations in Transformers. arXiv:2509.06938 [cs.LG] https://arxiv.org/abs/2509.06938

[30] Alexandru Ternar, Alena Denisova, João Cunha, Kultima Annakaisa, and Christian Guckelsberger. 2025. Generative AI in Game Development: A Qualitative Research Synthesis. doi:10.48550/arXiv.2509.11898

[31] E. A. C. Thomas. 1969. Information Theory of Choice-Reaction Times. By D. R. J. Laming. *Brit. J. Math. Statist. Psych.* 22, 1 (1969), 103–104. arXiv:https://bpspsychub.onlinelibrary.wiley.com/doi/pdf/10.1111/j.2044-8317.1969.tb00423.x doi:10.1111/j.2044-8317.1969.tb00423.x

[32] Dani Valevski, Yaniv Leviathan, Moab Arar, and Shlomi Fruchter. 2025. Diffusion Models Are Real-Time Game Engines. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=P8pqeEkn1H

[33] Richard J Waldinger and Richard CT Lee. 1969. PROW: A step toward automatic program writing. In *Proceedings of the 1st international joint conference on Artificial intelligence*. 241–252.

[34] Noah Wardrip-Fruin. 2020. *How Pac-Man Eats*. The MIT Press. doi:10.7551/mitpress/10641.001.0001

[35] Jialong Wu, Shaofeng Yin, Ningya Feng, Xu He, Dong Li, Jianye Hao, and Mingsheng Long. 2024. ivideogpt: Interactive videogpts are scalable world models. *Advances in Neural Information Processing Systems* 37 (2024), 68082–68119.

[36] Jiwen Yu, Yiran Qin, Xintao Wang, Pengfei Wan, Di Zhang, and Xihui Liu. 2025. Gamefactory: Creating new games with generative interactive videos. *arXiv preprint arXiv:2501.08325* (2025).