Proceedings of the Twentieth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2024)

Mechanic Maker: Accessible Game Development Via Symbolic Learning Program Synthesis

Megan Sumner*, **Vardan Saini***, **Matthew Guzdial***
University of Alberta, Alberta Machine Intelligence Institute
Edmonton, Canada
msumner@ualberta.ca, vardan1@ualberta.ca, guzdial@ualberta.ca

Abstract

Game development is a highly technical practice that traditionally requires programming skills. This serves as a barrier to entry for would-be developers or those hoping to use games as part of their creative expression. While there have been prior game development tools focused on accessibility, they generally still require programming, or have major limitations in terms of the kinds of games they can make. In this paper we introduce Mechanic Maker, a tool for creating a wide-range of game mechanics without programming. It instead relies on a backend symbolic learning system to synthesize game mechanics from examples. We conducted a user study to evaluate the benefits of the tool for participants with a variety of programming and game development experience. Our results demonstrated that participants' ability to use the tool was unrelated to programming ability. We conclude that tools like ours could help democratize game development, making the practice accessible regardless of programming skills.

Introduction

Developing a video game requires significant time and skill, with the complexity of games increasing over time (Petrillo et al. 2009). This limits who can engage with games as an artistic medium, as a tool for procedural rhetoric (Treanor and Mateas 2013), or as an education aide (Freitas 2018). If the time and technical barrier of entry into game development could be reduced, game development would be more approachable, allowing access to game creation for those who have been historically excluded. However, this requires more accessible game development tools particularly focused on reducing the requirements of programming skills. Specifically, we anticipate that a tool for creating game mechanics, as a crucial component to video games, would be a useful first step towards this broader goal.

To provide clarity of our meaning when we discuss a game mechanic (often shortened to mechanic), we use the definition from Zubek (Zubek 2020). Zubek defines a game mechanic as "the basic activities of a game and the rules that govern them". When we discuss rules, we refer to rules as they are represented by code in a game. Rules are implemented in a specific programming language and are executable by a game engine. A game engine is a software environment that can be used to develop a video game. When we discuss a game engine throughout this paper we are narrowing the focus to a group of rules provided from the backend that execute to create a playable game mechanic. The game engine provides usable rules to execute the game mechanic the user is creating.

There are various tools that can help to create mechanics without the typical programming requirement, but they have limitations. Tools for game mechanic creation without typical programming take one of two forms: (i) visual programming, or (ii) pre-authored components with user-defined parameters. Visual programming allows users to create mechanics through visual elements instead of writing code in text. Visual programming tools include Scratch (Resnick et al. 2009) and Unreal Engine Blueprints (Games 2004). While helpful, visual programming languages do not fully remove the technical barrier as they still require programming knowledge, just with a different interface. Pre-authored components have proven popular in commercial game development tools like Kodu Game Lab (Stolee and Fristoe 2011) and Dreams (Molecule 2020). Recombining pre-authored components to produce new mechanics removes the need for programming, but pre-authored components are limiting in the types of mechanics that the user can create.

In this paper, we present Mechanic Maker, a game mechanic creation tool which requires no programming knowledge. Instead, it works through a backend symbolic Machine Learning (ML) approach that creates code (i.e. program synthesis), based on a user demonstrating mechanics they want to exist. In this way, a user can define a variety of mechanics. For example, a user could create a character moving through keyboard input, an object spawning when another object gets to a certain position, a character jumping when space bar on the keyboard is pressed, and so on. We do not have pre-authored rule templates or otherwise limit users, and our backend symbolic learning approach to program synthesis (SLPS) can learn any deterministic sequence of rules from Markovian states (Guzdial, Li, and Riedl 2017). In effect, this example-based mechanic development paradigm allows users to describe the end goal they want for their mechanics, with Mechanic Maker doing the "programming" for them.

To evaluate Mechanic Maker, we conducted a human sub-

*These authors contributed equally.

ject study with users with a range of prior programming experience. Our results suggest that users find Mechanic Maker valuable for creating game mechanics, and that it is equally useful for programmers and non-programmers. This demonstrates the potential for expanding this methodology through intelligent tools to democratize game creation. Our contributions include Mechanic Maker itself, the underlying SLPS approach extended from (Guzdial, Li, and Riedl 2017), and the results of our human subject study.

Related Work

Mechanic Maker builds on a few different areas of games research including co-creative tools, autonomous rule generation and program synthesis.

Co-Creative Tools

Human-computer co-creativity involves a human and computer working together in the creative process (Davis 2021). Specifically related to game development there are many co-creative tools for developing levels for games (Zhou and Guzdial 2021; Bhaumik, Khalifa, and Togelius 2021). Most approaches to co-creativity use search-based Procedural Content Generation (PCG) or another non-learning PCG approach (Partlan et al. 2021). Previous works have investigated machine learning approaches with explainable AI (Zhu et al. 2018). However, such approaches tend to still assume familiarity with machine learning, which we avoid in an effort to reduce technical barriers. Machado et al. (Machado et al. 2019) have a co-creative tool for game creation that uses an AI-driven game development assistant to suggest existing mechanics from other games based on what the user has developed so far. This tool does not learn and adjust based on input and uses existing rules instead of creating new ones like Mechanic Maker.

There are two prior examples of co-creative tools that learn and adjust their actions based on user feedback (Guzdial et al. 2019; Halina and Guzdial 2022). Guzdial et al. (Guzdial et al. 2019) developed a tool called Morai Maker which takes turns with users to create levels and attempts to learn the style of the human user. Halina and Guzdial (Halina and Guzdial 2022) developed a rhythm game generator called KiaiTime that similarly attempts to learn the design style of a human user. Our proposed tool, Mechanic Maker, does not involve a turn-taking interaction like these two tools, instead continually updating learned rules based on user demonstrations. Mechanic Maker also focuses on game rules instead of game levels.

Kruse et al. (Kruse, Connor, and Marks 2022) presented a human-in-the-loop game design study to determine the use of PCG tools for professional game designers. Their results showed that there is still work to be done to get game developers interested in AI assisted tools. This is due to the technical difficulty in using existing PCG tools in games and the unreliable results that the tools can provide. We aim for our tool to address the problem of technical difficulty as it removes the need for coding or parameter tuning.

Guzdial et al. (Guzdial, Li, and Riedl 2017) employed their Engine Learning algorithm to learn the rules from three existing games from gameplay video and then attempted to combine rules from these existing games to generate new rules. While we based our SLPS approach on their Engine Learning algorithm, which we discuss further below, we extend this algorithm in order to make it appropriate for real time interaction.

Autonomous Rule Generation

Autonomous generation stands at the other side of the spectrum from co-creative approaches, in which a system generates content without any human interaction outside of starting the process (Deterding et al. 2017). The majority of existing rule generation work has relied upon autonomous generation rather than anything with a human in the loop. We can split autonomous rule generation work into two groups, (i) search-based PCG and (ii) world models.

In search-based PCG, a generator searches over a space of possible rules to produce ones that are good according to a human-authored evaluation function. Search-based rule generation dates back to the early 1990s (Pell 1992). The most common approach requires authoring possible rules effects, which can then be selected based on a search-based optimization (Togelius and Schmidhuber 2008; Sorochan and Guzdial 2022). Cook et al. and the recent follow up by Gonzalez et al. (Cook et al. 2013; Gonzalez, Cooper, and Guzdial 2023) employed code reflection to identify possible public variables which could then appear in different rule effects. Their work, similarly to our own, did not rely on pre-authored rule effects, though we instead learn them from user examples.

With world models, a machine learning model is fed with examples of rules from a real game, and then attempts to approximate them either explicitly or implicitly (Ha and Schmidhuber 2018; Kim et al. 2020; Bruce et al. 2024). They train on a massive amount of data to approximate a particular game environment. Unlike our approach, world models do not generally learn explicit code, instead relying on fuzzy neural predictive models. Guzdial and Riedl (Guzdial and Riedl 2021) represents the only example, to the best of our knowledge, of combining world models and autonomous rule generation.

Program Synthesis

Program synthesis represents the task of automatically generating programs to accomplish some task (Gulwani, Polozov, and Singh 2017). It is not typically applied to game development, though some prior work exists within the domain of games (Kreminski and Mateas 2021). Similar to Mechanic Maker, Medeiros et al. (Medeiros, Aleixo, and Lelis 2022) synthesize programs that represent strategies based on player behaviors. But they do not apply program synthesis to generate game content.

Yang et al. (Yang et al. 2021) apply program synthesis for approximating unseen parts of partially observed environments. While Mittelmann et al. (Mittelmann et al. 2022) design game theoretic environments based on optimal strategies using program synthesis. Both of these prior approaches apply program synthesis to produce dynamic information about game-like environments. In comparison, our program

synthesis approach relies on human input and feedback to design playable video games.

Simplified game engines can be used to reduce the technical barrier of game development (Chover et al. 2020). The Gemini game generator (Summerville et al. 2018) creates games using a predefined set of rules. Gemini synthesizes these rules based on a provided meaning by a user in a domain-specific language (DSL). While this tool doesn't require directly programming a game, it does require specialized technical knowledge in terms of authoring intended meanings in its DSL.

Mechanic Maker

Mechanic Maker is a game development tool initially developed by Saini and Guzdial (Saini and Guzdial 2020). The Mechanic Maker tool is split into two different sections:

1. The Symbolic Learning Program Synthesis (SLPS) backend. This backend interfaces with the game mechanic editor to learn from the user inputs.
2. The game mechanic editor frontend. This frontend allows users to create game mechanics and receive predictions from the SLPS backend.

Symbolic Learning Program Synthesis

The SLPS backend extends the Engine Learning algorithm by Guzdial et al. (Guzdial, Li, and Riedl 2017). We made the following changes to the Engine Learning algorithm from Guzdial et al. First, the original algorithm assumes an unchanging sequence of game frames from a gameplay video. We instead changed the algorithm to run iteratively every time there was a new frame, initializing with the last learned engine instead of an empty engine. Second, we altered the types of Facts, breaking their "Spatial" Fact type into a PositionX and PositionY Fact type, and removing the CameraX Fact type. We did this to learn more nuanced rules for the former and as there is no camera movement for the latter. An example of the format of one of the learned rules can be found in the appendix of the arXiv version of this paper (Sumner, Saini, and Guzdial 2024). Third, we also set the Engine Learning loop to a maximum number of ten iterations based on initial testing, rather than requiring it to run until it has a perfect prediction. We needed this as the Engine Learning algorithm assumes no hidden information or randomness, and it could fail if the user introduced either. In such cases, we return the closest learned Engine after termination. With these changes, our SLPS backend can take in new frame content and output an engine as a sequence of rules at runtime that works with the frontend game editor to represent a game's mechanics.

We show our SLPS algorithm in Algorithm 1. The input is a sequence of frames. Each frame is defined as the set of inputs the user defined and a set of objects, which are themselves defined by their image, x and y coordinates, and velocity in the x and y direction. SLPS translates each frame into relevant fact types: animation facts (that track image usage), velocity x facts, velocity y facts, position x facts, position y facts, variable facts (tracking input information, each fact tracking one button being pressed or not), relationship x facts (tracking x-dimension relationships between objects), relationship y facts, and empty facts (which are necessary to handle appearing and disappearing rules).

For each frame in this fact representation, SLPS attempts to predict the next frame with a given current engine (defined as a sequence of learned rules). This is executed in Algorithm 1 inside the *Distance()* function. Initially this engine is empty. We populate the engine with rules using a search-based process, shown in Algorithm 1 as *EngineSearch()*. This function optimizes the engine by adding, modifying, and removing rules.

Adding a rule is a search operator that takes a pair of unmatched facts between the frames and creates a rule. This rule is created by setting all the facts of the current frame as the conditions that must be true for the rule to fire. It also adds the unmatched facts in the form of the pre-effect fact and post-effect fact. Modifying rules is a search operator that simplifies an existing rule by taking the union of the facts in a rule's condition and in the current frame as the modified rule's conditions. This allows for rules to generalize over time. Removing rules is a search operator that deletes rules. In this way, the engine is optimized by using the Hellman's metric to minimize the number of differences between the frames in terms of facts.

If we update the engine being used, shown in Algorithm 1 as *UpdatedSuccessfully()*, we restart the predict process from the first frame with the new engine now being used. This is done until a prediction within the threshold is reached, shown in Algorithm 1 as the *Predict()* function, or we have searched through 10 neighbouring engines, at which point we return the engine that minimized the Hellman's metric.

Game Mechanic Editor

There are three main components to our game mechanic editor frontend:

1. The frame editor shown in Figure 1.a.
2. Predictions from the SLPS backend, with an example shown in Figure 1.b.
3. The Play Mode shown in Figure 1.c.

The frame editor in Figure 1.a is the main point of interaction in our tool. We present the controls of our game mechanic editor at the bottom of each sub-figure in Figure 1. Here the user can specify the frame to modify and what inputs the player gives to demonstrate the desired mechanics. The user specifies the intended behaviours for their game mechanics by demonstrating them across these frames. For example, if a user wants to have an object moving to the right, they would add it to the grid for frame 0 then place the same object one position to the right in the next frame. The SLPS backend, discussed above, will then learn that the object should move to the right when the game is played. The frames allow the user to specify the desired effects of the game mechanics and represents the training data for the SLPS backend to learn what mechanics the player wants in the game.

As demonstrated in Figure 1.b, when the user moves to a new frame they see a semi-transparent "ghosting" of the

Algorithm 1: SLPS Engine Learning Algorithm

input: A sequence of continuous and valid frames of size f and threshold θ
output: $Engine\ engine$;
while $True$ (Until runtime stopped) **do**
  $e \leftarrow newEngine()$;
  $cF \leftarrow frames[0]$
  $MaxIterations \leftarrow 10$
  while $i \leftarrow 1$ to $len(frames)$ **do**
    Attempt to learn an engine within the given MaxIterations
    while $iterations \leq MaxIterations$ **do**
      Check if this engine predicts within the threshold.
      $frameDist \leftarrow Distance(e, cF, i+1)$;
      if $frameDist \leq \theta$ **then**
        $cF \leftarrow Predict(e, cF, i+1)$
        $break$;
      end if
      Update engine and start parse over;
      $e \leftarrow EngineSearch(e, cF, i+1)$
      if UpdatedSuccessfully(e, cf, i+1) **then**
        Reset frames and start again
        $i \leftarrow 0$;
        $cF \leftarrow frames[0]$;
        $iterations \leftarrow 0$;
        $break$;
      end if
      $iterations \leftarrow iterations + 1$;
    end while
    $i \leftarrow i + 1$;
  end while
end while

SLPS backend's current predictions based on its current learned mechanics. At first, when nothing has been learned, this will simply predict the prior frame (assuming nothing changes). However, as mechanics are learned, the tool will reflect them in its prediction. The user can choose to accept the prediction or ignore it, and edit the frame as they see fit.

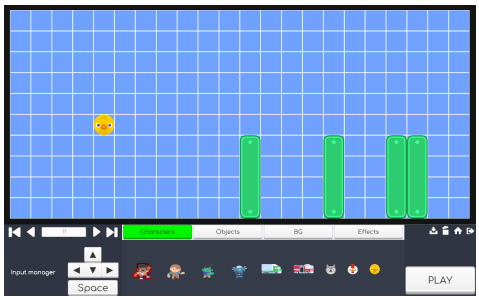
At any time, the player can test the current learned rules in real-time by pressing the play button in Figure 1.c. This allows them to verify that the game is working as intended, as shown in Figure 1.c. If not, they can keep iterating and adding more frames to correct the SLPS backend's learned mechanics.

Human Subject Study

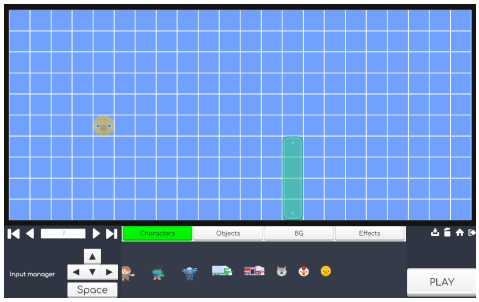
In this section we cover the setup of our human subject study. We ran our human subject study to evaluate our Mechanic Maker tool and investigate the hypotheses listed below. We obtained ethics approval via the University of Alberta Research Ethics Board (REB), Pro00102469.

Hypotheses

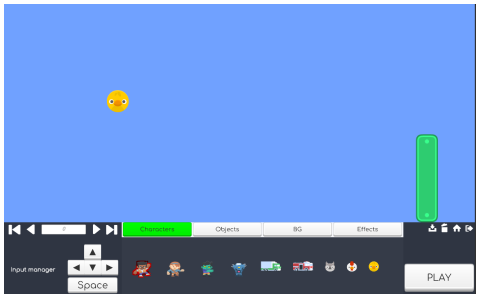
We focused on testing three hypotheses in our user study. The first hypothesis is that programming experience is not required to use our tool effectively. This hypothesis would help us determine if our tool is capable of lowering the technical barrier for creating game mechanics, which is the primary focus of this research. We next wanted to test that Mechanic Maker has value as a game development tool. Testing the value of the tool for game development is more difficult to measure objectively due to a lack of existing measures for game development quality. Thus we split this into two different hypotheses related to game development value and measure them separately. The second hypothesis is the *usability* of this tool. We define usability in this work as the extent to which the tool was able to help each participant create the game mechanics they desired. The third hypothesis is *participant enjoyment* when using the tool. We want to know if the tool was enjoyable for the participants to use, which will help us determine the overall user experience for creating game mechanics. Support for these hypotheses would demonstrate the potential for achieving our goal of democ-



(a) Frame Editor for frame 0



(b) SLPS Prediction



(c) Play Mode

Figure 1: The Mechanic Maker editor. (a) The user defines frames of their game by placing objects on a grid, (b) the SLPS backend attempts to learn the underlying rules suggested by the changes across the frames, and (c) the user can test the learned rules in real-time via the Play Mode.

ratizing game development by continuing to build similar tools or iterating upon Mechanic Maker.

Procedure

Our study consisted of an hour long process broken into four parts. Each part had a time limit. If the user did not complete a part within the time limit, a facilitator asked them to move on to the next part. We did this for consistency, to respect participants' time, and as we found longer periods with the tool did not improve outcomes.

The first part of the study was a step-by-step tutorial to provide users with an understanding of Mechanic Maker. It involved recreating the game mechanics for a game called Sokoban where the objective is for the player to move a crate onto a goal to beat the level. Our tutorial was a simplified version of this that introduced the participant to player movement and pushing the crate to the right. Participants were given a reference video of one of the study personnel walking through how to create the Sokoban mechanics with our tool, which we iterated on based on a pilot study of four individuals. We allocated 25 minutes to this portion of the study as the participants were still learning how the tool worked and how to interact with the UI.

The second part of the study involved asking the participants to replicate a simplified version of the game mechanics from Flappy Bird. Flappy Bird consists of pipes moving to the left with the player controlling a bird, attempting to dodge the pipes for as long as possible. We again gave participants access to a reference video. However, in this case, we only gave them video of the final game mechanics running, not of the steps needed to recreate them with Mechanic Maker. The video showed a simplified version of Flappy Bird involving only one pipe that would go to the end of the screen and then teleport to the other side to keep moving to the left, and a bird that would continually fall, and jump upwards when the space bar was pressed. A screenshot of the Flappy Bird example is shown in Figure 1.c. There were no Game Over elements or scoring involved in this process. The participants had 15 minutes for this part of the study. If participants could successfully recreate the simplified Flappy Bird regardless of programming ability that would support our first hypothesis that Mechanic Maker did not rely on programming ability. It would also support our second hypothesis around the usability of the tool.

The third part of the study gave participants the opportunity to create game mechanics of their choice. They were given 15 minutes to come up with game mechanics and implement them in Mechanic Maker. They were asked to take what they had learned from the previous exercises to "create a simple game". If participants were able to successfully create a wide variety of games, that would support our second hypothesis of the value of Mechanic Maker for game development in terms of usability. Further, if they enjoyed this process, this would provide support for the third hypothesis of participant enjoyment.

The final part of our study was a survey that took 5 minutes to complete. We cover the survey in the next subsection. We included the survey in order to collect self-reported and demographic information related to our hypotheses.

Survey Design

The survey was composed of 19 Likert scale questions, three short answer questions, and ten demographic questions. While we only gave participants five minutes for the survey, we found this to be ample time during our pilot studies. The first section of the survey used a four point Likert scale with 1 being not at all true, 2 being not true, 3 being true and 4 being very true. The reason for this was to attempt to minimize the neutrality bias. We adapted the first nineteen questions from the Intrinsic Motivation Inventory (IMI) (R. M. Ryan and Koestner 1983) due to the lack of a validated survey for co-creative tools. These questions were centered around our second and third hypotheses, asking users about their usability and enjoyment of the tool in each part of the study. We acknowledge that there are other surveys more related to usability and that IMI is not used as a complete measure, but adapting our questions from IMI was sufficient for our initial Mechanic Maker study.

The second section included three short answer questions for feedback around what users would change and overall thoughts around the tool. The final section asked demographic questions, including the users programming experience and game development experience, shown in Table 1, which we used to evaluate our first hypothesis. All original survey questions can be found in the appendix of the arXiv version of this paper (Sumner, Saini, and Guzdial 2024).

Results

In our human subject study we collected two types of information. First, from the survey we collected self-reported quantitative, qualitative, and demographic information. Second, we logged all major Mechanic Maker events, all final games, and all of the frames produced by users. In the below subsections we report our results related to our three hypotheses.

Participants

We had 16 participants with varying programming and game development backgrounds take part in our study, as shown in Table 1. We had 8 participants between the ages of 18-25, 7 between the ages of 25-35 and 1 between the ages of 35-45. We had 11 male, 3 female and 2 non-binary or other participants. This is not an equitable distribution, but shows similar gender bias to what is seen in the games industry and the tech industry broadly (Statista 2023).

Frame Error

To measure the success of users at replicating the reference games, we introduce a metric called *Frame Error*. As a reference point, we drew on the frames that were used to create the example games in the tutorial videos for both Sokoban and Flappy Bird. These examples frames were authored by study personnel. For each participant, we measured how well the learned mechanics from that participant could predict each example frame given the prior example frame. We defined *Frame Error* using the Hellman's Metric to determine how close a predicted frame was to the true next frame. This metric is useful for determining success at replicating

Participant ID	Programming	Game Development
1	Limited	Limited
2	Expert	Moderate
3	Limited	Limited
4	Expert	Expert
5	Expert	Moderate
6	Expert	Limited
7	Moderate	Moderate
8	None	None
9	Moderate	Limited
10	Expert	Moderate
11	Expert	Moderate
12	Expert	Moderate
13	Limited	None
14	Limited	Limited
15	None	None
16	Limited	Limited

Table 1: Experience of participants in programming and game development.

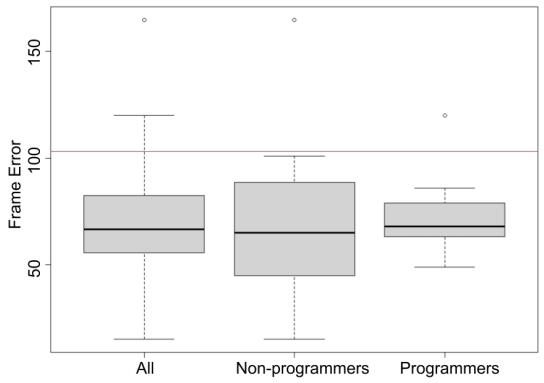
the reference games as it gives us an error equivalent to the dissimilarity of the participant's learned mechanics from the intended mechanics.

Baseline For our baseline, we used the previous example frame with no changes as the prediction, measuring the difference with Hellman's metric as above. This is equivalent to the prediction of an empty engine, and was found to significantly outperform specialized frame prediction models with low training data in prior work (Guzdial, Li, and Riedl 2017).

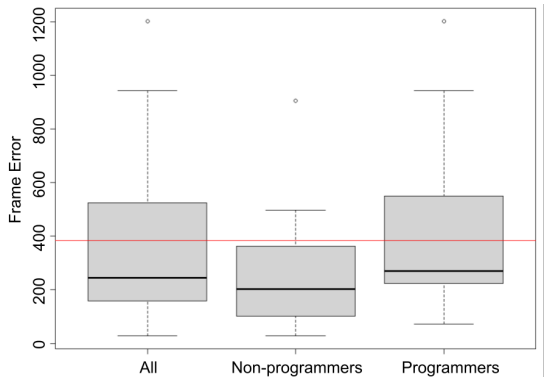
Frame Error Results The results are shown in Figure 2. For both of these plots, we reduced the programming experience groups from our survey participants shown in Table 1 from four groups (None, Limited, Moderate, Expert) to two (Non-programmers and Programmers). This was done to simplify the box plots as we are only looking at whether lower programming experience led to any difference in the results compared to higher levels of programming experience.

As shown in Figure 2.a, for the Sokoban part, programming experience did not have an impact on the frame error at all, and both values were below the baseline value (the red line). In the Flappy Bird example in Figure 2.b, the same is true. The median values of frame errors are lower than the baseline, indicating that the majority of participants led to learning useful rules. The non-programmer distribution is actually moderately lower than the programmer distribution. These results support our first hypothesis.

We also ran a Pearson correlation test for the frame error for both the Sokoban and Flappy Bird parts using the four choice options for programming experience. We found that in both parts there was no correlation between frame error and programming experience. Sokoban had a correlation value of 0.11 and Flappy Bird a correlation value of 0.21, with neither being significant.



(a) Sokoban Activity



(b) Flappy Bird Activity

Figure 2: Frame error grouped by programming experience for the (a) Sokoban and (b) Flappy Bird activities. From the survey results None and Limited programming were merged into the non-programmer category and Moderate and Expert programming experience were merged into programmer. The All box plot shows all programming experience combined. The red line marks the performance of our baseline.

Free Play Analysis

Given that there was no ground truth for the Free Play part of the study, we cannot calculate frame error. Instead, we determine whether participants were able to make a variety of game mechanics, or if participants were limited in terms of making game mechanics like those in Sokoban and Flappy Bird. Ideally we would have some measure of how different the games are from one another to demonstrate that Mechanic Maker can create a variety of games. Since this is difficult to quantify, we use standard deviation metrics as a proxy for the variation. Using this metric we found the standard deviation to be 67.06 ± 48.28 and 10.13 ± 5.74 for the number of frames and number of mechanics created, respectively. Given the large standard deviation values, in the number of frames in particular, this provides some support for there being a large variety of games, which can also be seen in the six selected games shown in Figure 3. This provides some evidence to support our second hypothesis, that Mechanic Maker can create a variety of different game me-

chanics.

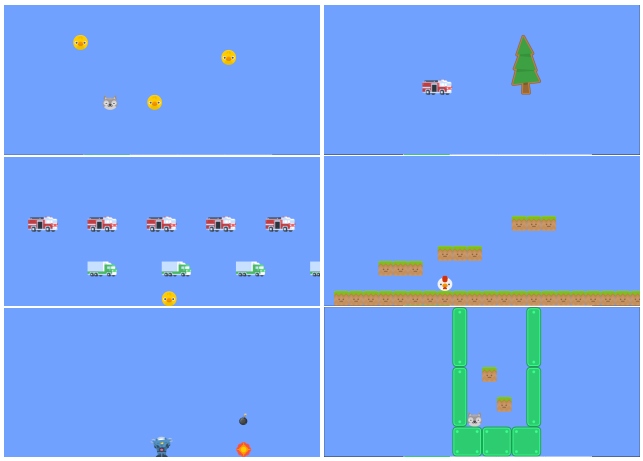


Figure 3: Example game outputs from the Free Play portion of the user study.

Mixture Model

In an attempt to better understand the variability of the free play results we created a Gaussian Mixture Model (GMM) from the user study data. This was inspired by previous attempts to better understand design tasks through clustering (Alvarez, Font, and Togelius 2022). We formatted each rule using a one-hot encoding of the fact type (velocity, position, animation) and the relevant value information of the fact for both the pre-effect and post-effect. We also included a count of how many of each type of condition (by fact type) was used in each rule. From there, we used a GMM to group the different rule representations together into clusters. We used the elbow method to determine the number of optimal Gaussians for our data and found that 7 clusters created a good representation. A visualization of the distribution of Gaussians is shown in Figure 4. Each point represents a learned rule, and each colour represents the Gaussian that each rule largely falls within. With this information we can describe the means of each cluster to get a better understanding of the groupings. In the top right of Figure 4 there are two clusters — clusters 2 and 4. These two clusters are associated with pre-effects and post-effects with velocities in the y direction, while all the other clusters are related to velocities moving in the x direction. The isolated cluster 3 is related to no movement in either the velocity x or y, encompassing rules that include collision information. This demonstrates a wide variety of rules learned during the free play portion, with a bias towards velocity rules.

Survey Results

Figures 5 to 8 show the relevant participant responses from the user study survey questions. We cover the survey results in three main categories.

1. The enjoyment of using the tool — Did players enjoy using the tool?

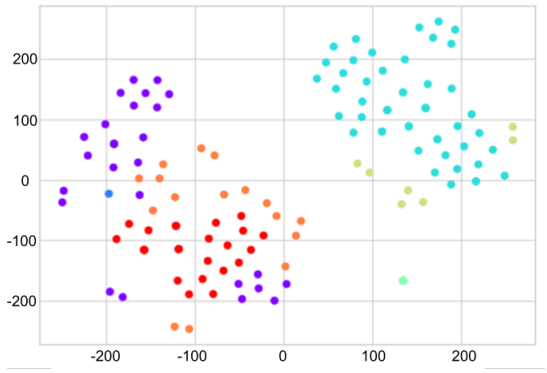


Figure 4: T-distributed Stochastic Neighbor Embedding (TSNE) to visualize the twenty dimensions of the GMM as two dimensions. This Figure displays a visualization of our GMM as seven clusters.

2. The usability of the tool — Were participants able to get what they wanted out of the tool?
3. The value of the tool — Did the participants think this tool provided benefit to them?

The results for enjoyment of the tool were largely positive. In Figure 5 participants agreed that they liked the tool and that the tool was fun to use. In the individual portions of the study, the participants mostly found the tool fun to use.

For usability of the tool, the results in Figures 5, 7 and 8 indicate that there is some issue for the participants achieving the results they wanted. This can be observed in the questions *General - Gave the results I wanted*, *Bird - Replicate video*, and *Free Play - Create what I wanted*. As shown in our frame error analysis, participants were able to use Mechanic Maker effectively to create the appropriate rules when compared to the example games for Sokoban and Flappy Bird. This indicates they are generally achieving the desired effects, but that there is some friction in the interaction with the tool which we hope to address in future work.

The survey questions related to the value of the tool indicate an overall perception that Mechanic Maker in general is beneficial for making games and has value to the participants. It is promising to see that in Figure 8 participants felt that the tool had value. This supports our second hypothesis that Mechanic Maker provides benefit as a game development tool.

Discussion

In our survey we asked users for feedback on the tool to get a sense of what stood out. Most of the feedback was positive and indicated that Mechanic Maker had potential for game development. A couple handpicked quotes from the short-answer portion include “I think this is a unique idea and approach to letting people make games. I feel like there is a great amount of potential for this tool, especially in teaching the basics of game design.” and “...It is also impressive in terms of being a ‘democratizing’ tool i.e. allowing access to game-dev to anyone. I believe that tools like this are going to be a big part of the future of computing; good work!”

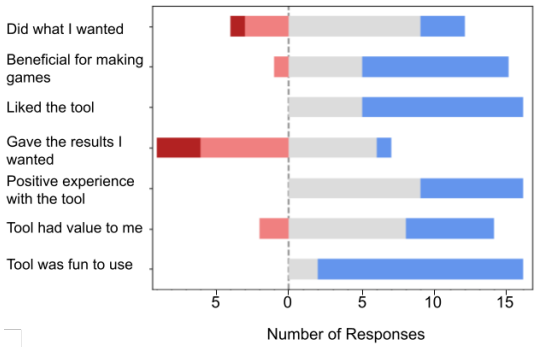


Figure 5: Survey results for the tool in general.

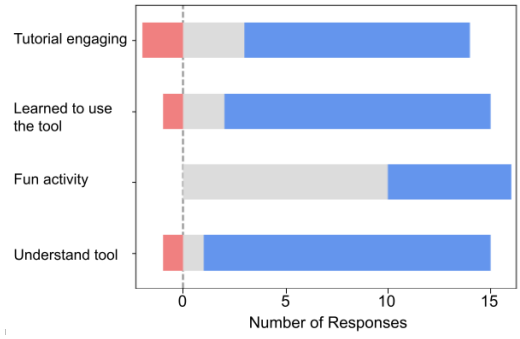


Figure 6: Survey results for the tool when doing the Sokoban activity.

These quotes provide support for future iterations of the tool making game development more accessible.

Ethical Statement

Using AI tools to automate different areas of game development can cause concern in the games industry for a variety of different reasons. Generative AI tools including ChatGPT and Dall-E have recently sparked debate due to copyright issues. These tools are based around models that require massive amounts of training data, which can lead to datasets including works without a creator's consent. Our tool instead only uses data provided by the user. Another major area of ethical concern with AI is the loss of creative jobs including game art, voice overs, writing and other areas of a game (Vimpari et al. 2023). Mechanic Maker does reduce the programming requirement for game development, which could lead to potential job losses in technical positions. However, this tool is focused on newcomers to game development. The emphasis is on reducing the technical barrier to allow more people to create games. Even though the focus is not on industrial applications, there is always a risk for new tools like this to lead to potential job losses. Regardless, since our tool is a collaborative effort between the AI and the user, there will always be the need to have people work with this tool co-creatively.

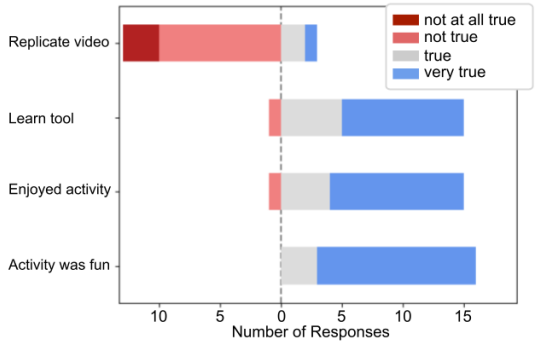


Figure 7: Survey results for the tool when doing the Flappy Bird activity.

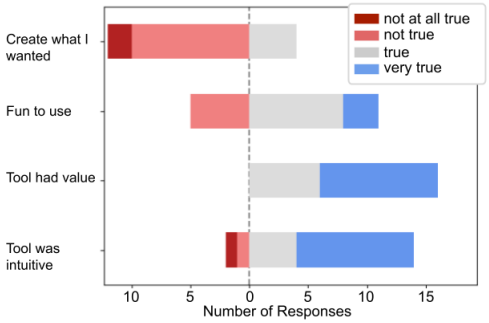


Figure 8: Survey results for the tool when doing the Free Play activity.

Future Work

We identify that Mechanic Maker tended to perform best with fewer frames and objects. In future iterations, we hope to improve the performance of the tool. Currently, we start every game creation session with a blank engine. However, we now have the results from the participants of this study, and so we hope to develop approaches to adapt knowledge from earlier learned engines to help better approximate a current game more quickly. Our current plan to improve the frame prediction is to use the GMM we've created to cluster mechanics. In particular, we anticipate a case-based reasoning or interpolation strategy to approximate new rules by querying the GMM may be effective at speeding up the rule learning process.

Conclusion

Game development is a complicated, technical field that traditionally requires significant programming skills. We propose Mechanic Maker, a new tool for game development that has the potential to eliminate, or at least mitigate, the necessity of programming to create game mechanics. Our user study shows that Mechanic Maker can be used to create intended games equally well between programmers and non-programmers. The study also provided an overall positive response to the tool. We believe Mechanic Maker has the potential to make game development more accessible and to democratize the field.

Acknowledgements

This work was funded by the Canada CIFAR AI Chairs Program, Alberta Machine Intelligence Institute, and the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

Alvarez, A.; Font, J.; and Togelius, J. 2022. Toward designer modeling through design style clustering. *IEEE Transactions on Games*, 14(4): 676–686.

Bhaumik, D.; Khalifa, A.; and Togelius, J. 2021. Lode Encoder: AI-constrained co-creativity. In *2021 IEEE Conference on Games (CoG)*, 01–08.

Bruce, J.; Dennis, M.; Edwards, A.; Parker-Holder, J.; Shi, Y.; Hughes, E.; Lai, M.; Mavalankar, A.; Steigerwald, R.; Apps, C.; Aytar, Y.; Bechtle, S.; Behbahani, F.; Chan, S.; Heess, N.; Gonzalez, L.; Osindero, S.; Ozair, S.; Reed, S.; Zhang, J.; Zolna, K.; Clune, J.; de Freitas, N.; Singh, S.; and Rocktäschel, T. 2024. Genie: Generative Interactive Environments. arXiv:2402.15391.

Chover, M.; Marín, C.; Rebollo, C.; and Remolar, I. 2020. A game engine designed to simplify 2D video game development. *Multimedia Tools and Applications*, 79.

Cook, M.; Colton, S.; Raad, A.; and Gow, J. 2013. Mechanic miner: Reflection-driven game mechanic discovery and level design. In *Applications of Evolutionary Computation: 16th European Conference, EvoApplications 2013, Vienna, Austria, April 3-5, 2013. Proceedings 16*, 284–293. Springer.

Davis, N. 2021. Human-Computer Co-Creativity: Blending Human and Computational Creativity. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 9(6): 9–12.

Deterding, S.; Hook, J.; Fiebrink, R.; Gillies, M.; Gow, J.; Akten, M.; Smith, G.; Liapis, A.; and Compton, K. 2017. Mixed-initiative creative interfaces. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, 628–635.

Freitas, S. 2018. Are games effective learning tools? A review of educational games. *Educational Technology and Society*, 21: 74–84.

Games, E. 2004. Unreal Engine. https://www.unrealengine.com. Accessed: 2024-02-04.

Gonzalez, J. J.; Cooper, S.; and Guzdial, M. 2023. Mechanic Maker 2.0: reinforcement learning for evaluating generated rules. In *Proceedings of the Nineteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE '23. AAAI Press. ISBN 1-57735-883-X.

Gulwani, S.; Polozov, O.; and Singh, R. 2017. Program Synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2): 1–119.

Guzdial, M.; Li, B.; and Riedl, M. O. 2017. Game Engine Learning from Video. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI-17)*, 3707–3713.

Guzdial, M.; Liao, N.; Chen, J.; Chen, S.-Y.; Shah, S.; Shah, V.; Reno, J.; Smith, G.; and Riedl, M. O. 2019. Friend, Collaborator, Student, Manager. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM.

Guzdial, M.; and Riedl, M. O. 2021. Conceptual game expansion. *IEEE Transactions on Games*, 14(1): 93–106.

Ha, D.; and Schmidhuber, J. 2018. World Models.

Halina, E.; and Guzdial, M. 2022. Threshold Designer Adaptation: Improved Adaptation for Designers in Co-creative Systems. arXiv:2205.09269.

Kim, S. W.; Zhou, Y.; Philion, J.; Torralba, A.; and Fidler, S. 2020. Learning to Simulate Dynamic Environments With GameGAN. 1228–1237.

Kreminski, M.; and Mateas, M. 2021. Opportunities for Approachable Game Development via Program Synthesis. In *AIIDE Workshops*.

Kruse, J.; Connor, A. M.; and Marks, S. 2022. Evaluation of a Multi-Agent "Human-in-the-Loop" Game Design System. *ACM Trans. Interact. Intell. Syst.*, 12(3).

Machado, T.; Gopstein, D.; Nov, O.; Wang, A.; Nealen, A.; and Togelius, J. 2019. Evaluation of a Recommender System for Assisting Novice Game Designers. arXiv:1908.04629.

Medeiros, L.; Aleixo, D.; and Lelis, L. 2022. What Can We Learn Even from the Weakest? Learning Sketches for Programmatic Strategies. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36: 7761–7769.

Mittelmann, M.; Maubert, B.; Murano, A.; and Perrussel, L. 2022. Automated Synthesis of Mechanisms. In Raedt, L. D., ed., *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, 426–432. International Joint Conferences on Artificial Intelligence Organization. Main Track.

Molecule, M. 2020. Dreams. https://indreams.me. Accessed: 2024-02-04.

Partlan, N.; Kleinman, E.; Howe, J.; Ahmad, S.; Marsella, S.; and Seif El-Nasr, M. 2021. Design-Driven Requirements for Computationally Co-Creative Game AI Design Tools. In *Proceedings of the 16th International Conference on the Foundations of Digital Games*, FDG '21. New York, NY, USA: Association for Computing Machinery. ISBN 9781450384223.

Pell, B. 1992. Metagame in symmetric chess-like games.

Petrillo, F.; Pimenta, M.; Trindade, F.; and Dietrich, C. 2009. What Went Wrong? A Survey of Problems in Game Development. *Comput. Entertain.*, 7(1).

R. M. Ryan, V. M.; and Koestner, R. 1983. Relation of reward contingency and interpersonal context to intrinsic motivation: A review and test using cognitive evaluation theory. *Journal of Personality and Social Psychology*.

Resnick, M.; Maloney, J.; Monroy-Hernández, A.; Rusk, N.; Eastmond, E.; Brennan, K.; Millner, A.; Rosenbaum, E.; Silver, J.; Silverman, B.; and Kafai, Y. 2009. Scratch: Programming for All. *Commun. ACM*, 52(11): 60–67.

Saini, V.; and Guzdial, M. 2020. A Demonstration of Mechanic Maker: An AI for Mechanics Co-Creation. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 16(1): 325–327.

Sorochan, K.; and Guzdial, M. 2022. Generating Real-Time Strategy Game Units Using Search-Based Procedural Content Generation and Monte Carlo Tree Search. *arXiv preprint arXiv:2212.03387*.

Statista. 2023. Distribution of game developers worldwide from 2014 to 2021, by gender. https://www.statista.com/statistics/453634/game-developer-gender-distribution-worldwide. Accessed: 2023-08-28.

Stolee, K. T.; and Fristoe, T. 2011. Expressing computer science concepts through Kodu game lab. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, 99–104.

Summerville, A.; Martens, C.; Samuel, B.; Osborn, J.; Wardrip-Fruin, N.; and Mateas, M. 2018. Gemini: Bidirectional Generation and Analysis of Games via ASP. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 14(1): 123–129.

Sumner, M.; Saini, V.; and Guzdial, M. 2024. Mechanic Maker: Accessible Game Development Via Symbolic Learning Program Synthesis. arXiv:2410.01096.

Togelius, J.; and Schmidhuber, J. 2008. An experiment in automatic game design. In *2008 IEEE Symposium On Computational Intelligence and Games*, 111–118. IEEE.

Treanor, M.; and Mateas, M. 2013. An Account of Proceduralist Meaning. In *DiGRA Conference*.

Vimpari, V.; Kultima, A.; Hämäläinen, P.; and Guckelsberger, C. 2023. "An Adapt-or-Die Type of Situation": Perception, Adoption, and Use of Text-to-Image-Generation AI by Game Industry Professionals. *Proceedings of the ACM on Human-Computer Interaction*, 7(CHI PLAY): 131–164.

Yang, Y.; Inala, J. P.; Bastani, O.; Pu, Y.; Solar-Lezama, A.; and Rinard, M. 2021. Program Synthesis Guided Reinforcement Learning for Partially Observed Environments. In Ranzato, M.; Beygelzimer, A.; Dauphin, Y.; Liang, P.; and Vaughan, J. W., eds., *Advances in Neural Information Processing Systems*, volume 34, 29669–29683. Curran Associates, Inc.

Zhou, Z.; and Guzdial, M. 2021. Toward Co-Creative Dungeon Generation via Transfer Learning. In *Proceedings of the 16th International Conference on the Foundations of Digital Games*, FDG '21. New York, NY, USA: Association for Computing Machinery. ISBN 9781450384223.

Zhu, J.; Liapis, A.; Risi, S.; Bidarra, R.; and Youngblood, G. M. 2018. Explainable AI for Designers: A Human-Centered Perspective on Mixed-Initiative Co-Creation. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8. IEEE Press.

Zubek, R. 2020. *Elements of game design*. The MIT Press.