Accessible Game Development Via Symbolic Learning Program Synthesis

by

Megan Johanna Sumner

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Video 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 knowledge, or have major limitations in terms of the kinds of games they can make.

In this thesis we introduce Mechanic Maker, a tool for creating a wide-range of game mechanics without programming. Game mechanics are defined as the basic activities in the game and the rules that govern them, for example, a character moving on screen due to input given on the keyboard. Instead of requiring programming to create game mechanics, Mechanic Maker 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 suggested that participants' ability to use the tool was unrelated to their level of programming ability.

We also explore how we can leverage data from the user study to create a database of learned rules. This database allowed us to create a Gaussian Mixture Model, which we use in a case-based reasoning system to attempt to speed up mechanic creation.

We conclude that tools like ours could help democratize game development by making the practice accessible regardless of programming skills. Addition-

ally, the usability assessment of Mechanic Maker showed that users could effectively use the tool. Finally, we determine that the case-based reasoning approach has the potential to speed up mechanic creation, but further work needs to be done to reduce user friction in Mechanic Maker.

Preface

This thesis is original work done by Megan Sumner. It was completed under the supervision of Dr. Matthew Guzdial at the University of Alberta.

The user study in Chapter 5 of the thesis received research ethics approval from the University of Alberta Research Ethics Board, Pro00102469.

Chapters 4 and 5 of this thesis will be published with Vardan Saini and Dr. Matthew Guzdial as co-authors. The work has not yet been published, but will be part of the Artificial Intelligence and Interactive Digital Entertainment conference (AIIDE-24) in November 2024. V. Saini contributed to the creation of the tool used in this research and M. Guzdial was the supervisory author and contributed to the tool creation and writing and editing the published work.

To everyone who believed in me and showed me how to believe in myself. Thank you to my family, my friends and especially my partner, Nick, for supporting me through everything.

Imagination will often carry us to worlds that never were, but without it we go nowhere.

– Carl Sagan

Acknowledgements

Thank you to my supervisor, Dr. Matthew Guzdial, for guiding me through my Master's degree and being supportive throughout the whole process. He taught me so much about research and reinvigorated my passion for computer science.

I would also like to extend a thank you to Dr. Carrie Demmans Epp and Dr. Nathan Sturtevant for being committee members to my thesis work. Their feedback improved the research in this thesis immensely.

Lastly, a large thank you to my parents, my brother and his wife, my partner and all my friends for supporting me through everything. They helped get me through the toughest times and kept my spirits up.

Contents

List of Tables

List of Figures

Glossary

A*

A search algorithm that can guarantee a shortest path between a start and a goal state.

Autonomous Generation

A system generates content with little or no human interaction outside of starting the process and reviewing the outputted content.

Case-based Reasoning

An Artificial Intelligence framework that tries to solve newly inputted cases by looking at similar past cases. For every new case that is inputted, the system will look in its database of current cases. It will find the case that is most similar and either reuse it outright, or modify it, test it is successful at solving the problem and then add it back to the database.

Clustering

An unsupervised learning technique in Machine Learning. Unsupervised learning involves analyzing unlabeled data and focuses on pattern recognition without human guidance. Clustering is a specific application of unsupervised learning that can be used to visually divide data into related groups.

Co-creativity

A user works with a computational system to iterate on a task. It requires both the user and the computer to make contributions towards a task together.

Game Engine

An environment that is used to develop a video game. It generally includes software to create all parts of a video game. When we discuss a game engine throughout this paper we are narrowing the focus to mechanic creation through a visual editor

Game Mechanic

The basic activities of a game and the rules that govern them.

Gameplay System

A collection of content that creates a fully playable portion of a game.

Gaussian Mixture Model (GMM)

A probabilistic clustering model that generates clusters assuming that the data is generated from multiple Gaussian distributions. Unlike other clustering methods, it generates the clusters over probability fields, instead of on the data itself.

Machine Learning (ML)

A type of algorithm that allows the system to learn and adapt to the problem through training related to the application.

Mechanic Maker

The game mechanic generator tool being worked on in this research. It allows users to place objects frame-by-frame through a user interface and the backend SLPS system predicts the user's intended mechanics in terms of a domain specific language. They can then test out the game to see if the backend predicted correctly and iterate on the results.

Procedural Content Generation via Machine Learning (PCGML)

The algorithmic process used to generate content procedurally using Machine Learning

Procedural Content Generation (PCG)

A group of methods that allow users to create content through computer algorithms without having to define the content outright.

Program Synthesis

The field of computer science related to generating domain specific code. It allows a user to specify an outcome or goal for the system and the program synthesis algorithm will generate code that works within the constraints of the domain.

Rule

An activity represented by code in a game.

Search-based PCG

A user defines a space of possible content to search through for a given domain. A search algorithm will run on this search space to find the best configurations of the desired content. It will execute the search procedure through these possibilities and use a human-authored evaluation function, called a fitness function, to grade the generated content.

Symbolic Learning Program Synthesis (SLPS)

The program synthesis algorithm that runs the backend engine of Mechanic Maker.

User Interface (UI)

The part of the tool that the user interacts with.

Chapter 1

Introduction

Video game development involves a variety of disciplines to create a game due to its multifaceted nature [13]. It requires many different skill sets, over many years to come together to create a game [34]. These skills range from animation, art, audio, design, production, programming, quality assurance, writing and more. We anticipate that all these different facets of game development can appeal to different audiences. This could potentially, in turn, account for the popularity of game development among non-game developers, by providing multiple avenues of artistic expression and creativity while creating a game.

Despite significant interest, it is difficult for novices to make games. Games can take multiple years to create and much of this time is sunk into technical challenges that exist in each section of game development [34]. This limits who can engage with games as an artistic medium, as a tool for procedural rhetoric [46], or as an education aide [12]. 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 [51]. Zubek defines

a Game Mechanic as "the basic activities of a game and the rules that govern them". We also seek to define "Rule" in this quote. 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 thesis we are narrowing the focus to mechanic creation through the User Interface (UI) of our tool.

There are existing 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 [36] and Unreal Engine Blueprints [10]. 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 [41] and Dreams [30]. 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. For example, a tool might offer users the ability to add a "ground" object to their games, with built-in collision rules so others objects do not move through it. However, if a user wanted to make a game about burrowing into the ground, this may not be supported.

In this thesis, we present Mechanic Maker, a game mechanic creation tool which requires no programming knowledge. Instead, it works through a back-end symbolic Machine Learning (ML) approach that creates code (as discussed in Section 2.4 of Chapter 2 on Program Synthesis), based on a user visually 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 a player presses space bar

on the keyboard, and so on. We do not have pre-authored rule templates or otherwise limit users, and our backend Symbolic Learning Program Synthesis (SLPS) approach can learn any deterministic sequence of rules from Markovian states [16]. In effect, this example-based mechanic development paradigm allows users to describe the end design goal they want for their mechanics, with Mechanic Maker doing the "programming" for them.

The goal of this thesis is to investigate three research questions:

R1 Can we build a co-creative tool (Mechanic Maker) that can help a user create a variety of game mechanics?

R2 When tasked with recreating a given set of mechanics with Mechanic Maker, will programming ability significantly impact the result?

R3 Can we use a Gaussian Mixture Model (GMM) to improve the backend SLPS engine results to create generalized game rules more quickly?

The answers to our research questions are discussed throughout the remainder of this thesis. To evaluate Mechanic Maker, and answer [R1] and [R2], we conducted a human subject 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. We also conducted an experiment with a GMM to answer [R3] using the dataset from our user study as a starting point.

Chapter 2

Background

In this chapter we cover topics necessary for understanding the work in this thesis. It is important to first understand what a Game Mechanic is as this thesis covers work related to the creation of mechanics. Procedural Content Generation (PCG) is discussed due to it being the field researching how game content can be created automatically. We cover the topic of Program Synthesis because it provides an understanding for the backend of Mechanic Maker. Clustering is discussed as we conducted a clustering experiment on our data from the user study. Lastly, we describe Case-based Reasoning as it is an alternative approach we propose for Mechanic Maker's backend.

2.1 Game Terminology

There are a few terms common in game development that are useful to our work. A Game Mechanic as defined by [26] is "the basic activities of a game and the rules that govern them". We use this definition going forward in this thesis. When we discuss a Rule, we are describing activities as they are represented by code in a game. Game mechanics and the rules that produce them are what makes a game playable. Game mechanics start to build upon each other to make full Gameplay Systems, but they can start quite granular. Game mechanics include player movement, collision between a player and an object, interaction with an object, scores increasing, the health of the player changing and much more. A gameplay system as defined by [26] is a collection of content that creates a fully playable portion of a game. A visualization of

the hierarchy is shown in Figure 2.1.

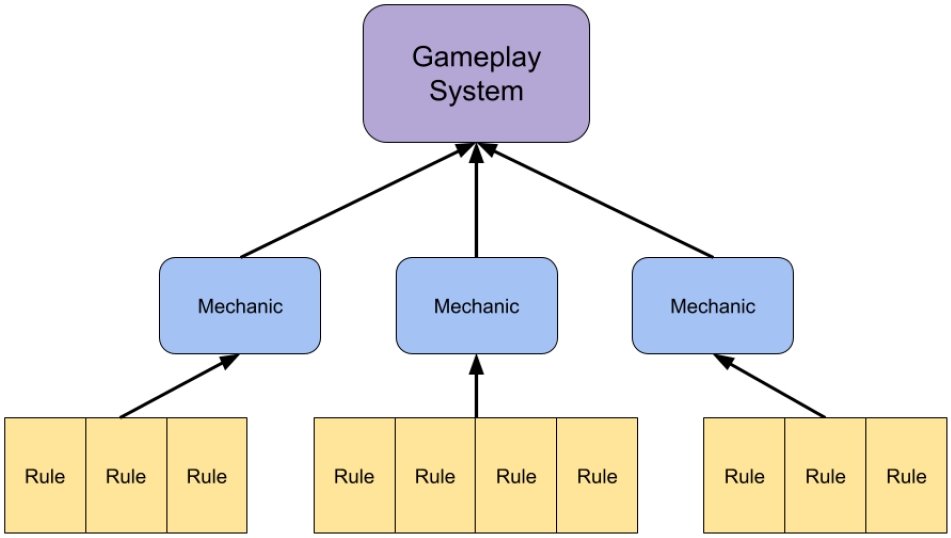


Figure 2.1: A rule is the basic code that affords game actions. Multiple rules form a mechanic and multiple mechanics form a gameplay system.

As an example, a user pressing the right arrow on the keyboard causing an object to move to the right on screen, would be a rule. If more of these rules are added to the game, for example, for each direction on a keyboard allowing an object to move in all directions on the screen, this would represent the game mechanic of player movement. Once we have player movement, player-object interactions can be added and eventually all these mechanics working together create whole game systems, and eventually a playable game. Making game mechanics in most Game Engines requires significant programming effort for a developer to get the mechanics to work as desired. Using the above example, a programmer would have to read right keyboard input from the game engine and set the velocity of a game object to move a specific speed in the right direction on screen. When the arrow key is lifted, they would have to set the right velocity back to zero. This would then need to be done for each direction for just the player movement mechanic. Mechanic Maker aims to remove the necessity of programming when creating these mechanics.

2.2 Procedural Content Generation

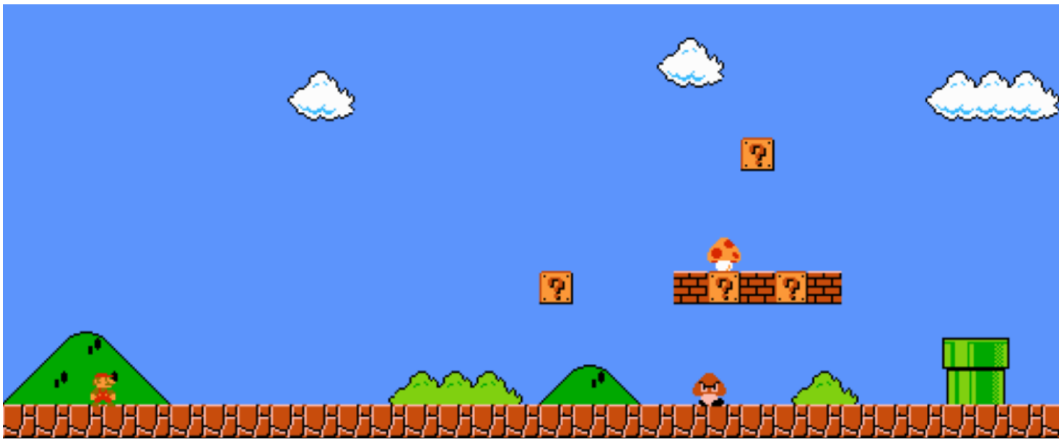
Procedural Content Generation (PCG) is the field of research focused on automated generation. Breaking down the definition into its components, *procedural* means that something is being done using an algorithm. *Content* in video games refers to anything that is created for the video game. Examples of content in a video game include sound effects, art assets, game mechanics, and levels. *Generation* is the process of creating something. Putting these words together we get PCG, which is a group of methods that allows developers to create content through computer algorithms without having to define the content outright. For instance, if we were generating trees for a game, we would want each tree to be visually different from each other. A tool called SpeedTree [43] allows developers to use PCG to generate new tree automatically. There are many tools that generate a variety of different types of content procedurally. In this thesis, we focus on a specific application of PCG to generate game mechanics.

2.2.1 PCG Via Machine Learning

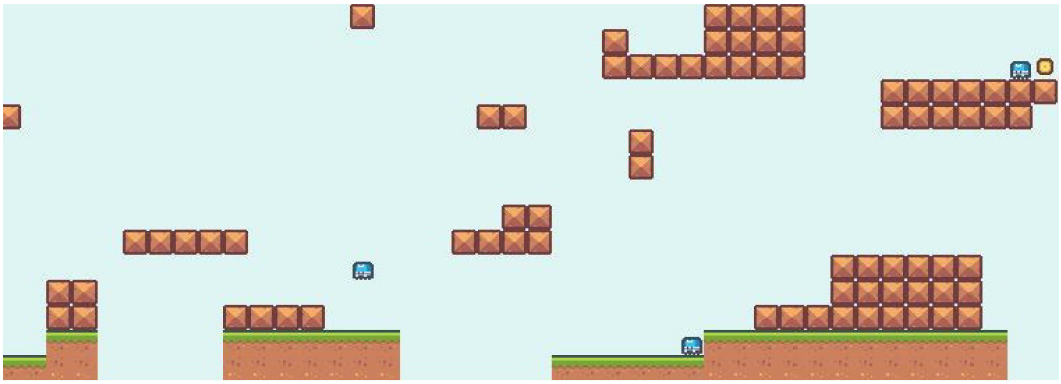
A subset of PCG, known as Procedural Content Generation via Machine Learning (PCGML), generates game content using machine learning techniques. ML refers to a group of algorithms that allows a system to learn and adapt to a problem by training on data or through experience. We refer to ML applied to PCG as PCGML. It provides different methods for game content generation using a variety of ML techniques [19].

An example of how computer science methods can be used within PCGML is with Markov chains. Markov chains predict the next state based on the probabilities of actions from the current state [38]. Using Markov chains, new video game levels can be generated, as shown in Figure 2.2 for the game Super Mario Bros. The game has many levels that serve as training data, and new levels are generated that resemble the original content. Subfigure 2.2.a shows a level from Super Mario Bros. created by the original developers whereas 2.2.b shows an instance of a level generated through a Markov chain trained

on the original levels. The game assets for this example were provided by Kenney[1], which allowed us to visualize the generated Mario-like level. Despite some visual differences, such as the enemies which are brown 'Goombas' in the real level, shown in Figure 2.2.a, compared to the enemies being blue in the generated level, shown in Figure 2.2.b, there are still structural similarities between the example level and the generated level.



(a) Level 1-1 of the Super Mario Bros. game created by the game developers.



(b) A level generated using machine learning, in particular with a Markov chain.

Figure 2.2: A comparison between level 1-1 of Super Mario Bros., shown in subfigure a, and a level generated through PCGML, shown in subfigure b. The generated level used the original Super Mario Bros. levels as training data to create it.

The algorithm used in the backend of Mechanic Maker can be understood as PCGML because it learns from user input. There are a few PCGML frameworks related to Mechanic Maker, which we discuss below.

[1]https://www.kenney.nl/assets

2.2.2 Co-creativity

Co-creativity refers to any PCG approach that involves a user working with a computational system to iterate on a task [25]. This is also known as mixed-initiative PCG. It requires both the user and the computer to make contributions towards a task together. One framework for co-creative PCG is shown in Figure 2.3. A user and an Artificial Intelligence (AI) work in turns to modify an artifact. It is a useful way to use PCG as it allows the user to have more control over the final product. Mechanic Maker works using this method. The AI offers suggestions to the user for where to place different objects on the screen. As shown in the diagram, the user has the final say on the artifacts that the AI generates. The user can either accept or reject the suggestions that the AI creates. This leads to a system that forces the AI to adjust to the user instead of the user adjusting to the AI.

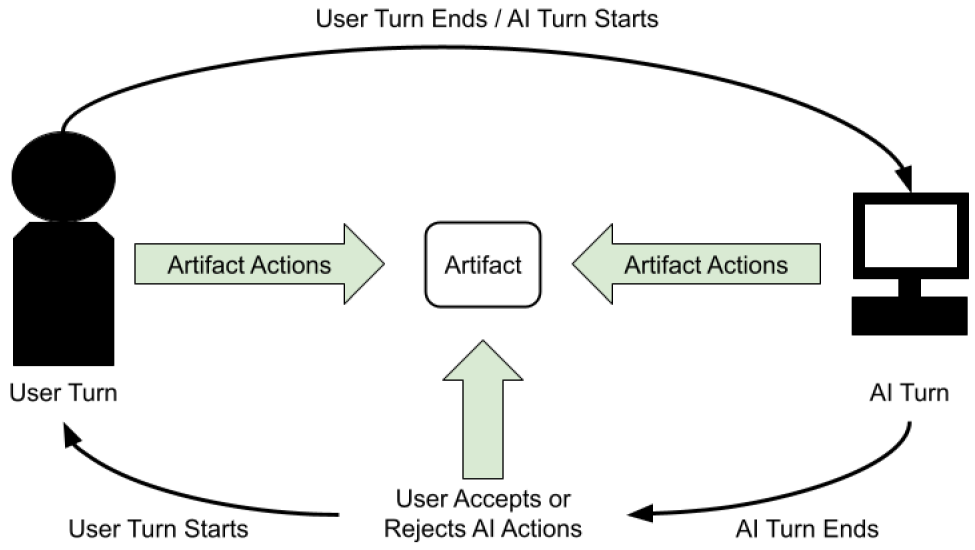


Figure 2.3: Co-creativity between a user and an AI. The two take turns modifying an artifact to achieve a goal.

2.2.3 Autonomous Generation

While co-creativity is a mixture of human and computer interaction, autonomous generation involves the computer generating content with little or

no human interaction outside of starting the process [9]. Figure 2.4 illustrates the autonomous generation process. The autonomous generation system generates the output to the best of its knowledge from the input provided from the user, typically in the form of an evaluation function. There is no back and forth between the user and the PCG system over the course of generation.

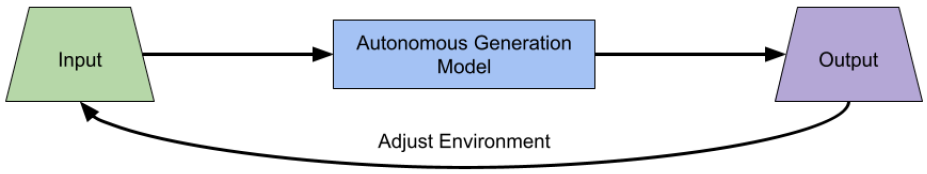


Figure 2.4: A co-creation framework between a user and an AI. The two take turns modifying an artifact to achieve a goal.

2.2.4 Search-based PCG

Search-based PCG is a subset of PCG methods reliant on search algorithms. A user defines a space of possible content to search through for a given domain. A search algorithm will run on this search space to find the best configurations of the desired content [45]. It will execute the search procedure through these possibilities and use a human-authored evaluation function, called a fitness function, to grade the generated content. The fitness function of the search space is used to evaluate the best result for the search and can be authored based on the needs of the domain. The search-based PCG process is illustrated in Figure 2.5.

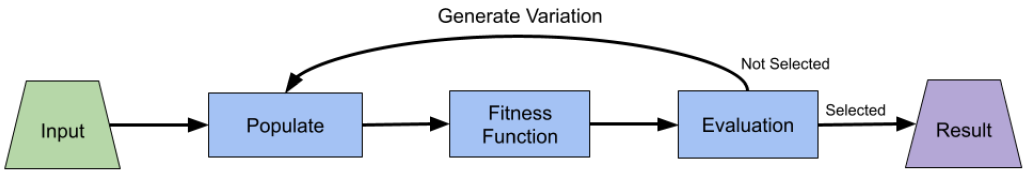


Figure 2.5: A visualization of a search-based PCG approach. This particular search-based PCG process uses a population of content to generate new content.

Search-based rule generation for game mechanics dates back to the early 1990s [33]. The most common approach requires authoring possible rules, effects and facts, which can then be selected using a search-based optimization [39], [44]. Cook et al. [6], [14] in their Mechanic Miner framework employed code reflection to identify possible public variables which could then appear in different rule effects. Cook et al.'s work, similarly to our own, did not rely on pre-authored rule effects, though we instead learn the rules from user examples.

2.3 A*

The backend of Mechanic Maker employs a search-based PCG method to determine the best possible engine for the provided input from the user. It uses an algorithm similar to the A* algorithm to do this. A* is a search algorithm that can guarantee a shortest path between a start and a goal. It is used in video games to efficiently navigate characters throughout a world [7]. For instance, a character in a game could start at a specific location, (0,0), which would be the start state. From this start state, the A* algorithm expands all possible directions that can be moved to as children states of the initial location. If we require the character to move on a grid, these children states would be (0,1), (1,0), (0, -1) and (-1, 0). A* finds the state with the lowest cost and sets that state to the current state and repeats until the goal state is found. The cost is determined by the function shown in Figure 2.6. Using this formula to determine cost and having a heuristic distance function that is admissible and consistent allows A* to find the shortest path from the start state to the goal state [11].

$$f(n) = g(n) + h(n)$$

Figure 2.6: The cost function of A*. g(n) is the cost from the start to the current node. h(n) is the predicted cost of the current node to the goal state.

Mechanic Maker uses an algorithm similar to A* to determine possible

predictions to provide the user. A state in Mechanic Maker is considered an engine, which contains a sequence rules. For our heuristic, we use Hellman's distance. This distance determines how different two states are. For example, if between two states, no objects are moved on the level editor screen, the Hellman's distance would return 0. However, if an object is at position (0,0) and moves to position (0,1) the Hellman's distance would return a higher heuristic cost. There are multiple rules that we could learn to approximate some observed change and we generate all of them. For instance, if we observe an object changing position, the object could have disappeared at (0, 0) and reappeared at (1, 0) or the object could have changed velocity in the x direction from 0 to 1. We use this to determine which set of rules account for the observations most accurately. We discuss this algorithm in depth in the Symbolic Learning Program Synthesis section of the Mechanic Maker chapter of this thesis.

2.4 Program Synthesis

Program Synthesis is the field of computer science related to generating domain-specific code [15]. Domain-specific code, also known as a domain-specific programming language, targets a particular problem and creates code that can only be used to solve that problem. This contrasts general programming languages which can be used to solve many problems across a variety of domains. Program synthesis allows a developer to specify an outcome or goal for the system and program synthesis algorithms will generate code that works within the constraints of the domain. Program Synthesis tends to use a search algorithm to generate the domain-specific code. Mechanic Maker can be understood as a search-based program synthesis to find the best code for the given observation.

2.5 Clustering

Clustering is an unsupervised learning technique in ML. Unsupervised learning involves analyzing unlabeled data and focuses on pattern recognition without human guidance. Clustering is a specific application of unsupervised learning

that can be used to divide data into related groups. It is a useful way to organize data based on similarities without having to have more information on the data. An example of this is shown in Figure 2.7. We can see on the left a selection of unlabeled data. This data can then be ran through a clustering algorithm, in this case K-means clustering, to group the data into k clusters (three clusters in this example).

2.5.1 Gaussian Mixture Models

A GMM is a probabilistic clustering model that generates clusters assuming that the data is generated from multiple Gaussian distributions. Unlike other clustering methods, it generates the clusters over probability fields, instead of on the data itself. This allows for more complex models and datasets, as the data doesn't need to have as much understandable information. Figure 2.8 shows how these clusters can be determined from a probability field. The probability distribution is shown as the dotted line and a GMM will cluster Gaussian distributions to best fit the data.

We used a GMM to analyze the data that we gathered from the user study we ran on Mechanic Maker. We clustered learned rules and found similarities between these rules. This was done as a potential way to visualize rule relationships and determine if we could find commonalities between different user games. Our hypothesis was that many users would end up with similar rules. For example, many mechanics require the player to be able to move the character with keyboard input.

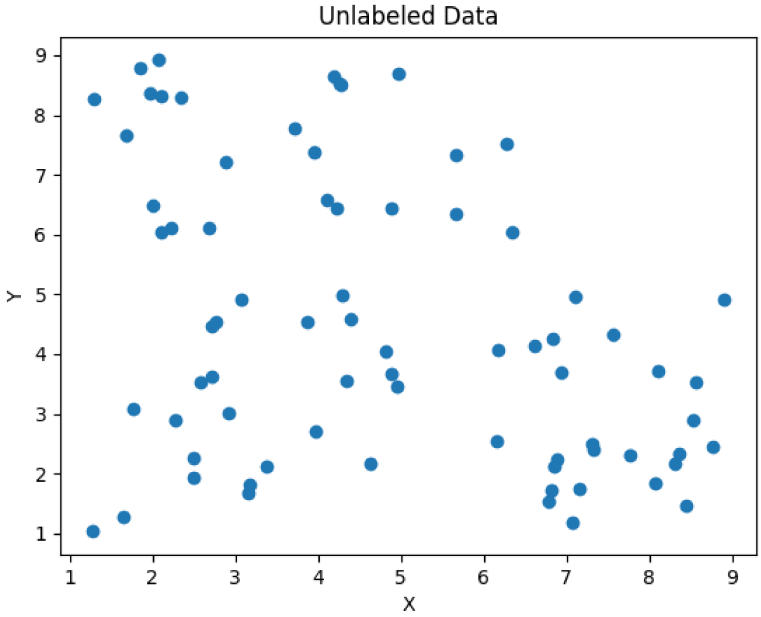
2.6 Case-Based Reasoning

Case-based Reasoning is an AI framework that tries to solve newly inputted cases by looking at similar past cases. For every new case that is inputted, the system will look in its database of current cases. It will find the one that is most similar and either reuse it outright, or modify it, test it is successful at solving the problem and then add it back to the database.

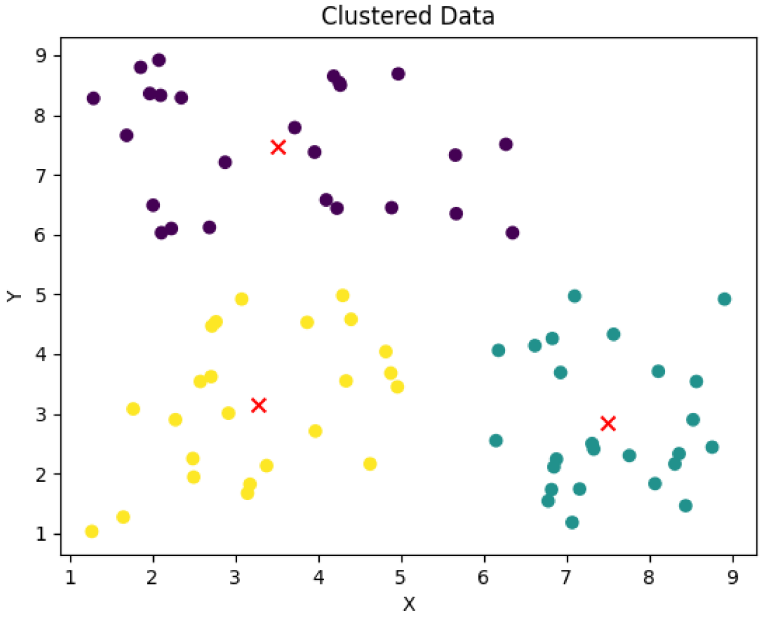
Figure 2.9 shows how the system works. If we input a new case to find a

solution for the given problem, it will try to retrieve a similar case from our database. If it does find a similar example in the database, it will reuse the retrieved case as the output, if not it will modify the retrieved case with information from the inputted case and store this modified case into the database of cases.

This case-based reasoning system is used within our work as an alternative approach to approximating rules. Without this approach, rules have to be learned from scratch each time a user wants to create a new game. Instead, using the case-based reasoning approach, we could pull from a database of rules to potentially learn rules more quickly. This would be done by using the GMM we created as a database to pull from. The idea with our case-based reasoning approach was to input a new case (an approximated rule in our instance) into the GMM. We would then retrieve a cluster that fits the new case best and modify the new case with information from the retrieved cluster. We would then use this modified rule as the outputted solution and store the modified rule in the GMM.



(a) A plot of unlabeled data.



(b) A plot of clustered data.

Figure 2.7: Subfigure a. shows unlabeled data plotted on a graph. Subfigure b. shows the results of applying the k-means clustering algorithm to group this same data into three clusters around a centroid point marked as a red x. This provides us with some understanding of the similarities between the data. This data is artificial and was created for the purpose of visualizing clustering.

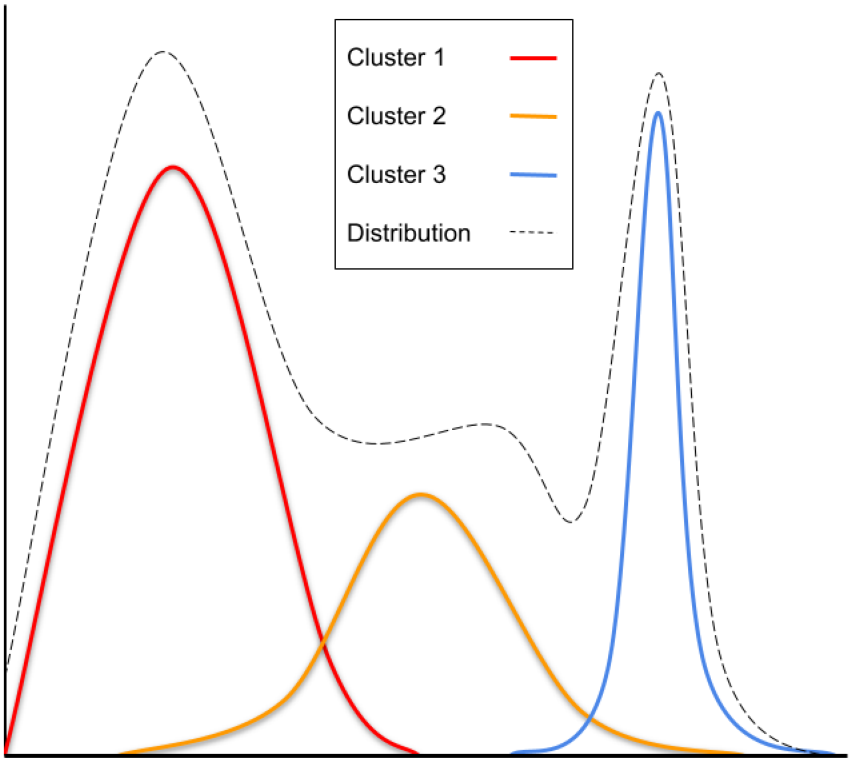


Figure 2.8: Gaussian Mixture Model distribution broken up into clusters.

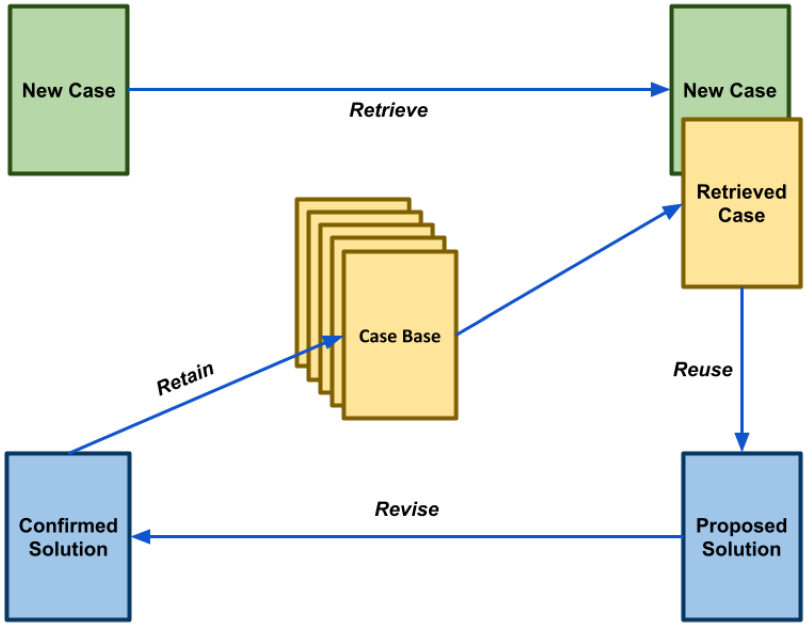


Figure 2.9: A representation of how Case-Based Reasoning works.

Chapter 3

Related Work

Mechanic Maker builds on a few different areas of games research including Co-creativity, Autonomous Generation and Program Synthesis.

3.1 Co-Creative Tools

Human-computer co-creativity involves a human and computer working together in the artistic creative process [8]. Specifically related to game development there are many co-creative tools for developing levels for games [2], [49]. Most approaches to co-creativity use Search-based PCG or another non-learning PCG approach [32]. Previous works have investigated ML approaches with explainable AI [50]. However, such approaches don't create Game Mechanics in the same way as Mechanic Maker. Machado et al. [27] 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 [17], [21]. Guzdial et al. [17] 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 [21] 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. [24] 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. They conclude that 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.

3.2 Autonomous Rule Generation

We can split autonomous rule generation work into two groups, (i) search-based PCG and (ii) world models.

Guzdial et al. [16] employed their Engine Learning algorithm for autonomous rule generation. They learn rules from three existing games from gameplay video and then attempted to combine rules from these existing games to generate new rules [18]. 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.

Simplified game engines can be used to reduce the technical barrier of game development [5]. The Gemini game generator [42] 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.

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 [4], [20], [22]. 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 [18] represents the only example, to the best of our

knowledge, of combining world models and autonomous rule generation.

3.3 Program Synthesis

Program synthesis represents the task of automatically generating programs to accomplish some task [15]. It is not typically applied to game development, though some prior work exists within the domain of games [23]. Similar to Mechanic Maker, Medeiros et al. [29] synthesize programs that represent strategies based on player behaviors. But they do not apply program synthesis to generate game content.

Yang et al. [48] apply program synthesis for approximating unseen parts of partially observed environments. While Mittelmann et al. [31] 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.

Chapter 4

Mechanic Maker

4.1 What is Mechanic Maker?

Mechanic Maker is a game development tool initially developed by Saini and Guzdial [37]. Mechanic Maker was created as a co-creative way for an AI and a user to work together to create Game Mechanics. The goal with Mechanic Maker is to remove the need for programming in game mechanic development. It works by users creating a mechanic frame-by-frame and having the backend learn the game from the user's example.

4.2 Tool Walkthrough

Our Mechanic Maker tool is split into two different components:

1. The 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 frames to demonstrate their desired mechanics and receive predictions from the SLPS backend.

4.2.1 Symbolic Learning Program Synthesis

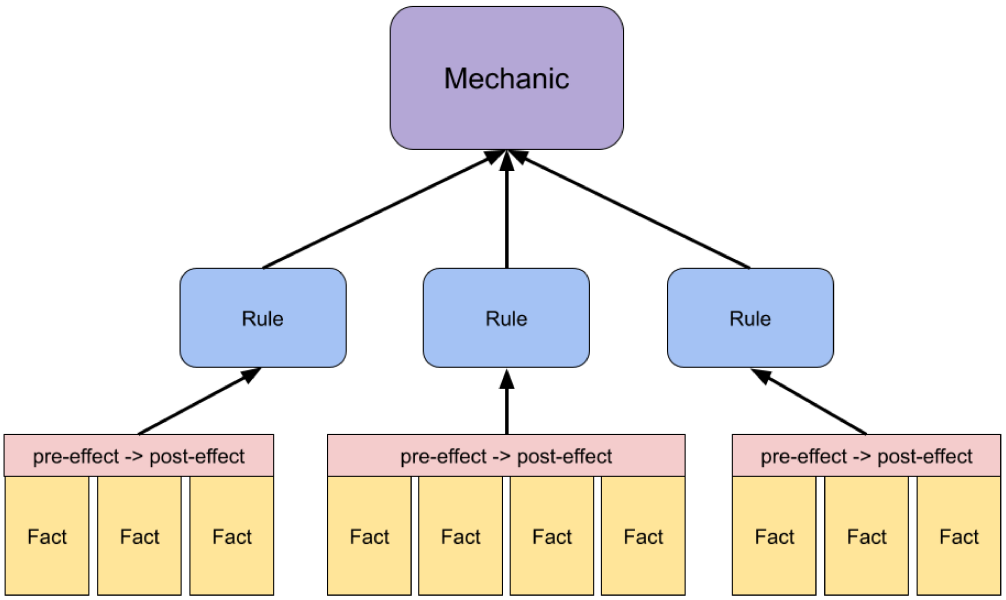


Figure 4.1: High-level overview of the hierarchy of terminology for Mechanic Maker.

The SLPS backend extends the Engine Learning algorithm by Guzdial et al. [16]. This Engine Learning algorithm takes as input a sequence of video game frames and outputs an approximation of game rules. The basis of the algorithm is a representation of a Game Engine as a sequence of game Rules, where each game rule is composed of an "if-then" statement. The "if" portion of each "if-then" statement is a sequence of percept-like variables referred to as "facts" that must be true for the "then" portion to fire. The "then" portion replaces one fact/variable from one state (or frame) called a *pre-effect* with the appropriate value for the next state called a *post-effect* in order to simulate the changes observed in the input frames. The learned engine starts entirely empty without any pre-authored rules, and is iteratively expanded and refined via an A*-like search process. Guzdial et al. found that their algorithm was able to outperform a deep neural network given the same small amount of

gameplay video, and was able to approximate actual game rules with a high degree of accuracy.

Figure 4.2 shows an example of a learned rule. The top line shows the pre-effect of the bird's velocity moving down at a speed of -1, and then changing in the post effect to a y velocity of +1. The first number in the values is the id of the object moving, in this case 0. All the lines below the pre-effect and post-effects are the conditions that must be true in order for the rule to fire. In this case the space bar has to be pressed, the longblock has to be moving left and the bird has to be falling. Multiple of these rules create a learned mechanic, as shown in Figure 4.1.

```
RULE: 2 VelocityYFact: [0, -1.0]->VelocityYFact: [0, 1.0]
    VariableFact: ['space', True]
    VariableFact: ['up', False]
    VariableFact: ['down', False]
    VariableFact: ['left', False]
    VariableFact: ['right', False]
    VariableFact: ['upPrev', False]
    VariableFact: ['downPrev', False]
    VariableFact: ['leftPrev', False]
    VariableFact: ['rightPrev', False]
    VelocityYFact: [1, 0]
    VelocityXFact: [1, -1.0]
    AnimationFact: [1, 'longblock', 1.0, 4.0]
    PositionYFact: [1, 0.0]
    VelocityXFact: [0, 0]
    VelocityYFact: [0, -1.0]
    AnimationFact: [0, 'bird', 1.0, 1.0]
```

Figure 4.2: An example of a rule generated by the backend SMPS approach for the Flappy Bird game to make the bird jump.

Our SLPS algorithm adapted from Guzdial et al. [16] is shown in Algorithm 1. We made the following changes. 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 it observes a new frame, inheriting the last learned engine instead of an initial 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.

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 to do 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 to the Engine Learning algorithm, our SLPS backend can take in new frame content and output a program at runtime that works with the frontend game editor to represent a game's mechanics.

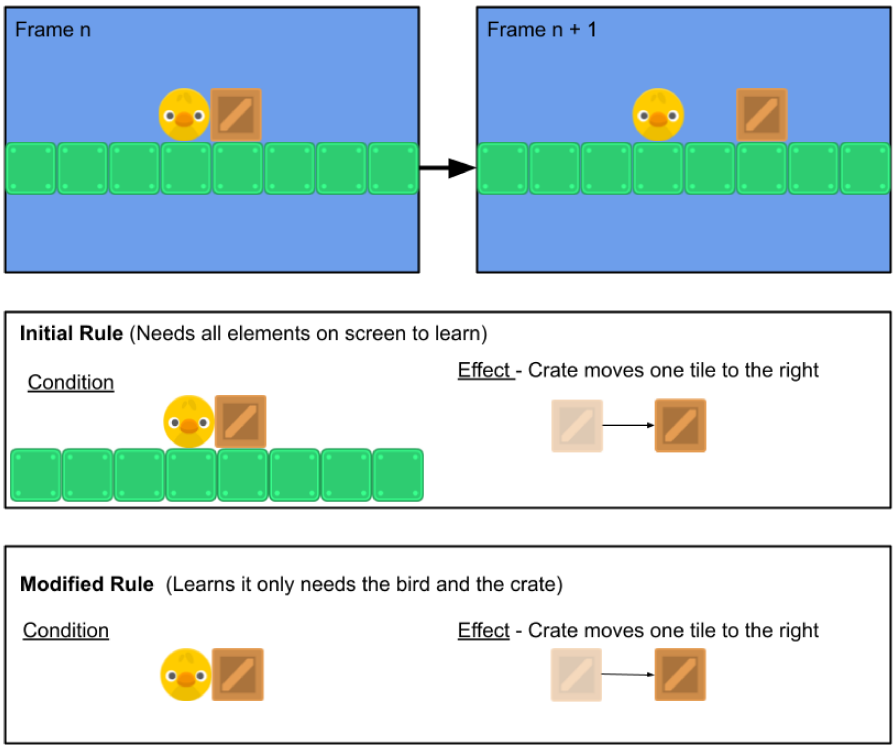


Figure 4.3: The rule generation process showing how a rule gets modified to be more generalized using the SLPS backend. Note that the rule would only be modified if the system observed additional examples of the same effect.

The SLPS backend uses this engine to output rules that can be used to create games. The rules it learns are modified to generalize to the situation. Figure 4.3 shows how this works. When a rule is first created, all the objects are added to the rule as the SLPS backend starts with the assumption that the state relies on all the objects being present. The SLPS backend will then run the Engine Learning algorithm to remove unnecessary facts to create a rule

that is more generalized. In the Figure, initially the ground is necessary for the bird to move the crate one square to the left, but after it runs the Engine Learning algorithm, it learns that the chick and the crate are the only objects necessary for the rule to fire. The type of facts that Mechanic Maker uses to represent each frame and create rules are:

- VariableFact — determines if any input was pressed.
- VelocityXFact — the velocity of an object in the x directions. Contains information for how fast the object is going in the x direction and the ID of the object affected.
- VelocityYFact — the velocity of an object in the y directions. Contains information for how fast the object is going in the y direction and the ID of the object affected.
- PositionXFact — the position of an object in the x directions. Contains information for the location of the object in the x position and the ID of the object affected.
- PositionYFact — the position of an object in the y directions. Contains information for the location of the object in the y position and the ID of the object affected.
- RelationshipXFact — associates the position of an object compared to another object in the x direction. Contains information about the ID of the object and the distance away from another object in the x position.
- RelationshipYFact — associates the position of an object compared to another object in the y direction. Contains information about the ID of the object and the distance away from another object in the y position.
- AnimationFact — provides an ID of an object and its location on screen. This fact allows us to distinguish between multiple instances of the same object.
- EmptyFact — Tracks when objects appear or disappear on the screen.

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

4.2.2 Game Mechanic Editor

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

1. The frame editor shown in Figure 4.4.a.

2. Predictions from the SLPS backend, with an example shown in Figure 4.4.b.

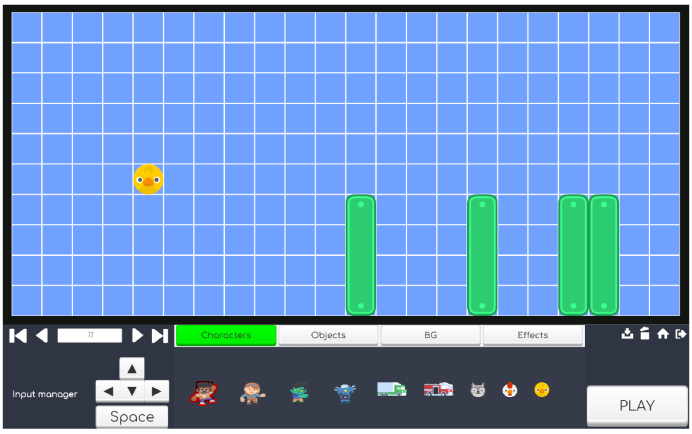
3. The Play Mode shown in Figure 4.4.c.

The frame editor in Figure 4.4.a is the main point of interaction in our tool. We present the controls of our game mechanic editor at the bottom of each

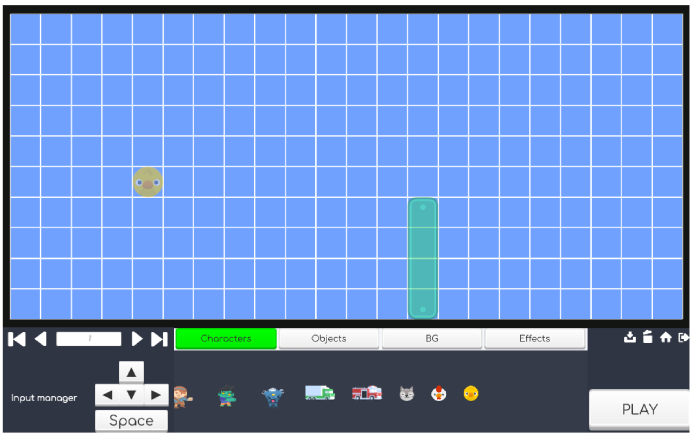
sub-figure in Figure 4.4. The user specifies the intended behaviours for their game mechanics by demonstrating them across these frames. For example, if a user wants to have an object moving to the right, they would add it to the grid for frame 0 then place the same object one position to the right in 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 4.4.b, when the user moves to a new frame they see a semi-transparent "ghosting" of the 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 new frame as they see fit.

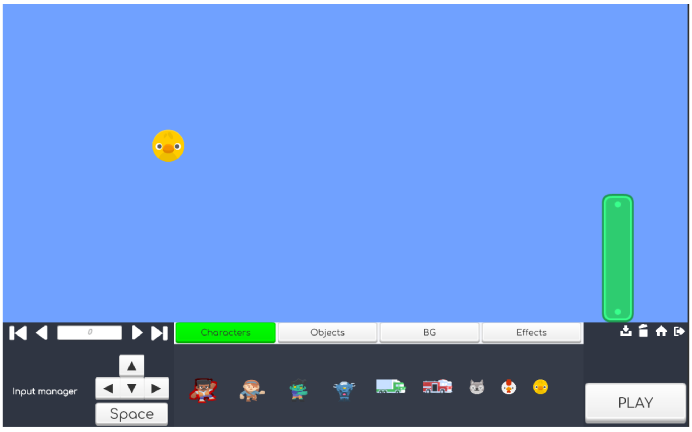
At any time, the player can test the current learned rules in real-time by pressing the play button in Figure 4.4.c. This allows them to verify that the game is working as intended, as shown in Figure 4.4.c. If not, they can keep iterating and adding more frames to correct the SLPS backend's learned mechanics.



(a) Frame Editor for frame 0



(b) SLPS Prediction



(c) Play Mode

Figure 4.4: 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.

Chapter 5

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.

5.1 Hypotheses

We focused on testing two 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. Our second hypothesis is 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 identify two different factors related to game development value and measure them separately. The first factor 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 second factor 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 provide us some validation towards

our research questions. In particular, it would provide us answers toward [R1] because the second hypothesis directly measures the use of Mechanic Maker as a tool. It would also provide answers for [R2] because hypothesis one is determining whether programming experience is required to use Mechanic Maker.

5.2 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. This is because Mechanic Maker requires perfect information to work correctly, so the longer a user works with the tool, the more likely they are to make a mistake that worsens their results.

The first part of the study was a step-by-step, interactive 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 user interface.

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 in-

volving 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 4.4.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 hypothesis that Mechanic Maker did not rely on programming ability. It would also support our hypotheses 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". We included this part to test our second hypothesis. If participants were able to successfully create a wide variety of games, that would support our 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 participant enjoyment factor.

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

5.3 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 [28]. We adapted the first nineteen questions from the Intrinsic Motivation Inventory (IMI) [35] 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 we felt adapting these questions from IMI was sufficient for our initial Mechanic Maker study. The second section included short answer questions for feedback around what users would change and thoughts around the tool. The final section asked demographic questions, including the users programming experience and game development experience, shown in Table 5.1, which we used to evaluate our first hypothesis. We include all original survey questions in the *Survey Questions from the User Study* of the Appendix.

5.4 Results

In our study we collected two types of information. First, from the survey we collected self-reported Likert questions related to the tool, self-reported short answer questions related to the tool 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 two hypotheses.

5.4.1 Participants

We had 16 participants with varying programming and game development backgrounds take part in our study, as shown in Table 5.1. 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 [40].

5.4.2 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

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 5.1: Experience of participants in programming and game development.

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. The Hellman's metric calculates error by matching each object between frames and determining per object, the differences between facts. The more different the facts between the object in the first frame compared to the second frame, the higher the Hellman's metric error will be.

This metric is useful for determining success at replicating the reference games as it gives us an error equivalent to the dissimilarity of the participant's learned mechanics from the intended mechanics.

Baseline

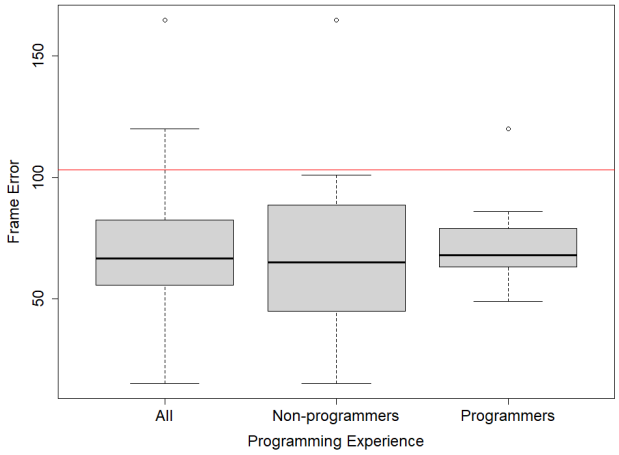
For our baseline, we used the previous example frame with no changes as the prediction, measuring the difference with Hellman's metric as above. This is

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 [16].

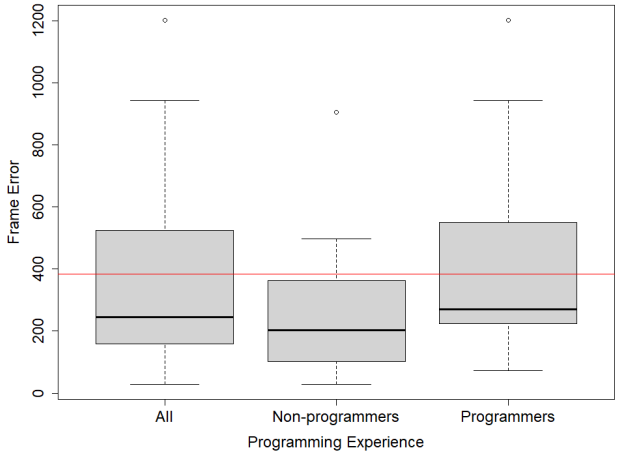
Frame Error Results

The results are shown in Figure 5.1. For both of these plots, we reduced the programming experience groups from our survey participants shown in Table 5.1 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 5.1.a, for the Sokoban part, programming experience did not appear to impact the frame error at all, and both values were below the baseline value (the red line). In the Flappy Bird example in Figure 5.1.b, the same is true. The median values of frame errors are lower than the baseline, indicating that the majority of participants were able to create useful rules. The non-programmer distribution is actually moderately lower than the programmer distribution, however there is no significant difference between the distributions according to the Wilcoxon Mann Whitney U-test. These results support both hypotheses and provide us some useful answers towards [R1] and [R2].



(a) Sokoban Activity



(b) Flappy Bird Activity

Figure 5.1: 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.

We also ran a Pearson correlation test for the frame error for both the Sokoban and Flappy Bird sections 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.

To ensure that programming experience wasn't significant to frame error, we also ran a Mann Whitney test for the Sokoban and Flappy Bird sections. Doing this we were able to determine there is no significant difference between programmer ability when it comes to frame error. For Sokoban we had a p-value of 0.6336 and for Flappy Bird we had a p-value of 0.204.

5.4.3 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 5.2. This provides some evidence to support our second hypothesis and [R1], that Mechanic Maker can create a variety of different game mechanics. We provide further analysis on the variety of games that can be created in the next chapter.

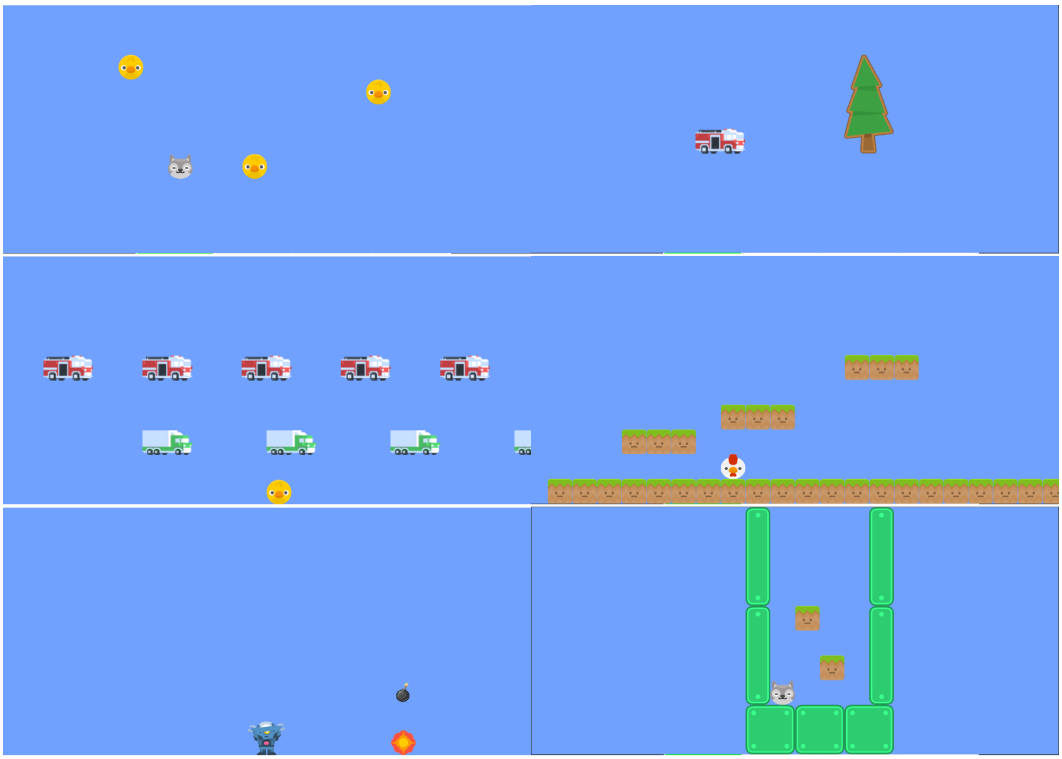


Figure 5.2: Example game outputs from the Free Play portion of the user study.

5.4.4 Survey Results

Figures 5.3 to 5.6 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?
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 5.3 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 5.3, 5.5 and 5.6 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 5.6 participants felt that the tool had value. This supports our second hypothesis that Mechanic Maker provides benefit as a game development tool.

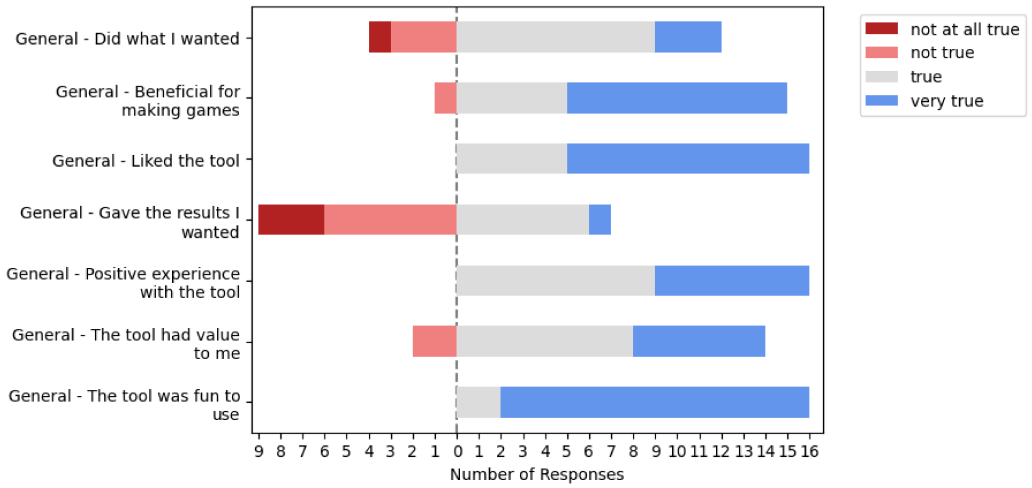


Figure 5.3: Survey results for the tool in general.

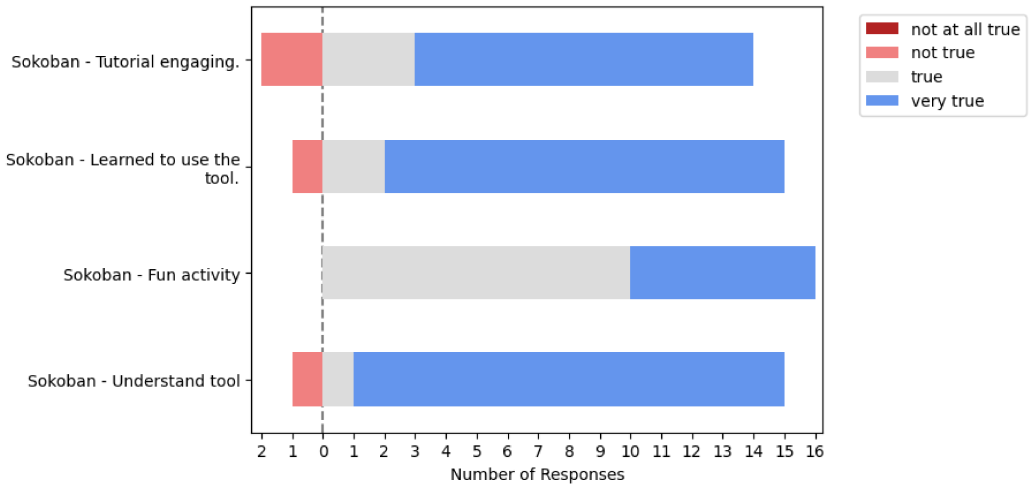


Figure 5.4: Survey results for the tool when doing the Sokoban activity.

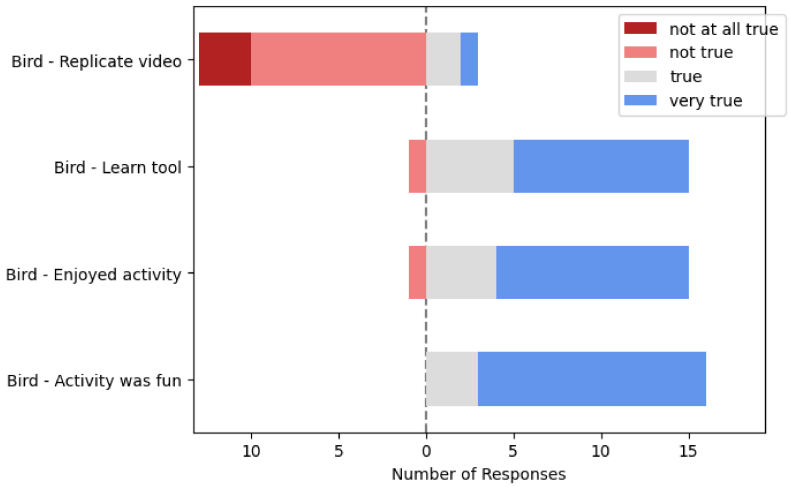


Figure 5.5: Survey results for the tool when doing the Flappy Bird activity.

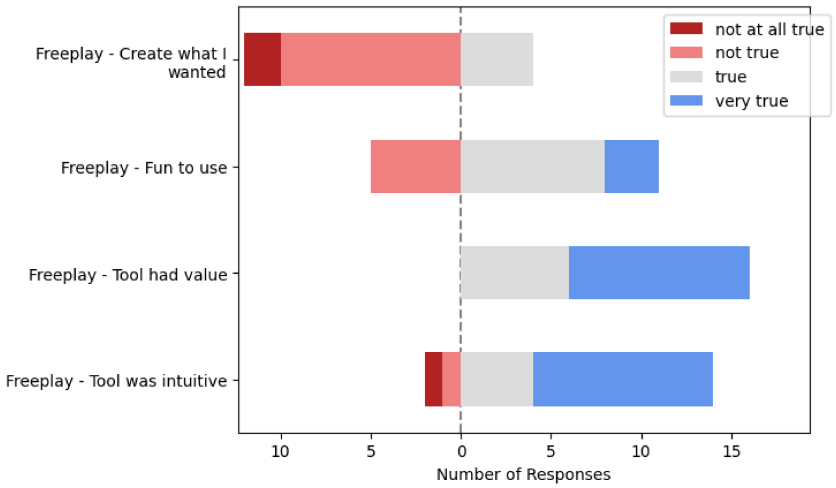


Figure 5.6: Survey results for the tool when doing the Free Play activity.

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

Chapter 6

Gaussian Mixture Model Analysis

In an attempt to better understand the variability of the free play results we created a GMM from the user study data. This was inspired by previous attempts to better understand design tasks through clustering [1]. 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 identify rule clusters. We used the elbow method to determine the number of optimal Gaussians for our data and found that seven clusters created a good representation.

A visualization of the distribution of Gaussians is shown in Figure 6.1. This visualization demonstrates a wide variety of rules learned during the free play portion, with a bias towards velocity rules. The variety of the GMM results provides us some evidence to [R1]. 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 6.1 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. We speculate that this might be due to most of the game examples used in the dataset relying on velocity rules. The only collision rules that would have occurred, outside of rules created in the free play portion of the user study, would have been with the player and the box in the Sokoban tutorial and with the pipes and the left side of the screen in the Flappy Bird section.

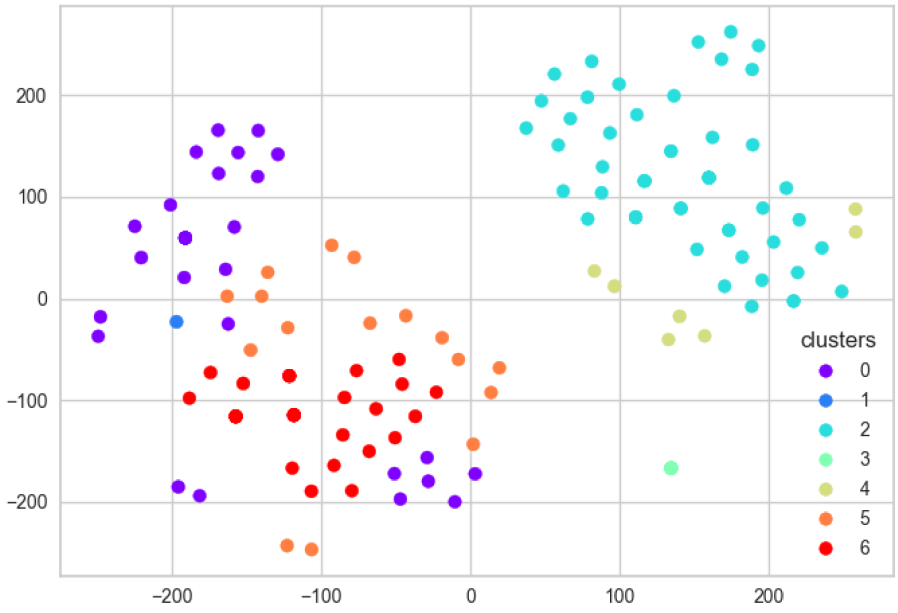


Figure 6.1: T-distributed Stochastic Neighbor Embedding (TSNE) to reduce the 20 dimensions to 2 from the GMM. This is a visualization of 7 clusters from our GMM.

6.1 Case-Based Reasoning

With this dataset of clustered rules, we attempted to approximate rules for new games. This worked by considering pairs of frames (a current and the next frame), and finding the differences between the two frames. We then created potential new rules by considering the differences as pre-effects and post-effects, and the conditions as everything true in the current frame. Thus far, this is similar to our SLPS algorithm.

We then used our GMM as the basis for a Case-based Reasoning approach. We passed our newly created potential rule to our GMM and retrieved the cluster that was closest to this new rule. We modified the rule by reducing the number of conditions by removing them one at a time and seeing if it increased the probability of that rule falling within the chosen cluster. If it did, we removed the condition to simplify the rule because simplifying the rules allowed them to be applied to more generic use cases. We used this simplified rule as our proposed solution.

Figure 6.2 shows the rules that we generated with our case-based reasoning approach for frames one and two of the Flappy Bird example game from the user study. If we compare these results to the rules generated for the same frames from the SLPS approach in Figure 6.3, we can see they are quite different. Our new approach using the GMM dataset does provide less facts overall, which allows for more generic rules quicker, but it also tends to eliminate facts that it should not. We discuss the reasons why this is the case in the limitations section. This new approach does pull a relevant case from the database and modify the case to create a new solution. These results provide some answers to [R3], but as discussed in the limitations section, more work needs to be done towards this question.

```
RULE: 0 VelocityXFact: [0, 0]->VelocityXFact: [0, -1.0]
Predicted Cluster for rule 0 is: Cluster 3
VelocityXFact: [0, 0]
PositionYFact: [1, 0.0]
VelocityYFact: [1, 0]
VariableFact: ['up', False]
VariableFact: ['left', False]
VariableFact: ['spacePrev', False]
VariableFact: ['downPrev', False]
VariableFact: ['rightPrev', False]

RULE: 1 VelocityYFact: [1, 0]->VelocityYFact: [1, -1.0]
Predicted Cluster for rule 1 is: Cluster 2
VelocityXFact: [0, 0]
PositionYFact: [1, 0.0]
VariableFact: ['up', False]
VariableFact: ['right', False]
VariableFact: ['upPrev', False]
VariableFact: ['leftPrev', False]"]
```

Figure 6.2: An example of the rules generated by the case-based reasoning approach for frames one and two of the Flappy Bird example game.

6.1.1 Limitations

Our case-based reasoning approach did not perform as well as desired. We ran into issues with the selection process of the clusters. Most of the time, the new rule would predict a cluster that matched the conditions rather than the pre-effect or post-effect. For example, if the new rule being predicted was a velocity x rule moving from left to right, it might select a cluster in the velocity y direction and not put as much weight on the coordinates moving from left to right. One of the issues with this was that we only had a small amount of clusters that couldn't explain every variation of fact types, fact values and conditions. We attempted to add weightings to different parts of the input to mitigate this. To do this, we added the highest weighting towards the type of the fact, the second highest weighting for the values of the fact and the lowest weighting towards the number of each condition type involved in the rule. This helped the GMM predict based on the fact type, but it still ran into issues with values changing by incorrect amounts.

Our case-based reasoning approach currently finds the most likely cluster and modifies the cluster rule with the information from the frames. It does not currently know which sprite is associated with the rules. Work would need to be done to determine which animation fact is associated with which movement when adjusting the cluster rule. We could then use this or a similar approach to quickly learn rules for new users without needing to start from scratch each time.

The last issue that we ran into was that when we modified the rules to remove conditions from the rules, it would end up removing too many conditions, or focusing on conditions that shouldn't be removed. It ended up causing issues with the output rule not being useful for future occurrences of the same effect. Future work for this research is to focus on semantically-relevant condition removal. This could potentially be done by using a threshold before determining when to remove a condition, or adding weighting to particular condition types to be removed over others.

In addressing our research question, [R3], we conclude that our current

case-based reasoning approach isn't quite at the point that it is a viable method for generating game rules. Either more work needs to be done to get this to be an effective approach, or a more complex model needs to be created to handle the rule data provided by Mechanic Maker.

```
RULE: 0 VelocityYFact: [0, 0]->VelocityYFact: [0, -1.0]
    VelocityXFact: [0, 0]
    VelocityYFact: [0, 0]
    AnimationFact: [0, 'chick', 1.0, 1.0]
    PositionXFact: [0, 3.0]
    PositionYFact: [0, 4.0]
    VelocityXFact: [1, 0]
    VelocityYFact: [1, 0]
    AnimationFact: [1, 'longblock', 1.0, 4.0]
    PositionXFact: [1, 8.0]
    PositionYFact: [1, 0.0]
    VariableFact: ['space', False]
    VariableFact: ['up', False]
    VariableFact: ['down', False]
    VariableFact: ['left', False]
    VariableFact: ['right', False]
    VariableFact: ['spacePrev', False]
    VariableFact: ['upPrev', False]
    VariableFact: ['downPrev', False]
    VariableFact: ['leftPrev', False]
    VariableFact: ['rightPrev', False]
    RelationshipFactX: [0, 1, 'East', 'West', -4.0]
    RelationshipFactY: [0, 1, 'North', 'South', 0.0]
    RelationshipFactX: [1, 0, 'West', 'East', 4.0]
    RelationshipFactY: [1, 0, 'South', 'North', 0.0]
RULE: 1 VelocityXFact: [1, 0]->VelocityXFact: [1, -1.0]
    VelocityXFact: [0, 0]
    VelocityYFact: [0, -1.0]
    AnimationFact: [0, 'chick', 1.0, 1.0]
    PositionXFact: [0, 3.0]
    PositionYFact: [0, 4.0]
    VelocityXFact: [1, 0]
    VelocityYFact: [1, 0]
    AnimationFact: [1, 'longblock', 1.0, 4.0]
    PositionXFact: [1, 8.0]
    PositionYFact: [1, 0.0]
    VariableFact: ['space', False]
    VariableFact: ['up', False]
    VariableFact: ['down', False]
    VariableFact: ['left', False]
    VariableFact: ['right', False]
    VariableFact: ['spacePrev', False]
    VariableFact: ['upPrev', False]
    VariableFact: ['downPrev', False]
    VariableFact: ['leftPrev', False]
    VariableFact: ['rightPrev', False]
    RelationshipFactX: [0, 1, 'East', 'West', -4.0]
    RelationshipFactY: [0, 1, 'North', 'South', 0.0]
    RelationshipFactX: [1, 0, 'West', 'East', 4.0]
    RelationshipFactY: [1, 0, 'South', 'North', 0.0]
```

Figure 6.3: An example of the rule generated by the SLPS backend for frames one and two of the Flappy Bird example game.

Chapter 7

Conclusion

Making game development more accessible to those who might be novices at programming is an important step to democratizing game development. Our research has shown that there is potential for PCGML tools to work with users to reduce these barriers of entry. Mechanic Maker is an example of how co-creative tools can work with users in a beneficial way to create desired games. We believe tools like this are a useful way for AI and users to work together on a common task, while still providing the user all the agency. There are many PCGML approaches that do this, but Mechanic Maker is a tool that allows for users to provide demonstrations and rules to be output that work with the game engine. This process, along with the user working with the AI frame-by-frame in a co-creative way, is novel to PCGML.

7.0.1 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 [3]. 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 [47]. Mechanic Maker does reduce the programming requirement for game development, which

could lead to potential job losses in technical positions. This tool, however, 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 Mechanic Maker co-creatively.

7.0.2 Future Work

We identify that Mechanic Maker tended to perform best with fewer frames and objects due to the tendency for mistakes to occur the longer the tool is used. In future iterations, we hope to improve the performance of the tool. In the survey from our user study, we also asked participants for feedback on the tool itself, which outlined issues with usability. Future iterations of the tool will take this feedback into consideration to help improve the usability of Mechanic Maker.

Currently, we start every game creation session with a blank engine. However, we now have our GMM with 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 our case-based reasoning approach to approximate new rules by querying the GMM. More work needs to be done to determine if this is effective at speeding up the rule learning process.

7.1 Takeaways

We now reflect on our three research questions from the introduction.

Firstly, **can we build a co-creative tool (Mechanic Maker) that can help a user create a variety of game mechanics?** From the results of our user study, we have determined that Mechanic Maker is effective for this as users were able to replicate the mechanics in the Sokoban and Flappy Bird

sections of the user study. It was also shown that there was a high degree of variability in the games created in the free play section, which gives us some evidence that users can create many types of games with Mechanic Maker. Our mixture model also provides evidence towards this question as it was shown that a variety of clusters are created from the user study database. That being said, the survey results provide some evidence that usability with the tool suffers. We expect this is most likely due to the tool needing perfect information in order to work properly. Future iterations of the tool would need to focus on minimizing mistakes made using Mechanic Maker by either changing the user experience of the frontend, or allowing randomness in the backend.

Secondly, **when tasked with recreating a given set of mechanics with Mechanic Maker, will programming ability significantly impact the result?** It was shown that our tool is equally usable for both programmers and non-programmers through the user study we ran. This gives us some reassurance that Mechanic Maker can be useful for reducing the programming barrier for game development. More studies would have to be done to verify that this is the case because the amount of participants involved in the study was small, the amount of non-programmers was lower than the number of programmers and also it is unclear if these results generalize outside of University of Alberta students.

Lastly, **can we use a GMM to improve the backend SLPS engine results to create generalized game rules more quickly?** There is promise for this being true, but requires future work to implement a full case-based reasoning approach. Preliminary results have shown potential for case-based reasoning to be used as a way to learn rules more quickly, though future work must be done to get past the issues we encountered. With either more data to train or more work done to create a more complex case-based reasoning model, there is promise that this method could create generalized game rules more quickly than the SLPS approach.

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.

References

[1] A. Alvarez, J. Font, and J. Togelius, "Toward designer modeling through design style clustering," *IEEE Transactions on Games*, vol. 14, no. 4, pp. 676–686, 2022.

[2] D. Bhaumik, A. Khalifa, and J. Togelius, "Lode encoder: Ai-constrained co-creativity," in *2021 IEEE Conference on Games (CoG)*, 2021, pp. 01–08. DOI: 10.1109/CoG52621.2021.9619009.

[3] M. A. Boden and E. A. Edmonds, "What is generative art?" *Digital Creativity*, vol. 20, no. 1-2, pp. 21–46, 2009. DOI: 10.1080/14626260902867915. eprint: https://doi.org/10.1080/14626260902867915. [Online]. Available: https://doi.org/10.1080/14626260902867915.

[4] J. Bruce, M. Dennis, A. Edwards, *et al.*, *Genie: Generative interactive environments*, 2024. arXiv: 2402.15391 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2402.15391.

[5] M. Chover, C. Marín, C. Rebollo, and I. Remolar, "A game engine designed to simplify 2d video game development," *Multimedia Tools and Applications*, vol. 79, May 2020. DOI: 10.1007/s11042-019-08433-z.

[6] M. Cook, S. Colton, A. Raad, and J. Gow, "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*, Springer, 2013, pp. 284–293.

[7] X. Cui and H. Shi, "A*-based pathfinding in modern computer games," vol. 11, Nov. 2010.

[8] N. Davis, "Human-computer co-creativity: Blending human and computational creativity," *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 9, no. 6, pp. 9–12, Jun. 2021. DOI: 10.1609/aiide.v9i6.12603. [Online]. Available: https://ojs.aaai.org/index.php/AIIDE/article/view/12603.

[9] S. Deterding, J. Hook, R. Fiebrink, *et al.*, "Mixed-initiative creative interfaces," in *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, 2017, pp. 628–635.

[10] Epic Games, *Unreal engine*, 2004. [Online]. Available: https://www.unrealengine.com/ (visited on 02/04/2024).

[11] R. W. Floyd, "Algorithm 97: Shortest path," *Communications of the ACM*, vol. 5, no. 6, pp. 345–345, 1962.

[12] S. Freitas, "Are games effective learning tools? a review of educational games," *Educational Technology and Society*, vol. 21, pp. 74–84, Jan. 2018.

[13] Gitnux, *Diversity in the video game industry: Striking disparities revealed*, 2024. [Online]. Available: https://gitnux.org/diversity-in-the-video-game-industry-statistics/r (visited on 07/17/2024).

[14] J. J. Gonzalez, S. Cooper, and M. Guzdial, "Mechanic maker 2.0: Reinforcement learning for evaluating generated rules," in *Proceedings of the Nineteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, ser. AIIDE '23, Salt Lake City: AAAI Press, 2023, ISBN: 1-57735-883-X. DOI: 10.1609/aiide.v19i1.27522. [Online]. Available: https://doi.org/10.1609/aiide.v19i1.27522.

[15] S. Gulwani, O. Polozov, and R. Singh, "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017, ISSN: 2325-1107. DOI: 10.1561/2500000010. [Online]. Available: http://dx.doi.org/10.1561/2500000010.

[16] M. Guzdial, B. Li, and M. O. Riedl, "Game engine learning from video.," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI-17)*, 2017, pp. 3707–3713.

[17] M. Guzdial, N. Liao, J. Chen, *et al.*, "Friend, collaborator, student, manager," in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, ACM, May 2019. DOI: 10.1145/3290605.3300854. [Online]. Available: https://doi.org/10.1145%2F3290605.3300854.

[18] M. Guzdial and M. O. Riedl, "Conceptual game expansion," *IEEE Transactions on Games*, vol. 14, no. 1, pp. 93–106, 2021.

[19] M. Guzdial, S. Snodgrass, and A. J. Summerville, "Classical pcg," in *Procedural Content Generation via Machine Learning: An Overview*. Cham: Springer International Publishing, 2022, pp. 7–22, ISBN: 978-3-031-16719-5. DOI: 10.1007/978-3-031-16719-5_2. [Online]. Available: https://doi.org/10.1007/978-3-031-16719-5_2.

[20] D. Ha and J. Schmidhuber, "World models," 2018. DOI: 10.5281/ZENODO.1207631. [Online]. Available: https://zenodo.org/record/1207631.

[21] E. Halina and M. Guzdial, *Threshold designer adaptation: Improved adaptation for designers in co-creative systems*, 2022. arXiv: 2205.09269 [cs.LG].

[22] S. W. Kim, Y. Zhou, J. Philion, A. Torralba, and S. Fidler, "Learning to simulate dynamic environments with gamegan," Jun. 2020, pp. 1228–1237. DOI: 10.1109/CVPR42600.2020.00131.

[23] M. Kreminski and M. Mateas, "Opportunities for approachable game development via program synthesis.," in *AIIDE Workshops*, 2021.

[24] J. Kruse, A. M. Connor, and S. Marks, "Evaluation of a multi-agent "human-in-the-loop" game design system," *ACM Trans. Interact. Intell. Syst.*, vol. 12, no. 3, Jul. 2022, ISSN: 2160-6455. DOI: 10.1145/3531009. [Online]. Available: https://doi.org/10.1145/3531009.

[25] G. Lai, F. F. Leymarie, and W. Latham, "On mixed-initiative content creation for video games," *IEEE Transactions on Games*, vol. 14, no. 4, pp. 543–557, 2022. DOI: 10.1109/TG.2022.3176215.

[26] P. Lo, D. Thue, and E. Carstensdottir, "What is a game mechanic?" In *Entertainment Computing – ICEC 2021*, J. Baalsrud Hauge, J. C. S. Cardoso, L. Roque, and P. A. Gonzalez-Calero, Eds., Cham: Springer International Publishing, 2021, pp. 336–347, ISBN: 978-3-030-89394-1.

[27] T. Machado, D. Gopstein, O. Nov, A. Wang, A. Nealen, and J. Togelius, *Evaluation of a recommender system for assisting novice game designers*, 2019. arXiv: 1908.04629 [cs.AI].

[28] R. McGrath, M. Mitchell, B. Kim, and L. Hough, "Evidence for response bias as a source of error variance in applied assessment," *Psychological bulletin*, vol. 136, pp. 450–70, May 2010. DOI: 10.1037/a0019216.

[29] L. Medeiros, D. Aleixo, and L. Lelis, "What can we learn even from the weakest? learning sketches for programmatic strategies," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, pp. 7761–7769, Jun. 2022. DOI: 10.1609/aaai.v36i7.20744.

[30] Media Molecule, *Dreams*, 2020. [Online]. Available: https://indreams.me (visited on 02/04/2024).

[31] M. Mittelmann, B. Maubert, A. Murano, and L. Perrussel, "Automated synthesis of mechanisms," in *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, L. D. Raedt, Ed., Main Track, International Joint Conferences on Artificial Intelligence Organization, Jul. 2022, pp. 426–432. DOI: 10.24963/ijcai.2022/61. [Online]. Available: https://doi.org/10.24963/ijcai.2022/61.

[32] N. Partlan, E. Kleinman, J. Howe, S. Ahmad, S. Marsella, and M. Seif El-Nasr, "Design-driven requirements for computationally co-creative game ai design tools," in *Proceedings of the 16th International Conference on the Foundations of Digital Games*, ser. FDG '21, Montreal, QC, Canada: Association for Computing Machinery, 2021, ISBN: 9781450384223. DOI: 10.1145/3472538.3472573. [Online]. Available: https://doi.org/10.1145/3472538.3472573.

[33] B. Pell, "Metagame in symmetric chess-like games," 1992.

[34] F. Petrillo, M. Pimenta, F. Trindade, and C. Dietrich, "What went wrong? a survey of problems in game development," *Comput. Entertain.*, vol. 7, no. 1, Feb. 2009. DOI: 10.1145/1486508.1486521. [Online]. Available: https://doi.org/10.1145/1486508.1486521.

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

[36] M. Resnick, J. Maloney, A. Monroy-Hernández, *et al.*, "Scratch: Programming for all," *Commun. ACM*, vol. 52, no. 11, pp. 60–67, Nov. 2009, ISSN: 0001-0782. DOI: 10.1145/1592761.1592779. [Online]. Available: https://doi.org/10.1145/1592761.1592779.

[37] V. Saini and M. Guzdial, "A demonstration of mechanic maker: An ai for mechanics co-creation," *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 16, no. 1, pp. 325–327, Oct. 2020. DOI: 10.1609/aiide.v16i1.7450. [Online]. Available: https://ojs.aaai.org/index.php/AIIDE/article/view/7450.

[38] S. Snodgrass and S. Ontañón, "Learning to generate video game maps using markov models," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 9, no. 4, pp. 410–422, 2017. DOI: 10.1109/TCIAIG.2016.2623560.

[39] K. Sorochan and M. Guzdial, "Generating real-time strategy game units using search-based procedural content generation and monte carlo tree search," *arXiv preprint arXiv:2212.03387*, 2022.

[40] Statista, *Distribution of game developers worldwide from 2014 to 2021, by gender*, 2023. [Online]. Available: https://www.statista.com/statistics/453634/game-developer-gender-distribution-worldwide/#statisticContainer (visited on 08/28/2023).

[41] K. T. Stolee and T. Fristoe, "Expressing computer science concepts through kodu game lab," in *Proceedings of the 42nd ACM technical symposium on Computer science education*, 2011, pp. 99–104.

[42] A. Summerville, C. Martens, B. Samuel, J. Osborn, N. Wardrip-Fruin, and M. Mateas, "Gemini: Bidirectional generation and analysis of games via asp," *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 14, no. 1, pp. 123–129, Sep. 2018. DOI: 10.1609/aiide.v14i1.13013. [Online]. Available: https://ojs.aaai.org/index.php/AIIDE/article/view/13013.

[43] U. TECHNOLOGIES, *Speed tree*, 2024. [Online]. Available: https://store.speedtree.com.

[44] J. Togelius and J. Schmidhuber, "An experiment in automatic game design," in *2008 IEEE Symposium On Computational Intelligence and Games*, IEEE, 2008, pp. 111–118.

[45] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011. DOI: 10.1109/TCIAIG.2011.2148116.

[46] M. Treanor and M. Mateas, "An account of proceduralist meaning.," in *DiGRA Conference*, 2013.

[47] V. Vimpari, A. Kultima, P. Hämäläinen, and C. Guckelsberger, ""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*, vol. 7, no. CHI PLAY, pp. 131–164, Sep. 2023. DOI: 10.1145/3611025. [Online]. Available: https://doi.org/10.1145%2F3611025.

[48] Y. Yang, J. P. Inala, O. Bastani, Y. Pu, A. Solar-Lezama, and M. Rinard, "Program synthesis guided reinforcement learning for partially observed environments," in *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., vol. 34, Curran Associates, Inc., 2021, pp. 29 669–29 683. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2021/file/f7e2b2b75b04175610e5a00c1e221ebb-Paper.pdf.

[49] Z. Zhou and M. Guzdial, "Toward co-creative dungeon generation via transfer learning," in *Proceedings of the 16th International Conference on the Foundations of Digital Games*, ser. FDG '21, Montreal, QC, Canada: Association for Computing Machinery, 2021, ISBN: 9781450384223. DOI: 10.1145/3472538.3472601. [Online]. Available: https://doi.org/10.1145/3472538.3472601.

[50] J. Zhu, A. Liapis, S. Risi, R. Bidarra, and G. M. Youngblood, "Explainable ai for designers: A human-centered perspective on mixed-initiative co-creation," in *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, Maastricht, Netherlands: IEEE Press, 2018, pp. 1–8. DOI: 10.1109/CIG.2018.8490433. [Online]. Available: https://doi.org/10.1109/CIG.2018.8490433.

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

Appendix A

Additional Information

A.1 Survey Questions from the User Study

All the Likert questions throughout the survey were rated on a scale of 1 to 4 to remove the neutrality bias. They were pulled from the Intrinsic Motivation Inventory [35] and adapted for our use case. We asked similar questions in both a positive and negative framing.

We started with 7 general questions related to the tool as a whole. These questions related to the usability, enjoyment and value of the tool. The questions were:

1. I was able to use the tool to do what I wanted.
2. I did not find the tool beneficial for making games.
3. I liked the tool.
4. I had a largely negative experience using the tool.
5. The tool did not let me get the results I wanted.
6. I believe that using this tool could be of some value for me.
7. This tool was fun to use.

The final twelve Likert questions were broken into three groups of four questions, each group referencing one of the three game creation parts from the study. The questions were:

1. I found the Sokoban tutorial engaging.
2. Following the tutorial helped me learn how to use the tool.
3. I thought following the Sokoban tutorial was a very boring activity.
4. I did not understand how the tool worked from the tutorial.
5. I found it difficult to replicate the Flappy Bird example.
6. I was able to learn to use the tool through this activity.
7. I enjoyed doing this activity very much.
8. I did not have fun doing this activity.
9. I was able to create what I wanted with the tool in Free Play
10. Using the tool in Free Play was frustrating.
11. I would be willing to do this activity again because it has some value for me.
12. I found the tool too complicated to make what I wanted.

Following these Likert questions we asked three short answer questions in order to identify additional qualitative information related to our hypotheses and to guide future development:

1. How would you change the tool? Assume there's no limit to the possible changes.
2. What would you want us to keep the same about the tool?
3. What aspects of the tool stood out and why?

We ended with ten mixed demographic questions. We ended with the demographic questions in order to minimize the possibility of earlier results being impacted by stereotype threat. These questions were:

1. “What is your gender? (Short answer)”
2. “How old are you?” With the options: 18-24, 25-35, 35-45, 45-55, and 55+
3. “Please pick the category of game design experience that best fits you:” With the options: “No experience (Never designed a game before)”, “Limited game design experience (I’ve tried game development before)”, “Regular game design experience (I’ve worked on multiple game projects before on my own or as a team)”, and “I am a game design expert (Currently or in the past as part of daily life)”
4. “How would you describe your game design experience?” (Short answer)
5. “How often do you play games?” With the options: Daily, A few times a week, Weekly, Monthly, an Less than monthly
6. “Pick the category of programming experience that best fits you:” With the options: “No experience (Never programmed)”, “Limited programming experience (Have programmed before)”, “Regular programming experience (Currently or in the past program weekly or monthly)”, and “Programming expert (Currently or in the past program as part of daily life)”
7. “How would you describe your programming experience?” (Short answer)
8. “Have you ever used a game design tool that did not require programming before? Ex) Scratch, Sony’s Dreams, Project Spark, Game Builder Garage for Nintendo Switch, etc.” With the options Yes and No.
9. “If yes to the above, what was the tool?” (Short answer)
10. “Would you optionally want to provide an email address to be invited to a future study with an updated version of the tool?” (Short answer)

All but the last two questions were non-optional in order to ensure we collected the required results.