Bulletin of the *Transilvania* University of Braşov
Series III: Mathematics and Computer Science, Vol. 4(66), No. 1 - 2024, 191-208
https://doi.org/10.31926/but.mif.2024.4.66.1.14

AUTOMATA CONCEPTS OVER GAME DESIGN PROCESS

Alexandra BĂICOIANU[1], Cosmin DOBRE[2], Mihnea LOPĂTARU[3], and Ioana Cristina PLAJER[*,4]

Abstract

Discrete states linked together by events can describe a series of classical video game scenarios. Therefore, the formal concept of automata seems appropriate for describing such games. However, as simple finite automata need to keep track of past events, they are less appropriate for complex designs. By contrast, push-down automata featuring a stack present better capabilities in this context. This paper discusses an enhanced version of the classic PAC-MAN game with push-down automata. The modular design of this game allows the extraction of discrete states, and its minimalist concept enables various approaches and new versions without sacrificing its original essence. Using push-down automata expanded the game with new features, allowing an enhanced game experience. The automaton controls all aspects of the game, ensuring a consistent gameplay. The presented project demonstrates the capabilities of push-down automata in the gaming industry and their potential in future game development projects.

2000 *Mathematics Subject Classification:* 68Q01, 68Q45, 03D05, 68N20.
Key words: game design, push-down automaton, formal methods, PAC-MAN, Unity.

1 Introduction

The field of computer science that deals with formal methods and compilers is widely recognized for its ability to establish precise data formats and programming language syntax [1]. Furthermore, different authors underline the importance

[1]Faculty of Mathematics and Informatics, *Transilvania* University of Braşov, Romania, e-mail: a.baicoianu@unitbv.ro
[2]Faculty of Mathematics and Informatics, *Transilvania* University of Braşov, Romania, e-mail: cosmin.dobre@student.unitbv.ro
[3]Faculty of Mathematics and Informatics, *Transilvania* University of Braşov, Romania, e-mail: mihnea.lopataru@student.unitbv.ro
[4*] *Corresponding author*, Faculty of Mathematics and Informatics, *Transilvania* University of Braşov, Romania, e-mail: ioana.plajer@unitbv.ro

of formal methods in software development, emphasizing their role in software specifications and describing the system properties, like functional behavior or system performance [2, 3].

Unsurprisingly, the video game industry has rapidly embraced many aspects of this field for various purposes, such as managing game behavior and AI behavior, as illustrated in literature [4, 5, 6].

Automata, such as finite state machines (FSMs) and Turing machines, play a central role in formal methods. Moreover, in the video game industry, automata can be utilized to establish a modeling interface that controls game behavior [7]. Based on these considerations, we demonstrate in this paper that push-down automata (PDA) can be successfully used to increase the features and capabilities of a well-known game to allow users an enhanced experience. The project also underlines the possibilities offered by such design patterns in the reiteration of classical games [8].

PAC-MAN is a well-known game, even for people not passionate about this field. At its core, the game is based on a rather minimalist concept, where the player must eat the dots in the level and avoid the ghosts that roam the environment. Nevertheless, the mechanics and logic behind the game reveal a complexity that enables new implementations and extensions in which the game's behavior can be automated using a PDA. Compared to a standard automaton, the additional memory of a PDA proved to be very useful in managing the player's actions.

Moreover, an overall redesign was performed, resulting in the enhanced game *PAC-MAN Survival*. This new version of the game adds randomness, a luck component, to the game, making it harder to develop winning strategies and thus creating a more modern and challenging version of this classic game.

1.1 Background considerations

Due to the popularity gained by PAC-MAN in its early years, it is not surprising that since it was released, many interpretations and remakes have been made [9], some of them using automata. However, most of these reiterations of the game feature finite state machines, using them to either handle the AI of the ghosts as in [10], or to handle the general behavior of the game [11].

There also exist implementations that use PDAs, as presented in [12, 13]. However, while they respect the basic concepts of PDAs, the implementation is slightly forced, as the stack is not essential in running the game. The results of these implementations are relatively similar to the original game and do not offer new gaming experiences, in contrast to our new design.

Our implementation aligns with recent developments in the field, as evidenced by the use of PDA in the design methodology [12, 13]. However, our approach differs notably, by the addition of a new state, *walking poisoned pac*, which enhances the game's difficulty and makes our implementation distinct. On the other hand, alternative approaches have been taken in other projects, such as using finite state machines [10] and applying decision theory for player analysis [14].

An investigation was conducted to determine the feasibility of utilizing a PDA in implementing a video game. The research reviewed relevant sources, such as [11, 13], which confirmed the potential of the initial concept. The process's subsequent phase involved experimentation, testing different implementations and making modifications or starting anew as necessary to accommodate the selected case.

2 Problem formulation

2.1 Aim of the project

The project presented in this paper aims to redesign the classical PAC-MAN game in a new and challenging way through a PDA and thus to underline the relevance of theoretical formal methods concepts in game development.

As stated in section 1.1 of this paper, many iterations of PAC-MAN employ automata, even PDAs. However, the ones that use it do not provide a meaningful role for the PDA's stack. For example, the implementations featured in [12, 13] load all the food items at the beginning of the game, only performing POP operations during the game's execution. While the approaches mentioned above may not be inherently incorrect from a theoretical standpoint, they fail to utilize the available memory adequately as they attempt to make minimal alterations to the original PAC-MAN game. Our project's objective is to exploit the potential of the additional memory provided by a PDA to enhance the game's interactivity and complexity.

2.2 Original game description

PAC-MAN is an action, maze traveling game in which the player has to eat all the pellets/dots in the level to finish it while avoiding the four colored ghosts Blinky (red), Pinky (pink), Inky (cyan), and Clyde (orange) which roam through the maze. Some of the pellets in the maze called *Power Pellets* are larger, and eating them, allows the player, for a specific time to increase the speed and eat the ghosts, thus gaining more points.

The original game has a relatively linear play style, and, while it may seem minimalist compared to modern games, it has many subtleties [14]. The game outcome depends on the choices made by the player, but also on the randomness of the game itself, and can therefore be classified as a stochastic outcome game [15]. A primary characteristic of such games is the dynamic change of the environment based on the player's decisions.

The game features a series of agents, the four ghosts, each with its unique behavior and walking pattern. There is a constant interaction between the player and the ghosts, creating the concept of *chasing and fleeing*, one of the game's main elements. The game environment changes when the player eats a *power-up*, becoming invulnerable to the ghost attack. The ghosts no longer chase the PAC-MAN but run away, and the player can eat the ghosts himself.

It is worth noting that the game's layout remains constant, and the ghosts' behavior can be learned over time, enabling the development of winning strategies. By following these strategies, players can successfully complete each level. These strategies typically take the form of specific walking patterns, with the *Apple pattern* [16] being one of the most widely used.

2.3 Enhanced game proposal

Our iteration of PAC-MAN aims to retain the original game's essence while providing a new, more challenging experience. The core aspects of the game, such as decision making and the *chase and flee* concept, are maintained. However, the randomness present in the original version is increased, requiring the player to constantly adapt to a changing and unpredictable environment.

One difference between the original game and our version is that the ghosts no longer have a predefined walking pattern that can be learned. In our iteration, the ghosts scatter randomly throughout the maze, each time choosing a different path, making it impossible to learn their behavior. This forces the player to be observant and carefully calculate the moves to avoid being caught by one of the ghosts. If the player gets too close to one of the ghosts, that particular agent will enter chase mode, following the player until either the player is caught and killed or the player escapes its attack range.

The *survival* aspect of our game also adds another random factor, as food items have a chance of being either a power-up or power-down or a normal pellet, leading to one of the states:

- *poisoned-pac* (25% probability): the player's speed is decreased, and he can no longer eat the dots, making him susceptible to the ghost attacks and also to losing by running out of food;
- *super-pac* (15% probability): the player's speed is increased, and he becomes invulnerable, being able to attack the ghosts and eat them;
- *normal-state* (60% probability): the player is in the usual state.

Thus the new game introduces an added layer of complexity through the unknown abilities a player may acquire. Additionally, it determines a constant pursuit of food. A previously eaten item is removed from the automaton's stack at a specific time interval, the necessary modifications are applied, and the process repeats. If an attempt is made to remove an item from an empty stack, the player loses the game. As a result, the player must always have at least one item in the stack, leading to critical decision making and risk-taking to ensure the PAC-MAN's survival.

Similar to the original PAC-MAN game, the primary objective for the player is to gather pellets generated in the level while evading ghosts. However, players must continually consume food in our modified version to avoid starvation. After a specific period, the stack has POPped a pellet and changes the player state

according to the type of the POPped pellet. If the player does not eat enough, he can reach the point where the PAC-MAN's *stomach* is empty, resulting in the loss of the game.

Another notable distinction is the inclusion of a *poisoned pac* state, which slows the player down and disables the ability to consume food. In addition to the two standard states of normal and super-pac, this further complicates the gameplay, adding a layer of challenge for the player to navigate. This use case highlights the importance of quick reflexes, strategic thinking, and precise movement in navigating the game's challenges while working towards completing the level.

The role of the PDA is to efficiently implement the new concept of *survival*, in which the stack's PUSH operation adds a food item whenever the player consumes one, and the POP operation removes the item from PAC-MAN's *stomach* and apply the resultant modifiers (power-up or power-down), as well as to enable the modeling and management of the overall game behavior, by creating a general mapping of the model onto specific states.

3 Materials and methods

3.1 Push-down automata basics

A push-down automaton is a finite state machine augmented with additional memory in the form of a stack [17]. This stack allows the PDA to recognize a larger class of languages, namely the context-free ones. A PDA is a finite state control in which the transitions are determined by the current state and the current element in the input, as well as by the content of the stack. The head of the stack is analyzed for transitioning, then the element is popped and processing continues. The transition between states is defined through the use of a transition function, which models the PDA's behavior. If the PDA encounters a nonvalid input, a phenomenon named deadlock occurs. This is why paying attention when designing an automaton is important, as unwanted deadlock effects can lead to unpredictable behavior.

Since a PDA is an extension of a FSM, the main characteristics of a FSM still apply, as they are presented in [3, 17]. The difference is in the additional definitions needed to integrate the stack, including the stack's alphabet and the PUSH/POP operations. A scheme of a PDA is represented in Figure 1.
Formally, a push-down automaton M is defined by a structure:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

where Q is the finite set of states of the PDA, in which $q_0 \in Q$ is the initial state, Σ is the set of input symbols, Γ represents the alphabet of the stack, with $Z_0 \in \Gamma$ the initial symbol in the stack, δ gives the mathematical definition of the transition function and $F \subseteq Q$ a set of final/accepting states.

The transition function δ defines the behavior of the PDA for a given triplet (q, a, X), where q is the current state, a is the input symbol, and X the symbol

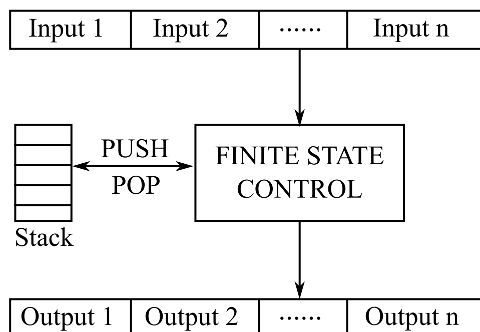

Figure 1: Push-down automaton.

on top of the stack. For a given transition $(q, a, X) \rightarrow (p, Y)$ (Figure 2), the automaton performs the following actions:

- scans the input symbol a (which may be null);
- shifts from state q to state p;
- POPs the symbol X, located at the top of the stack, and PUSHes the symbol Y onto the stack. Each symbol may be null, in which case the corresponding action (PUSH/POP) is not performed.

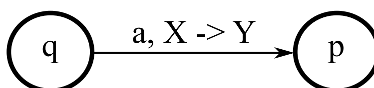

Figure 2: Transition function for PDA.

The general definition of a PDA is inherently non-deterministic. However, considering the use-case for our study, a deterministic approach was necessary to ensure the game's predictable and consistent behavior during each run.

3.2 PDA representation for PAC-MAN extension

This section focuses on the development of the PDA in the context of the game considered. Therefore, it is relevant to follow the generalization of the initial game, the description of the newly defined PDA, and the deployment of the extended game.

3.2.1 States description

Analyzing the original game, several game states are essential for a logical sequence of events. These states can be mapped out and implemented into an automaton, allowing for smooth transitions based on what is happening in the game. The main states detected in the authentic game are:

- *a loading state*
- *a normal walking PAC-MAN state*
- *a super pac walking state*

One of the significant factors affecting PAC-MAN players' performance is strategy representation. Different techniques are used to represent players' behavior, and one of the successful techniques is using the automata-based model to represent and control participated agents. Therefore, the specified states can be implemented using a finite state machine, resulting in a functioning version of PAC-MAN. However, another crucial game aspect must also be considered which is *eating the dots*. This aspect of the game would need to be implemented as well in order to replicate the gameplay of the original PAC-MAN fully.

To implement such a feature, additional memory is required to check what the last eaten dot was and change the game state if a power-up dot was eaten. One solution to this issue is to use a PDA instead of an FSM, as its stack provides the necessary memory. Additionally, the stack's Last In First Out (LIFO) property allows for quick access to the last item that the player ate, as it will be at the top of the stack.

It is obvious that PDAs can easily be used to implement the PAC-MAN game, as its design allows for a series of states that can be modeled and added to the automaton. However, even though the stack of the PDA is useful for implementing the original game, it could easily be replaced by another data structure.

Additionally, the specific PUSH and POP operations are not evenly distributed throughout the game but instead concentrated in the beginning and end. Furthermore, the POP operations may not be necessary, as checking the top of the stack would be sufficient instead of removing the element.

The PDA's stack is no longer optional for the proposed PAC-MAN version, as we have distributed the PUSH and POP operations throughout the game. This data structure provides the necessary supplementary memory and ensures that these operations are completed in $O(1)$ complexity time, resulting in a faster running application.

Overall, the use of a PDA in the implementation of the PAC-MAN game can provide a flexible and efficient way to model the game's states and transitions and provide a way to keep track of critical game-related scenarios.

3.2.2 Formal PDA description

The purpose of the PDA is to manage the operation of the whole game perspective. The entire flow is illustrated in Figure 3. When creating the automaton, defining the possible game states and specifying the transitions between them was a crucial step. The diagram shows the blueprint behind the implementation of the PDA.

Further on, we will comment on the role and meaning of each state introduced:

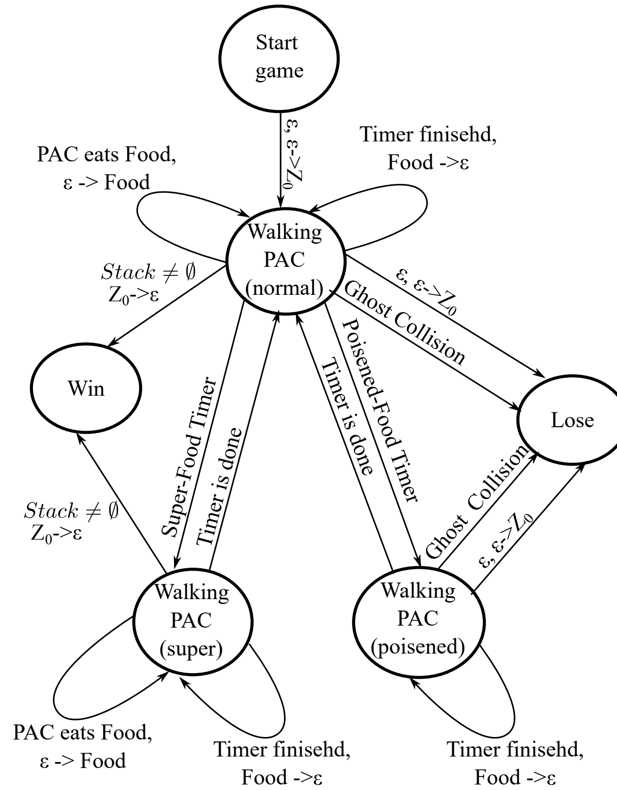

Figure 3: PDA Diagram.

- *Start game* - the necessary initialization is done, and assets are loaded, player navigation is disabled;
- *Walking PAC (normal)* - the PAC-MAN is in its normal state, no modifiers are applied and the player is able to navigate around the maze;
- *Walking PAC (super)* - the PAC-MAN has a series of modifiers applied in order to give the player an advantage: it becomes invulnerable, the speed is increased alongside the dimensions of the PAC-MAN in order to facilitate dots consumption;
- *Walking PAC (poisoned)* - the PAC-MAN has a series of modifiers applied in order to give the play a disadvantage: its size is decreased, making it harder to eat the dots, and the speed also receives a lower value making it an easier target for the ghosts and the player can no longer eat the dots, making it susceptible to loosing from running out of food;
- *Win* - the game is stopped, and a winning message is displayed alongside a series of options from which the player can choose;
- *Lose* - the game is stopped, and a loss message is displayed alongside a series of options from which the player can choose, such as retrying the level.

As the PDA diagram highlights, the transition from one state to another is done if the necessary criteria are met. It is essential to mention that while from a given state, we can transition to multiple ones, the deterministic characteristic of the automaton is not invalidated, as the requirements for each transition are different.

Another important aspect illustrated in the above diagram is the continuous flow of PUSH and POP operations from the stack, which is constantly modified during the running of the states, highlighting the idea of spreading these operations across the entire game.

Considering all these aspects, the proposed PDA aligns with the mathematical model outlined in the previous section 3.1.

3.2.3 Game development

The primary objective during the implementation of the PDA for our redesign was to create a generalized version that could be adapted to meet the unique requirements of any similar project. A graphical representation of the application flow can be seen in Figure 4, which illustrates the connections between the entities implemented in our game. From a technical point of view, it was essential

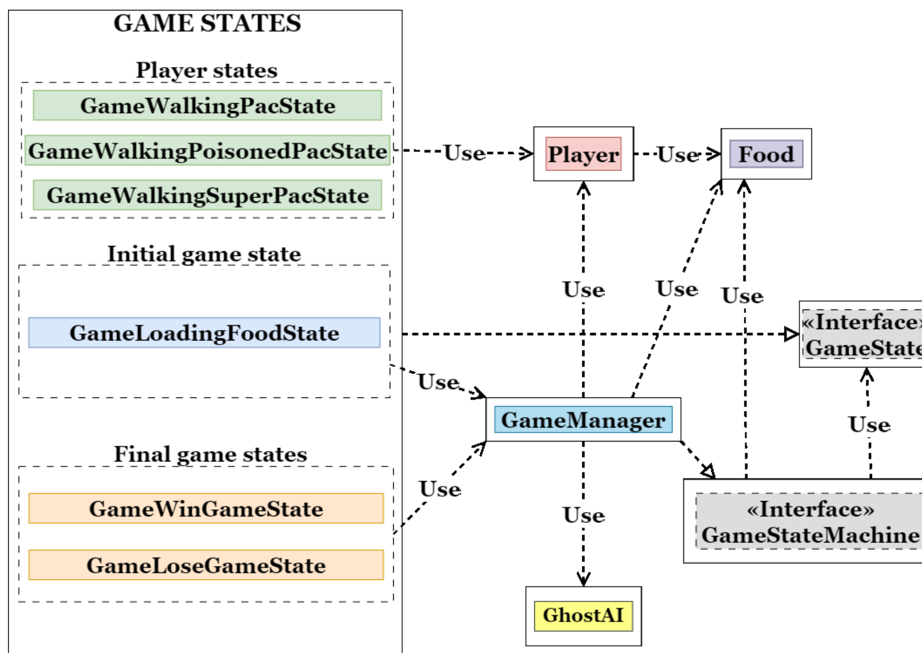

Figure 4: Architectural UML.

to develop an implementation that was as generic as possible, extensible, scalable and, very important, to follow the principles of a clean code. The game state architecture illustrated in Figure 4 encompasses all the dynamic phases the game undergoes: Player States, Initial Game State, and Final Game States. These states play a crucial role in guiding the player character's interactions and reactions to in-game events.

The Game States

Within the **Player States** section, a suite of classes orchestrates the various conditions the character undergoes in response to consuming different types of pellets.

1. *GameWalkingPacState* - is the default state of the character when navigating the game environment without any enhancements. In this state, the character maintains standard attributes and capabilities.

2. *GameWalkingPoisonedPacState* - upon consuming a *poisoned* pellet, the character transitions into this state that imposes a reduction in the character's size and a decrement in movement speed, simulating the debilitating effect of the poison.

3. *GameWalkingSuperPacState* - conversely, consuming a *super-power* pellet ushers the character into this state, which bestows augmented attributes such as increased size and movement speed. Additionally, this state empowers the character to consume ghosts, thereby gathering extra points for the player, enhancing the interactive game experience.

The **Initial Game State** section encompasses the **GameLoadingFoodState**. This state employs a specialized Grid Mesh, a native feature of the Unity game engine, which facilitates the randomized deployment of pellets throughout the game map for dynamic gameplay initiation.

The **Final Game State** contains two pivotal classes: **GameWinGameState** and **GameLoseGameState**. The first one is triggered upon the successful collection of all pellets, culminating in a display of the player's score and duration of play. The second one is initiated if the player's character is captured by a ghost or if the supply of consumable pellets depletes, resulting in a display of the final score achieved.

The Game Manager

This component operates as the core conduit for controlling the game flow, interfacing seamlessly with the player state classes as well as other integral game components, such as Player, Food, and Ghost AI. It orchestrates the transitions between states, ensuring a consistent gameplay experience.

In the architecture, we introduced two interfaces that highlight the general in-game behavior for the most important instances in the game. The interfaces conform to the structure and mathematical model of the PDA.

The GameState Interface

This interface has the role to establish a general framework to which all states must adhere, in order to ensure the automaton's proper functioning. Therefore, when creating a new state, it must implement the following methods:

1. *EnterState()* - performs the necessary initialization to enable the state to start and run smoothly
2. *UpdateState()* - implements the execution of the state, covering the necessary actions to be taken while the state is active
3. *ExitState()* - ensures that when transitioning to a different state, the current state is properly closed without leaving any memory leaks or lingering modifications that could impact the next scene. This helps to maintain stability and prevents unintended effects.

By mandating that all states implement the required methods, the integration of new states into the automaton becomes smooth and error-free, leading to better game operation. This promotes code reusability and simplifies maintenance. Furthermore, if a state needs additional methods, they can be defined and customized to suit its specific needs.

The PDA Interface

In a manner similar to the GameState Interface, the **PDA Interface** facilitates the creation of a generic model that aligns with the definitions of a PDA. This model can then be modified to suit a specific purpose. In our scenario, the PDA Interface encompasses the following methods that must be implemented by any class seeking to fulfill the role of the automaton:

- *SwitchState()* - performs the transition from one state to another
- *PopFromStack()* - POPs an element from the automaton's stack
- *PushToStack()* - PUSHes an element in the automaton's stack

Following the implementation of the interface, a class was established to manage the overall behavior of the game. This class, which is derived from the PDA Interface, implements its specific methods and subsequently declares supplementary functions and members to guarantee the proper operation of the game. The class, referred to as *GameManager*, implements the mandatory functions inherited from the interface and includes additional functions and members to guarantee a successful game run, such as holding all possible game states.

The suggested implementation demonstrates the versatility and feasibility of using a PDA in developing video games. We enhanced the classic PAC-MAN by implementing the automaton, resulting in an automated game with a dynamic and fluid gaming experience directly impacted by the player's choices. Ensuring code reusability is one of the most important aspects when developing a game. This technique is still used and assures that once a feature is implemented, it can be reused and perhaps adapted to serve a similar role in a completely different game. Our project considers this aspect, offering a general interface for the automaton, which can be adapted and specialized for a specific game. Furthermore, the way in which the states of the game are designed assures that, if needed, a new state can be easily added without altering the functionality of the previous ones.

In addition to traditional gameplay, we have enriched PAC-MAN with a *Survival* mode, incorporating a new layer of complexity that demands players to adapt to ever-evolving scenarios. Central to this mode is the introduction of a randomly generated environment, which significantly elevates the unpredictability and challenge. Specifically, the map is designed with a series of predefined spawn locations for food pellets. At the start of each game session, a random selection process activates, choosing a varying number of points from these predefined positions as the initial spawn points for the food pellets. This method ensures that each game presents a unique challenge, intertwining elements of luck and strategy due to the dynamic and unpredictable nature of the environment. Consequently, developing a consistent winning strategy becomes more challenging, as the game environment differs with every playthrough.

Despite increasing the original game's difficulty, we made it easy for new players to learn the basics and mechanics of the game through standard keyboard mapping and intuitive level designs that guide the player through the map. This makes it simple for new players to understand the main aspects of the game, while the challenge lies in applying these concepts to achieve the desired outcome of winning.

One final adaptation is the behavior of the ghosts, PAC-MAN's enemies. In the original game, each ghost had a distinct walking pattern, making their behavior predictable. In our variant, the ghosts roam freely throughout the maze and will chase the player if they come within a certain range of the ghost. This ensures that the core design concept of the original PAC-MAN, the chase and flee dynamic, is still present in our version.

3.2.4 Performance assessment

Determining the performance profiler of an algorithm is part of its subsequent testing and aims to determine the precise order of complexity of the algorithm's execution time. In addition, the resulting information is usually used to validate or invalidate, respectively, to refine the results of the a priori estimation.

Profiling is the process of measuring and analyzing the performance of a game, including its runtime behavior and resource usage. This information provides insights into areas of the game that may be causing performance issues, such as low frame rates, slow loading time, or excessive memory usage. In addition, the profiling data can be used to identify and address bottlenecks in the game's performance, such as areas of code that are taking longer to execute than expected or objects that are consuming more memory than necessary. This information can then be used to optimize the game's performance and improve the overall player experience.

A combination of quantitative and qualitative measures was used in evaluating the performance of the proposed PAC-MAN game implementation. On the quantitative side, metrics such as average completion time, survival rate, and score were investigated. These measures allowed us to assess the game's difficulty and track players' progress over time. The analysis has focused on the perfor-

mance metrics of critical game mechanics, specifically the stack overview, death text, and current player state given by their crucial role in the game's mechanics. By examining the performance, we aim to understand better how they impact the overall game experience and identify any areas for potential optimization or improvement.

The profiling process was performed using Unity Profiler (a built-in application) to examine the impact of PAC-MAN state transitions on CPU and Memory performance. Unity Profiler is part of the Unity Editor, and it comes with a low-level native plug-in Profiler API [18].

As depicted in Figure 5, the results indicate that altering the PAC-MAN state does not significantly affect CPU usage. The profile's CPU section displays orange lines corresponding to script loading, while purple lines indicate other components, such as the Unity Profiler and Unity Editor, which consume a substantial CPU side. Examining the cyan graph at the bottom of the diagram, which illustrates the rendering of the scene, it can be observed that the game maintains a consistent state within the boundaries delineated by the green boxes. These encapsulated segments signify periods of stability where no state transition occurs. In contrast, the transition from one state to another is represented by a noticeable fluctuation, which is depicted by the pink frame. This spike symbolizes the dynamism of the game's state changes, marking the points of departure from one state to an ensuing state, reflective of game events or player interactions that necessitate such transitions.

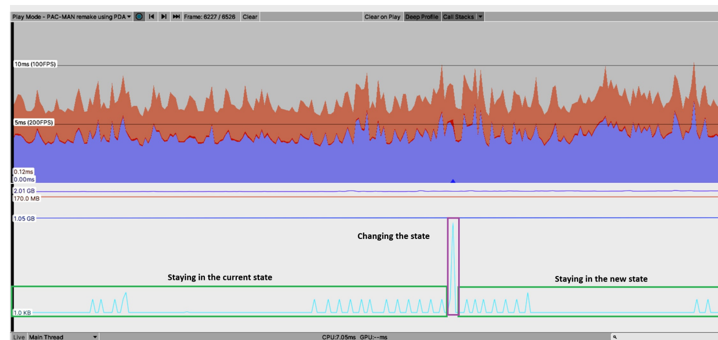

Figure 5: Performance when staying in the same state and when changing the state.

However, the key distinction between the two states is in the time required for state transition, see Figure 6. Specifically, the state transition process, which includes updating the stack overview and death text, takes $1ms$, representing 10% of all script loads. Conversely, remaining in the same state requires a mere $0.12ms$, accounting for only 1.7% of all script loads, which includes updating the stack overview and death text. In terms of memory, the state transition process is represented by a big blue line in the Memory section, indicating $5.5KB$ of memory usage, whereas the memory usage for remaining in the same state is $0.6KB$ and is represented by constant blue lines. The blue lines' minor peaks signify the

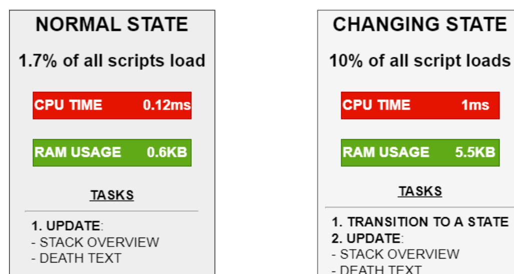

Figure 6: Implementation performance details.

occurrence of particle-camera collisions.

On the qualitative side, we conducted surveys and user testing with a sample group of ≈ 100 players. Participants were asked to provide feedback on their overall experience, including elements such as gameplay and graphics. Additionally, we collected data on player satisfaction with the game's level of challenge and the implementation of PDA for player states.

The results of our performance evaluation were overwhelmingly positive, with the majority of players reporting a satisfying and challenging gaming experience. PDAs were deemed successful in adding an extra layer of complexity to the game, as evidenced by the increased average completion time and decreased survival rate.

In conclusion, our performance assessment has verified the viability of incorporating PDAs for player states in PAC-MAN games and has revealed important avenues for future development. This study offers a valuable resource for game developers who are looking to adopt PDAs in their game designs.

4 Future developments

The implementation of *PAC-MAN Survival* using push-down automata (PDA) has demonstrated the potential of integrating formal methods concepts into game design, offering a new perspective on enhancing classic games. This approach has opened new fronts for future developments, not only for the game itself but also for the application of these concepts in other games. Firstly the proposed enhanced version can further be developed. Secondly the potential of PDAs can be exploited in other types of games.

4.1 Improvements to PAC-MAN survival

One potential area for improvement in *PAC-MAN Survival* is the implementation of dynamic difficulty adjustment (DDA). This involves the game automatically adjusting its difficulty level in real-time based on the player's performance. For instance, the game could monitor the player's success rate, adjusting the speed or intelligence of ghosts, or the effects of poisoned pellets, to maintain a challenging yet achievable gameplay experience. Incorporating DDA could enhance player

engagement and ensure a balanced challenge for players of all skill levels.

Furthermore, adding a multiplayer mode could significantly expand the game's appeal, allowing players to compete or cooperate within the same game environment. This could involve one player controlling PAC-MAN while others control the ghosts, or multiple players competing as different PAC-MAN characters trying to survive the longest in the maze. Multiplayer mode would require careful balancing to ensure fairness and competitiveness, potentially involving different PDA configurations for different roles.

Another potential development concerns customized mazes and mod Support. Allowing players to create and share their mazes could vastly increase the game's replayability and community engagement. This could be facilitated through in-game maze design tools or by supporting mods that let players import custom assets and game logic. This feature could also be enhanced by integrating a stack-based logic for defining custom game rules or behaviors within the mazes, leveraging the PDA's capabilities for more creative gameplay designs.

4.2 Application to Other games

The concept of using PDAs can be extended to role-playing games (RPGs) to manage complex character states and inventory systems. For example, a PDA could track the effects of consumables or spells on a character, where each item or spell has unique effects that can stack or counteract others. This approach could lead to a more strategic gameplay, where players must carefully consider the order of actions based on the current state of their character's abilities and status effects.

In real-time strategy games, PDAs could be used to manage the states of units or buildings, allowing for more complex interactions and strategies. For instance, a PDA could track the construction and upgrade paths of buildings, where certain upgrades or buildings unlock new abilities or units. This could introduce a deeper layer of strategy, as players would need to plan and execute their development paths strategically to counter their opponents effectively.

Educational games could benefit from PDAs by dynamically adjusting the difficulty of puzzles or challenges based on the player's performance. For example, in a game designed to teach programming concepts, a PDA could track the player's progress and understanding, offering hints or adjusting the complexity of challenges to suit the player's learning pace. This personalized approach could make educational games more effective and engaging.

5 Conclusions

Push-down automata can be successfully used in game development to simplify the processes that control the overall behavior of the game, including the implementation chosen in this study. Furthermore, their versatility has been demonstrated, by using them to revitalize a classic game such as PAC-MAN.

This study aims to use PDAs to generate a more challenging version of the game. The paper focused on enhancements and the game development phase. In addition to the utilization of PDAs, implementing this updated version of PAC-MAN was facilitated by using the Unity engine for its development. Unity streamlined the design of the GUI and graphics and accelerated the development process through the availability of built-in tools and scripts for everyday game actions, such as the character's movement through the utilization of a character controller integrated within Unity. Finally, the new game version was evaluated quantitatively (e.g ., average solution time or mean average core) and qualitatively (e.g ., user satisfaction and feedback).

In the final analysis, the utilization of a push-down automaton in the design of *PAC-MAN Survival* has evidenced its capability in managing player states and maintaining a record of the game's objectives. The PDAs' structure and adaptable nature enabled a more intricate and dynamic gaming experience with sophisticated mechanics. This method has proven to be a viable solution for the management of game logic and could open up opportunities for further advancements in the field. Integrating a push-down automaton in PAC-MAN exemplifies the potential of this technology in developing complex and dynamic games.

Compared to other approaches, such as finite state machines or decision theory, the push-down automata approach demonstrates a significant contribution to the field, highlighting its potential for future development and innovation in game design. These results have important implications for game designers, developers, and researchers and contribute to the field's evolution.

References

[1] Shallit, J., *A second course in formal languages and automata theory*, Cambridge University, 2009.

[2] Clarke, E.M. and Wing, J.M., *Formal methods: state of the art and future directions*, ACM Computing Surveys (CSUR), **28** (1996), no. 4, 626-643.

[3] Wang, J. and Tepfenhart, W., *Formal methods in computer science*, CRC Press, 2019.

[4] Ali, K.F., Kalyan, V. and Kumar, K.A., *Design and implementation of Ludo game using automata theory*, in Innovations in Power and Advanced Computing Technologies (i-PACT) vol. 1 (2019), 1-6.

[5] Kowalski, J. and Szykuła, M., *Game description language compiler construction*, in AI 2013: Advances in Artificial Intelligence: 26th Australasian Joint Conference, Dunedin, New Zealand, December 1-6, 2013. Proceedings 26, Springer, 234–245, 2013.

[6] Ta, v. and Xu, L., *Case study: designing and developing a game prototype*, Journal of Education and Social Development (2019), 35-40.

[7] Bourg, D.M. and Seemann, G., *AI for game developers*, O'Reilly Media, Inc., 2004.

[8] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Professional Computing Series, Pearson Education, 1994.

[9] Wardrip-Fruin, N., *How Pac-Man Eats*, MIT Press, 2020.

[10] Bairagi, S., *Emulating Pac-Man using finite state machines*, Dipl. Thesis, Dept. of Computer Science, Indian Institute of Information Technology, 2020.

[11] Dauber, K. and Devlin, R., *Pac-Man EXTREME!!!!!*, Massachusetts Institute of Technology, 2017.

[12] Vayadande, K., More, H., More, O., Mulay, S., Pathak, A. and Talnikar, V., *Pac Man: game development using PDA and OOP*. International Research Journal of Engineering and Technology (IRJET) **9** (2022), no. 1, 959-961.

[13] Yunita, A., Fadillah, R.Z. Darmawan, M.R. and Putra, A.D.G., *Re-designing the PACMAN game using push down automata*, 4th Asia Pacific Conference on Contemporary Research, 50-56, 2018.

[14] Cowley, B.U., Charles, D.K., Black, M.M. and Hickey, R.J., *Using decision theory for player analysis in Pacman*, in SAB'06 Workshop on adaptive approaches for optimizing player Satisfaction in computer and physical games, At: Rome, 41-50, 2006.

[15] Solan, E. and Vieille, N., *Stochastic Games*, Proceedings of the National Academy of Sciences of the United States of America (PNAS) **112** (2015) no. 45, 13743–13746.

[16] Mulliken, J.D., *Pac-Man. The ultimate key to winning*, Running Press Philadelphia, Pennsylvania, 1982.

[17] Linz, P. and Rodger, S.H., *An introduction to formal languages and automata*, Seventh edition, Jones & Bartlett Learning, 2022.

[18] *Unity Technologies*, Unity User Manual. Profiler overview, 2023.