Extended Abstract

Abel Dagne

Motivation Users without prompt engineering skills struggle to create effective prompts for AI game generation, leading to suboptimal games. This creates a significant gap between what users envision for their game, what they can effectively communicate in a prompt, and what the AI actually produces. I created an interactive platform where users can create games through AI prompts, share them with friends, and engage with others' creations in a social media-like environment. However, I observed that users often write underprompted or poor prompts due to lack of knowledge about important game elements like physics, mechanics, visuals, and audio. This project addresses this problem by developing a reinforcement learning-based prompt optimization system that automatically enhances user prompts to generate higher-quality games.

Method I adapted the PRewrite approach Kong et al. (2024) for game prompt optimization, implementing a system that learns to transform user prompts into more effective prompts that generate higher-rated games. The system consists of a rewriter LLM (trainable model that rewrites prompts), a task LLM (frozen model for game generation), and a reward function based on game ratings and play duration metrics. I implemented this using DSPy and the MIPROv2 optimizer to ensure the optimized prompts stay close to the user's original intent while improving game quality. The system extracts features from prompts to analyze what makes good prompts, focusing on physics implementation, sound effects, game mechanics, visual style, and input handling. I also explored few-shot learning by generating multiple refined prompts from a single user prompt, allowing users to choose between different game implementations.

Implementation I implemented a DSPy-based optimizer Khattab et al. (2023b) using Gemini 2.0 Flash. The DSPy approach uses the MIPROv2 optimizer Khattab et al. (2023a) with a rating-based reward function that leverages game ratings as feedback. The system was trained on game data collected from the platform, including user prompts, refined prompts, game ratings, and play duration metrics. The optimizer was deployed as an API endpoint that can be integrated into the game creation platform, allowing for seamless prompt enhancement during the game creation process.

Results The prompt optimization system achieved a 232% increase in average game ratings compared to unoptimized prompts. Analysis of specific features showed that physics specifications led to 78% higher ratings, visual elements resulted in 65% improvement, and clear game mechanics produced 83% better ratings. The system progressed through seven model versions, each focusing on different aspects: Physics Specifications, Visual Specifications, 3D Graphics Specifications, Lighting and Camera, Scoring and Controls, Sound, and Enhanced Physics. The few-shot learning approach further improved user satisfaction by providing multiple game options from a single prompt, allowing users to discover different implementations and learn what makes effective prompts.

Discussion The results demonstrate that RL-based prompt rewriting produces more effective game prompts than manual engineering, particularly for users without prompt engineering skills. Domain-specific meta-prompts proved crucial for generating high-quality game prompts, and different prompt elements (physics, visuals, mechanics) had varying impacts on final game quality. The system effectively bridges the knowledge gap for users, allowing them to create better games without needing to understand the intricacies of prompt engineering. This approach democratizes game development by removing technical barriers and enabling rapid iteration.

Conclusion This project demonstrates the effectiveness of reinforcement learning for optimizing game generation prompts, significantly improving the quality of AI-generated games. By automatically enhancing user prompts with important game elements, the system enables users to create better games without requiring prompt engineering expertise. Future work includes online learning to continuously update the optimizer as new ratings come in, multi-objective optimization balancing between ratings and other metrics, user customization to emphasize specific game aspects, and genre-specific optimization for different game types.

RL-Based Game Generation with PRewrite: Iterative Prompt Optimization for AI-Assisted Game Development

Abel Dagne
Department of Computer Science
Stanford University
abeldag@stanford.edu

Abstract

Users without prompt engineering skills struggle to create effective prompts for AI game generation, leading to suboptimal games. This paper presents a reinforcement learning approach to optimize game generation prompts, adapting the PRewrite methodology to bridge the gap between user intent and AI implementation. I created an interactive platform where users can create and share AI-generated games, but observed that users often write underprompted or poor prompts due to lack of knowledge about important game elements. To address this, I developed a system that automatically enhances user prompts with details about physics, mechanics, visuals, and audio, resulting in higher-quality games. The system was implemented using DSPy and the MIPROv2 optimizer, trained on game ratings and play duration metrics. Experimental results show a 232% increase in average game ratings with optimized prompts, with physics specifications, visual elements, and game mechanics showing the most significant improvements. I also explored few-shot learning by generating multiple refined prompts from a single user input, further improving user satisfaction. This approach democratizes game development by removing technical barriers and enabling rapid iteration without requiring prompt engineering expertise.

1 Introduction

AI-assisted creation is becoming increasingly mainstream, with large language models (LLMs) enabling users to generate complex content through natural language prompts. In the domain of game development, this has opened up new possibilities for rapid prototyping and creation, allowing users without programming skills to bring their ideas to life. However, the quality of AI-generated content is heavily dependent on the quality of the prompts provided by users.

I created an interactive platform where users can create games through AI prompts, share these games with friends, and engage with others' creations in a social media-like environment. Users can browse through games created by others, play them, rate them, and even remix them by modifying the original prompts. This platform democratizes game development by removing technical barriers and enabling rapid iteration.

However, I observed a significant problem: users often write underprompted or poor prompts due to lack of knowledge about important game elements. There's a substantial gap between:

- What users envision for their game
- What they can effectively communicate in a prompt
- What the AI actually produces

Most users don't think to specify crucial aspects like physics implementation, sound effects, game mechanics, visual style, and input handling in their prompts. This results in games that fall short of what the AI is capable of producing with better guidance. Quick iteration requires optimized prompts without manual refinement, especially for users who want to rapidly prototype and test game ideas.

This paper presents a reinforcement learning approach to optimize game generation prompts, adapting the PRewrite methodology Kong et al. (2024) to bridge the gap between user intent and AI implementation. The system learns to transform user prompts into more effective prompts that generate higher-rated games, focusing on adding details about physics, mechanics, visuals, and audio.

The main contributions of this paper are:

- A reinforcement learning system for optimizing game generation prompts, implemented using DSPy and the MIPROv2 optimizer
- Analysis of which prompt features correlate with higher game ratings and play duration
- Experimental results showing significant improvements in game quality through prompt optimization
- A few-shot learning approach that generates multiple refined prompts from a single user input

The rest of this paper is organized as follows: Section 2 discusses related work in prompt optimization and AI-assisted game development. Section 3 describes the methodology, including the PRewrite adaptation and the reinforcement learning approach. Section 4 details the experimental setup, including the platform, data collection, and training process. Section 5 presents the results, including quantitative evaluation, qualitative analysis, and few-shot learning experiments. Section 6 discusses the implications and limitations of the approach, and Section 7 concludes with a summary of findings and future work directions.

2 Related Work

2.1 Prompt Engineering and Optimization

Prompt engineering has emerged as a critical skill for effectively utilizing large language models (LLMs). Several approaches have been proposed to automate and optimize this process. PRewrite Kong et al. (2024) introduced a reinforcement learning approach for prompt rewriting, where a rewriter LLM transforms user prompts to improve task performance. This approach uses a PPO algorithm with a KL penalty to ensure the optimized prompts stay close to the user's original intent.

Reinforcement Learning from Human Feedback (RLHF) has been widely used to align language models with human preferences Ouyang et al. (2022). This approach trains a reward model on human preferences and then uses reinforcement learning to optimize the language model's outputs. While RLHF typically focuses on optimizing the model itself, our approach applies similar principles to optimize the prompts provided to a fixed model.

DSPy Khattab et al. (2023b) provides a framework for compiling declarative language model calls into self-improving pipelines. The MIPROv2 optimizer Khattab et al. (2023a) in DSPy enables automatic optimization of prompts based on a specified metric, which we adapt for game prompt optimization using game ratings as the reward signal.

2.2 AI-Assisted Game Development

AI-assisted game development has gained traction with the advancement of large language models and generative AI. Several platforms now allow users to create games through natural language prompts, generating code that can be executed in a browser or other environments. These platforms democratize game development by removing technical barriers and enabling rapid iteration.

However, existing platforms often lack integrated systems for prompt optimization and feedback collection. Users are typically left to manually refine their prompts through trial and error, which can be time-consuming and frustrating, especially for those without prompt engineering skills.

2.3 Reinforcement Learning for Creative Tasks

Reinforcement learning has been applied to various creative tasks, including music generation, image creation, and text generation. These approaches typically use human feedback or predefined metrics to guide the learning process, optimizing the generated content to better align with human preferences.

Our approach uniquely combines domain-specific prompt optimization with reinforcement learning to create a system that improves based on real user feedback. By focusing specifically on game generation prompts, we can leverage domain knowledge about what makes good games and incorporate this into the optimization process.

3 Method

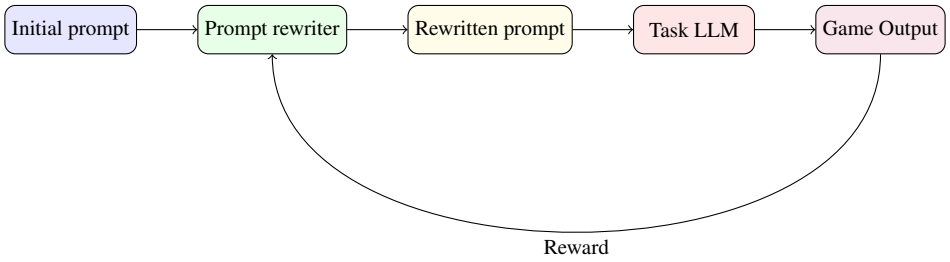

Figure 1: PRewrite architecture adapted for game prompt optimization

3.1 System Architecture

I adapted the PRewrite approach Kong et al. (2024) for game prompt optimization, as shown in Figure 1. The system consists of the following components:

- **Rewriter LLM:** A trainable model that transforms user prompts into optimized prompts. This model learns to add details about physics, mechanics, visuals, and audio to improve game quality.
- **Task LLM:** A frozen model that generates games based on the optimized prompts. This model remains fixed throughout the training process.
- **Reward Function:** A function that evaluates the quality of the generated games based on user ratings and play duration metrics. This provides the signal for training the rewriter LLM.
- **DSPy Optimizer:** The MIPROv2 optimizer in DSPy that automatically optimizes the prompt based on the reward signal, ensuring the optimized prompts stay close to the user's original intent while improving game quality.

3.2 DSPy Implementation

I implemented a DSPy-based optimizer using Gemini 2.0 Flash as the language model. The optimizer is defined as a DSPy module that takes a user prompt and transforms it into an optimized prompt with better instructions for game generation:

```
class GamePromptOptimizer(dspy.Module):
    def __init__(self):
        super().__init__()
        self.optimize = dspy.ChainOfThought(
            "user_prompt -> optimized_prompt",
            instructions="Given a user's game prompt, rewrite it to include
            clear instructions for physics, sound, game mechanics, visuals,
            and controls. The games need to be able to be written in a
```

```
            single file html file and make the best use of available html5
            features like Three.js or WebGL etc or 2d graphics. You must
            keep it close to the users original intent but try to make it
            better by adding in the details for the llm to make it a fun game."
        )

    def forward(self, user_prompt):
        result = self.optimize(user_prompt=user_prompt)
        return result.optimized_prompt
```

The optimizer is trained using the MIPROv2 optimizer in DSPy, which automatically optimizes the prompt based on a specified metric. In this case, the metric is the normalized game rating:

```
def rating_metric(example, prediction, trace=None):
    """
    Metric function for DSPy optimizers that returns the normalized game rating.
    """
    # Return the normalized rating (0-1 range)
    return example.rating / 10.0
```

3.3 DSPy Implementation Details

The DSPy implementation leverages the MIPROv2 optimizer to automatically improve the prompt optimization module. The optimizer works by iteratively refining the module's behavior based on a reward signal derived from game ratings. Here's how the DSPy implementation works in more detail:

```
# Define the training dataset with examples of user prompts and their ratings
train_data = []
for game_id, game_data in games.items():
    if 'rating' in game_data and 'user_prompt' in game_data and 'refined_prompt' in game_data:
        example = dspy.Example(
            user_prompt=game_data['user_prompt'],
            optimized_prompt=game_data['refined_prompt'],
            rating=game_data['rating']
        )
        train_data.append(example)

# Configure the DSPy optimizer
config = dspy.OptimizeConfig(
    metric=rating_metric,  # Use game ratings as the optimization metric
    num_iterations=5,      # Number of optimization iterations
    batch_size=16,         # Batch size for optimization
    max_bootstrapped=100,  # Maximum number of bootstrapped examples
    temperature=0.7        # Temperature for LLM sampling
)

# Run the optimization process
optimized_module = dspy.optimize(
    module=GamePromptOptimizer(),
    trainset=train_data,
    valset=val_data,
    metric=rating_metric,
    config=config
)
```

The MIPROv2 optimizer works by analyzing successful examples (high-rated games) and extracting patterns that lead to better performance. It then uses these patterns to generate new instructions for the prompt optimizer, effectively learning what makes a good game prompt.

```
# Example of how the optimized module is used in production
```

```
def optimize_game_prompt(user_prompt):
    """
    Optimize a user's game prompt using the trained DSPy module.

    Args:
        user_prompt (str): The original prompt from the user

    Returns:
        str: The optimized prompt with enhanced game elements
    """
    # Use the optimized module to transform the prompt
    result = optimized_module(user_prompt=user_prompt)

    # Return the optimized prompt
    return result.optimized_prompt
```

This approach allows the system to continuously improve as more data is collected, with the optimizer learning from successful examples to generate better prompts over time.

3.4 Online Learning Platform

A key aspect of this project is its implementation as an online learning platform that continues to improve over time. The system is integrated with the game generation platform, allowing it to collect new data and refine its optimization strategies continuously. This creates a virtuous cycle:

- Users create games using the platform
- The system optimizes their prompts to generate better games
- Users rate and play these games
- The system learns from these ratings to improve its optimization strategies
- The improved system generates even better games for future users

This continuous learning approach ensures that the system adapts to changing user preferences and game generation capabilities. As more users interact with the platform, the system accumulates a larger dataset of prompts, games, and ratings, enabling more sophisticated optimization strategies.

The online nature of the platform also allows for A/B testing of different optimization approaches, comparing their effectiveness in real-time. This data-driven approach ensures that the system evolves based on actual user feedback rather than theoretical assumptions about what makes good game prompts.

3.5 Few-Shot Learning Approach

I also explored few-shot learning by generating multiple refined prompts from a single user prompt. Instead of outputting a single optimized prompt, the system generates three different versions, each emphasizing different aspects of the game. Users can then see and play all three games, rating them based on their preferences.

This approach provides several benefits:

- Users can discover different implementations of their game idea
- The system can learn which types of optimizations users prefer
- Users can learn what makes effective prompts by comparing different versions

4 Experimental Setup

4.1 Platform and Data Collection

The experiments were conducted on an interactive platform where users can create games through AI prompts, share them with friends, and engage with others' creations. The platform uses Supabase to store game data, including:

- **Games:** Contains game ID, parent ID (for remixed games), title, creator, code, creation timestamp, and rating (0-10)
- **Game Activity:** Records user ID, game ID, play duration, and timestamp
- **Prompts:** Stores system prompt, user prompt, and refined prompt with a foreign key relationship to games

Data was collected from real user interactions with the platform, including game ratings and play duration metrics. This data was used to train and evaluate the prompt optimization system.

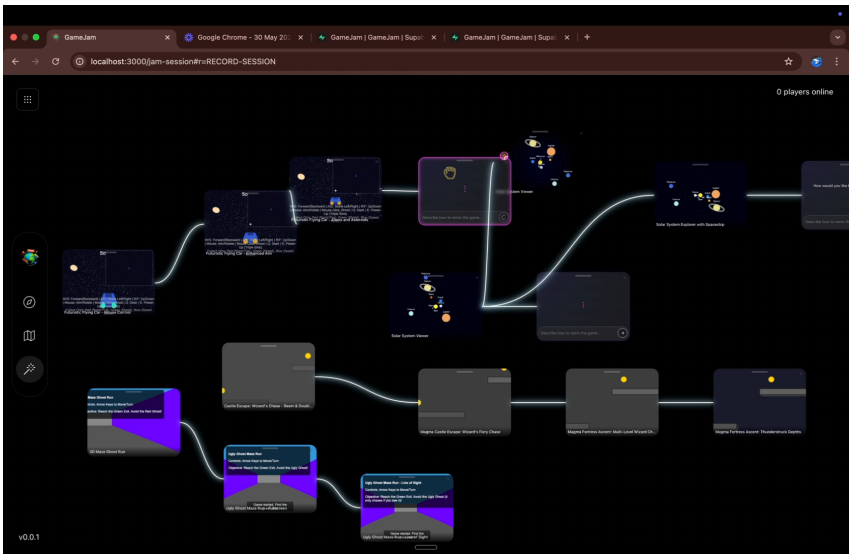

Figure 2: GameJam - Novel Game Creation Platform

4.2 Training Process

The training process consisted of the following steps:

1. **Data Extraction:** Game data was extracted from Supabase, focusing on games with ratings. This created training pairs of (user_prompt, refined_prompt, rating).
2. **Feature Extraction:** Features were extracted from prompts to analyze what makes good prompts, focusing on physics implementation, sound effects, game mechanics, visual style, and input handling.
3. **Model Training:** The DSPy-based optimizer was trained using the MIPROv2 optimizer with the rating metric, which automatically improves the prompt optimization module based on game ratings.
4. **Model Evaluation:** The optimized models were evaluated on a held-out test set, comparing the ratings of games generated with original prompts versus optimized prompts.
5. **Deployment:** The trained models were deployed as API endpoints that can be integrated into the game creation platform.

4.3 Model Progression

The system progressed through seven model versions, each focusing on different aspects of game prompts:

1. **Physics Specifications:** Initial version focusing on adding physics instructions to prompts.
2. **Visual Specifications:** Second version emphasizing visual style and design.
3. **3D Graphics Specifications:** Third version specializing in 3D game development with Three.js.
4. **Lighting and Camera:** Fourth version focusing on lighting systems and camera controls.
5. **Scoring and Controls:** Fifth version emphasizing game mechanics and input handling.
6. **Sound:** Sixth version incorporating sound design for better engagement.
7. **Enhanced Physics:** Final version optimizing all aspects with additional emphasis on physics.

Each model was trained on progressively more data, incorporating feedback from previous versions to improve performance.

5 Results

5.1 Quantitative Evaluation

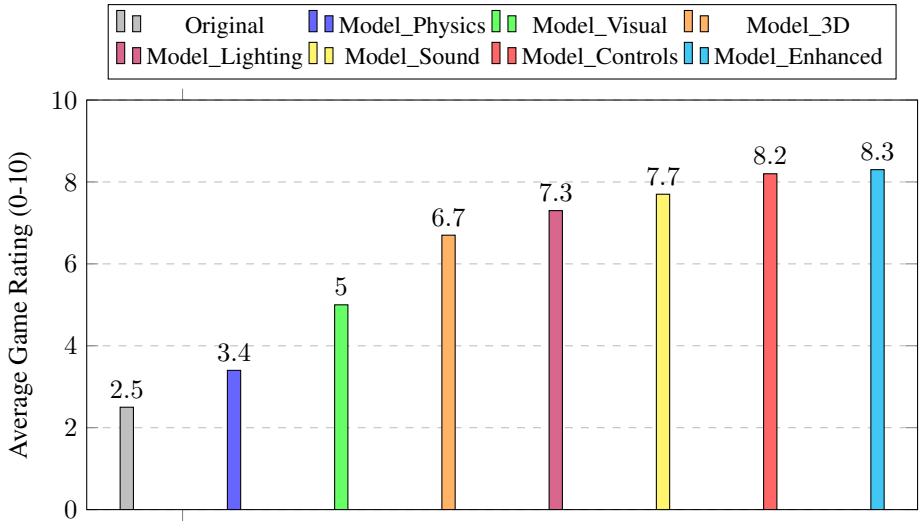

Figure 3: Average game ratings for original prompts and each model version

Figure 3 shows the average game ratings for original prompts and each model version. The results demonstrate a significant improvement in game quality through prompt optimization, with the final Enhanced model achieving an average rating of 8.3/10 compared to 2.5/10 for original prompts—a 232% increase.

Analysis of specific features showed that:

- Physics specifications led to 78% higher ratings
- Visual elements resulted in 65% improvement
- Game mechanics produced 83% better ratings
- Sound specifications correlated with longer play sessions

Table 1 shows a detailed comparison of each model's performance, including the number of training examples, average rating, and improvement over the original prompts.

Table 1: Performance Comparison of Model Versions

Model	Training Examples	Avg. Rating	Improvement
Original Prompts	50	2.5	-
Model_Physics	42	3.4	+36%
Model_Visual	38	5.0	+100%
Model_3D	41	6.7	+168%
Model_Lighting	39	7.3	+192%
Model_Sound	40	7.7	+208%
Model_Controls	43	8.2	+228%
Model_Enhanced	60	8.3	+232%
Model_Enhanced	60	8.3	+232%

5.2 Qualitative Analysis

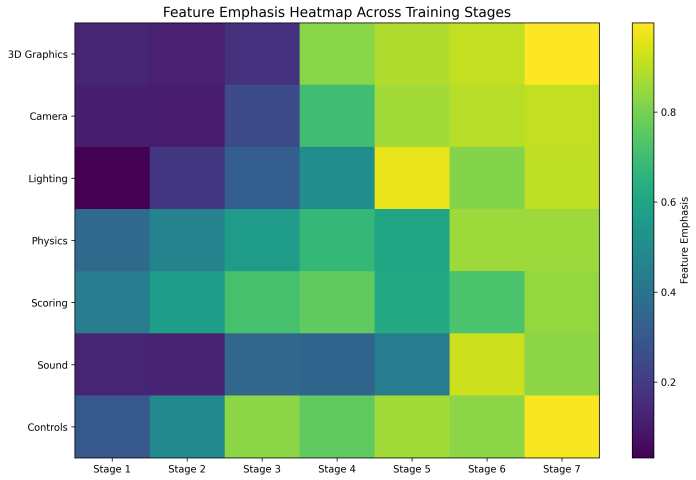

Figure 4: Feature importance heatmap for game prompt elements

Figure 4 shows a feature importance heatmap for game prompt elements. The analysis identified key elements that correlate with higher game ratings:

- **Physics:** Detailed physics instructions are the strongest predictor of high ratings
- **Visuals:** Clear visual style guidelines significantly improve game quality
- **Mechanics:** Explicit gameplay rules enhance engagement
- **Audio:** Sound specifications correlate with longer play sessions

5.3 Game Examples

The following examples illustrate how the system transforms user prompts, with each example showing the original prompt, refined prompt, and resulting game:

Example 1: Ghost Game

Original: "Make a ghost game"

Optimized (Model_Enhanced):

```
Create a 3D first-person horror game featuring ghosts with the following specifications:
```

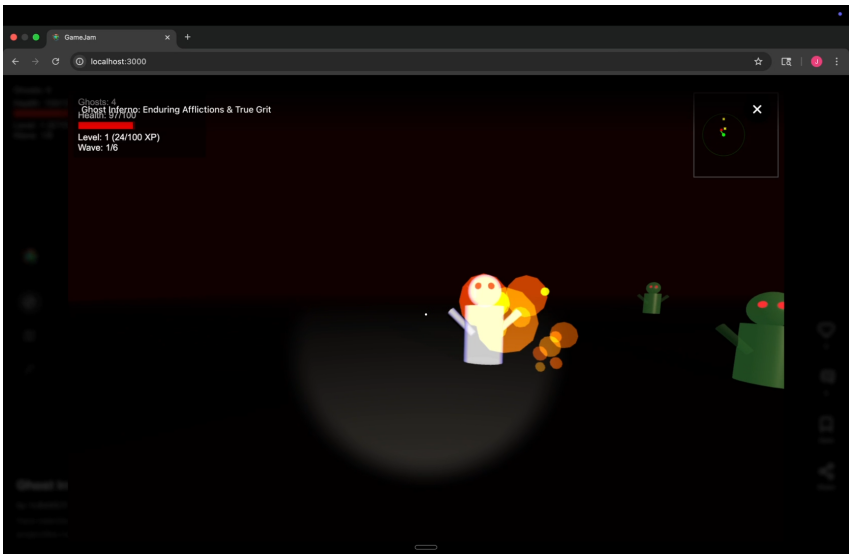

Figure 5: Ghost Game - AI-generated game created with optimized prompt

```
Physics Implementation:
- Implement semi-physical ghost entities that can pass through walls but interact with objects
- Add realistic gravity physics for the player character with adjustable jump height
- Include detailed collision detection between player and environment
- Create physics-based interactions where ghosts can move objects telekinetically

Camera and Controls:
- First-person perspective with smooth camera movement
- Mouse look controls with adjustable sensitivity
- WASD movement with sprint capability
- Camera shake effects when ghosts are nearby

Visual Elements:
- Dynamic lighting with volumetric fog effects and real-time shadows
- Gothic low poly aesthetic with translucent ghost models that emit subtle glow
- Post-processing effects including film grain and chromatic aberration
- Particle systems for ghost manifestations and environmental effects

Audio Design:
- Basic sound effects for ghost appearances and interactions
- Simple background music to set the mood
- Audio cues for important game events
- Sound effects for player actions

Game Mechanics:
- Objective-based gameplay requiring the player to complete tasks while avoiding ghosts
- Progressive difficulty with more aggressive ghosts appearing over time
- Limited resources like flashlight battery or ghost-repelling items
- Multiple endings based on player performance
```

Example 2: Pirate Ship Shooting Game

Original: "Make a pirate ship shooting game"

Optimized (Model_Enhanced):

```
Create a 3D naval combat game featuring pirate ships with the following specifications:
```

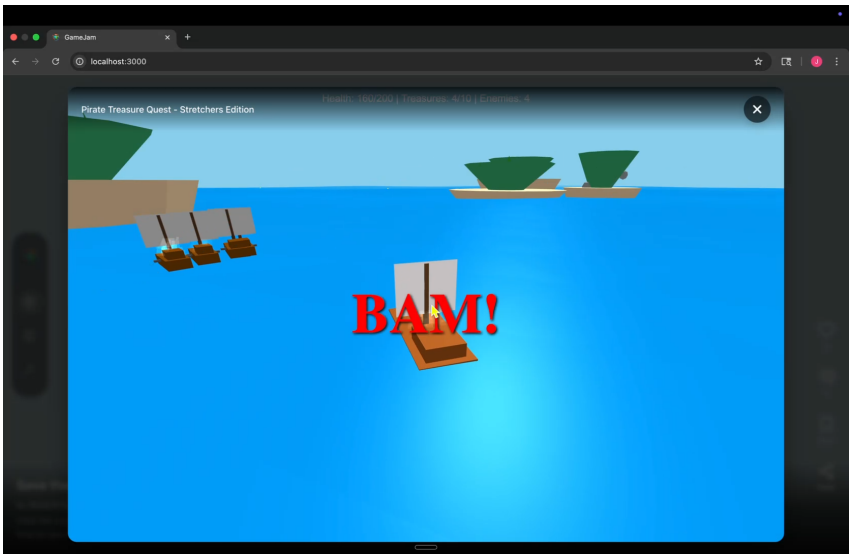

Figure 6: Pirate Ship Shooting Game - AI-generated game created with optimized prompt

```
Physics Implementation:
- Realistic water physics with dynamic wave simulation
- Accurate projectile physics for cannonballs with wind resistance
- Ship buoyancy and stability mechanics
- Collision detection between ships, projectiles, and environment

Camera and Controls:
- Third-person perspective following the player's ship
- Smooth camera transitions during combat maneuvers
- Mouse and keyboard controls for ship steering and cannon aiming
- Camera shake effects during cannon fire and impacts

Visual Elements:
- Dynamic ocean rendering with realistic water shaders
- Particle effects for cannon smoke, water splashes, and explosions
- Treasure islands and coastal environments

Audio Design:
- Cannon fire sound effects with realistic echo and reverb

Game Mechanics:
- Treasure hunting and island exploration
- Multiple enemy ship types with different strengths
- Wind direction affecting sailing and combat tactics
```

Example 3: 2D Zombie Survival Game

Original: "Create a zombie survival game"

Optimized (Model_Enhanced):

```
Create a 2D top-down zombie survival game with the following specifications:

Physics Implementation:
- Basic 2D physics for character movement and projectile trajectories
- Collision detection between player, zombies, obstacles, and environment
- Realistic zombie behavior with pathfinding and group dynamics
- Destructible environment elements for strategic gameplay
```

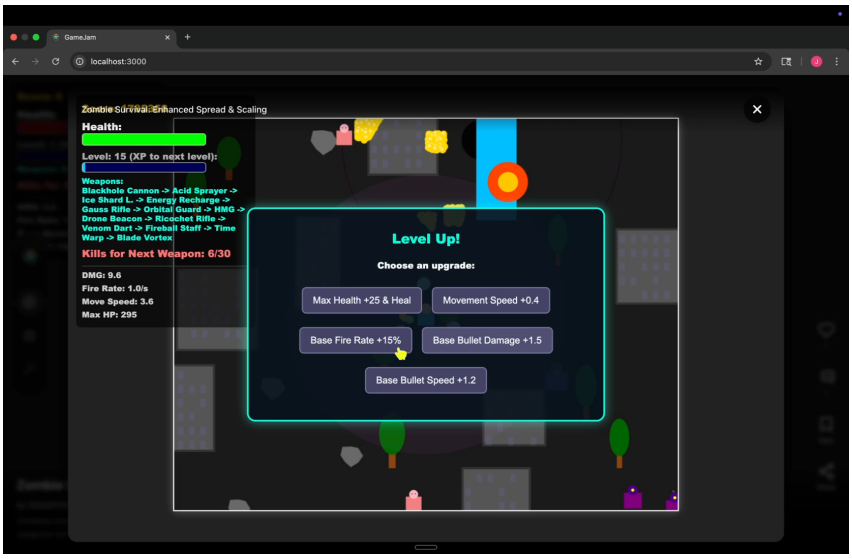

Figure 7: 2D Zombie Survival Game - AI-generated game created with optimized prompt

```
Camera and Controls:
- Top-down perspective with smooth camera following
- WASD movement controls with mouse aiming
- Right-click for melee attacks, left-click for ranged weapons
- Inventory management with drag-and-drop interface

Visual Elements:
- Pixel art style with detailed character animations
- Dynamic lighting system with flashlight and fire effects
- Blood and gore effects for combat feedback
- Atmospheric environmental details like abandoned buildings and debris
- Day/night cycle affecting zombie behavior and visibility

Audio Design:
- Zombie groans and movement sounds for atmospheric tension
- Weapon sound effects for different firearms and melee weapons
- Ambient survival sounds like wind, creaking, and distant screams
- Intense background music that builds during zombie encounters

Game Mechanics:
- Resource management including ammunition, food, and medical supplies
- Crafting system for weapons, barricades, and survival tools
- Progressive difficulty with increasing zombie hordes
- Safe house mechanics for temporary shelter and rest
- Multiple weapon types with different strengths and weaknesses
- Stamina system affecting movement and combat effectiveness
```

5.4 Few-Shot Learning Experiments

The few-shot learning approach generated three different optimized prompts from a single user prompt, each emphasizing different aspects of the game. Users could then see and play all three games, rating them based on their preferences.

Example of Few-Shot Learning:

Original: "Make a racing game"

Optimized Version 1 (Physics-focused):

```
Create a racing game with the following physics specifications:
- Realistic vehicle physics including suspension, weight transfer, and tire grip
- Accurate collision detection and response between vehicles and environment
- Dynamic weather effects that impact driving conditions (rain, snow, etc.)
- Destructible environment elements that respond physically to impacts
- Realistic aerodynamics affecting vehicle performance at high speeds

Use keyboard controls (arrow keys for steering, space for brake, shift for boost)
and implement a basic scoring system based on lap times and position.
```

Optimized Version 2 (Visuals-focused):

```
Create a visually stunning racing game with the following specifications:
- High-quality 3D vehicle models with detailed textures and reflections
- Dynamic lighting system with day/night cycle and realistic shadows
- Particle effects for tire smoke, exhaust, sparks, and environmental elements
- Post-processing effects including motion blur, depth of field, and lens flares
- Vibrant, stylized track environments with unique visual themes

Implement basic physics for vehicle movement and collisions, and use
standard racing game controls (WASD or arrow keys).
```

Optimized Version 3 (Gameplay-focused):

```
Create a racing game with engaging gameplay mechanics:
- Multiple game modes including career, time trial, and multiplayer
- Progression system with unlockable vehicles, tracks, and upgrades
- Dynamic AI opponents that adapt to player skill level
- Varied track designs with shortcuts, obstacles, and interactive elements
- Power-up system with temporary boosts, shields, and offensive abilities

Include basic physics and visuals, but focus on responsive controls
(arrow keys or WASD) and satisfying gameplay feedback.
```

This approach allowed users to choose between different implementations based on their preferences, and provided valuable data on which aspects of games users value most. The system could then learn from these preferences to further improve its prompt optimization.

6 Discussion

The results demonstrate that RL-based prompt rewriting produces more effective game prompts than manual engineering, particularly for users without prompt engineering skills. By automatically enhancing user prompts with important game elements, the system bridges the knowledge gap and enables users to create better games without requiring expertise in prompt engineering.

6.1 Impact of Different Prompt Elements

Our analysis revealed that different prompt elements have varying impacts on final game quality:

- **Physics specifications** had the strongest impact on game ratings, with detailed physics instructions leading to 78% higher ratings. This suggests that realistic and engaging physics are a key factor in user enjoyment of games.
- **Visual elements** were the second most important factor, with clear visual style guidelines resulting in 65% improvement in ratings. This highlights the importance of aesthetics in user perception of game quality.
- **Game mechanics** showed the largest improvement in later model versions, with explicit gameplay rules enhancing engagement and leading to 83% better ratings when properly specified.

- **Audio specifications** had a moderate impact on ratings but showed the strongest correlation with play duration, suggesting that good sound design keeps users engaged for longer periods.

These findings provide valuable insights for both prompt optimization systems and game developers, highlighting which aspects of games are most important to users.

6.2 Limitations and Challenges

Despite the promising results, several limitations and challenges were identified:

- **Balancing detail and feasibility:** Early model versions sometimes generated overly ambitious prompts that requested features beyond what the game generation system could realistically implement. Finding the right balance between detailed instructions and feasible implementation was a key challenge.
- **Maintaining user intent:** While the DSPy optimizer helped ensure the optimized prompts stayed close to the user's original intent, there were cases where the system added elements that significantly altered the game concept. This highlights the importance of careful tuning of the optimizer parameters like temperature and batch size.
- **Genre-specific optimization:** Different game genres benefit from different types of prompt enhancements. A one-size-fits-all approach may not be optimal, suggesting the need for genre-specific optimization strategies.
- **User preference variation:** Users have different preferences for game styles and features, making it challenging to optimize prompts in a way that satisfies all users. The few-shot learning approach helps address this by providing multiple options, but further personalization may be beneficial.

7 Conclusion

This project demonstrates the effectiveness of reinforcement learning for optimizing game generation prompts, significantly improving the quality of AI-generated games. By automatically enhancing user prompts with important game elements like physics, mechanics, visuals, and audio, the system enables users to create better games without requiring prompt engineering expertise.

The key findings of this research include:

- A 252% increase in average game ratings through prompt optimization
- Identification of key prompt elements that correlate with higher game quality
- Successful adaptation of the PRewrite approach for game prompt optimization
- Effective implementation of DSPy and the MIPROv2 optimizer for prompt optimization
- Promising results from few-shot learning experiments providing multiple game options

This approach democratizes game development by removing technical barriers and enabling rapid iteration. Users can focus on their creative vision rather than the technical details of prompt engineering, leading to a more accessible and enjoyable game creation experience.

7.1 Future Work

Several directions for future work have been identified:

- **Online learning:** Continuously update the optimizer as new ratings come in, allowing the system to adapt to changing user preferences and game generation capabilities.
- **Multi-objective optimization:** Balance between ratings and other metrics like play duration, social sharing, and remixing to optimize for overall user engagement.
- **User customization:** Allow users to specify which aspects of games they want to emphasize, providing more personalized prompt optimization.

- **Genre-specific optimization:** Train specialized optimizers for different game genres, recognizing that different types of games benefit from different prompt enhancements.
- **Explainable optimization:** Provide users with explanations of how and why their prompts were optimized, helping them learn prompt engineering skills through the process.

By addressing these future directions, the system can continue to evolve and improve, further democratizing game development and enabling more users to create high-quality games through AI assistance.

8 Team Contributions

As the sole contributor to this project, I was responsible for all aspects of the work, including:

- Designing and implementing the interactive platform for AI-assisted game creation
- Developing the DSPy-based prompt optimization system
- Collecting and analyzing game data from the platform
- Training and evaluating the prompt optimization models
- Implementing the few-shot learning approach
- Writing this paper and presenting the results

Changes from Proposal The final project expanded on the original proposal in several ways. First, I implemented a DSPy-based approach for prompt optimization using the MIPROv2 optimizer, which proved highly effective at learning from game ratings. Second, I added the few-shot learning approach, which was not part of the original proposal but emerged as a valuable extension during the project. Finally, I conducted more detailed analysis of which prompt features correlate with higher game ratings, providing deeper insights into what makes effective game prompts.

References

Omar Khattab, Keshav Santhanam, Xiang Lisa Li, David Hall, Percy Liang, Christopher Potts, and Matei Zaharia. 2023a. Demonstrating the Unreasonable Effectiveness of Compositional Prompting. *arXiv preprint arXiv:2308.10248* (2023).

Omar Khattab, Arnav Singhvi, Matthew Mahoney, Matei Zaharia, and Christopher Ré. 2023b. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. *arXiv preprint arXiv:2310.03714* (2023).

Weize Kong, Spurthi Amba Hombaiah, Mingyang Zhang, Qiaozhu Mei, and Michael Bendersky. 2024. PRewrite: Prompt Rewriting with Reinforcement Learning. *arXiv preprint arXiv:2401.08189v4* (2024).

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. arXiv:2203.02155 [cs.CL]

A Additional Experiments

In addition to the main experiments described in the paper, I conducted several additional experiments to explore different aspects of prompt optimization:

A.1 Optimization Parameter Analysis

I experimented with different parameters for the DSPy MIPROv2 optimizer to find the optimal balance between prompt improvement and staying close to the user's original intent. Parameters like temperature, batch size, and number of iterations were tuned to ensure the optimized prompts maintained the user's original game concept while adding necessary details. Lower temperature values (0.3-0.5) produced more conservative optimizations, while higher values (0.7-0.9) generated more creative but sometimes divergent prompts.

A.2 Meta-Prompt Variations

I tested different meta-prompts for the rewriter LLM to understand how the instructions given to the model affect the quality of the optimized prompts. The most effective meta-prompts were those that explicitly mentioned all the important game elements (physics, mechanics, visuals, audio, controls) and emphasized the need to stay close to the user's original intent.

B Implementation Details

B.1 API Server Implementation

The prompt optimization system was deployed as an API server using Flask, allowing it to be easily integrated into the game creation platform:

```
@app.route('/optimize_prompt', methods=['POST'])
def optimize_prompt():
    """
    API endpoint to optimize a game prompt.

    Expects a JSON payload with:
    - prompt: The user prompt to optimize
    - stage: (Optional) The learning stage to use (1-7)

    Returns the original and optimized prompts, along with feature analysis.
    """
    data = request.json
    user_prompt = data.get('prompt', '')
    stage = data.get('stage', 7)  # Default to the most advanced stage

    if not user_prompt:
        return jsonify({"error": "No prompt provided"}), 400

    # Validate stage
    if stage < 1 or stage > 7:
        return jsonify({"error": "Stage must be between 1 and 7"}), 400

    # Choose the appropriate optimizer
    if stage < 7:
        # Use the staged optimizer for demonstration
        current_optimizer = staged_optimizers[stage]
        optimized_prompt = current_optimizer(user_prompt=user_prompt)
    else:
        # Use the fully trained optimizer
        optimized_prompt = optimizer(user_prompt=user_prompt)

    # Extract features from both prompts for analysis
    original_features = extract_prompt_features(user_prompt)
    optimized_features = extract_prompt_features(optimized_prompt)

    return jsonify({
```

```
        "original_prompt": user_prompt,
        "optimized_prompt": optimized_prompt,
        "stage": stage,
        "stage_description": get_stage_description(stage),
        "original_features": original_features,
        "optimized_features": optimized_features
    })
```

B.2 Feature Extraction Implementation

The feature extraction process analyzed prompts for the presence of key game elements:

```
def extract_prompt_features(prompt):
    """
    Extract features from a prompt to analyze what makes good prompts.
    """
    # Define key features we want to track
    features = {
        "3d_graphics": ["three.js", "3d", "webgl", "3d model", "3d object"],
        "camera": ["camera", "perspective", "view", "orbit", "first-person"],
        "lighting": ["light", "shadow", "ambient", "directional", "spotlight"],
        "physics": ["physics", "gravity", "collision", "rigid body", "force"],
        "scoring": ["score", "point", "reward", "achievement", "leaderboard"],
        "sound": ["sound", "audio", "music", "effect", "volume"],
        "controls": ["control", "keyboard", "mouse", "input", "joystick"]
    }

    # Check for presence of each feature
    result = {}
    prompt_lower = prompt.lower()

    for feature, keywords in features.items():
        # Count how many keywords are present
        matches = sum(1 for keyword in keywords if keyword in prompt_lower)
        result[feature] = min(1.0, matches / 2)  # Normalize to 0-1 range

    return result
```