Design and Implementation of a 2D Game Engine: Algorithmic Approaches and Performance Optimization

Maria Pasca
Computer Science Department
Technical University of Cluj-Napoca
Str. Memorandumului 28
mariav.pasca@gmail.com
Constantin Nandra
Computer Science Department
Technical University of Cluj-Napoca
Str. Memorandumului 28
constantin.nandra@cs.utcluj.ro

ABSTRACT
This paper describes a solution for generating pseudo-random mazes to act as environments for a puzzle-type game that requires the player to make decisions while being chased by one or more opponents. We focused on developing a flexible and scalable solution, aiming to minimize the time required for the generation process. Throughout the paper, we will describe the inner workings of the proposed solution, while focusing on the optimization of the maze generation and path finding aspects. We also approach the problem from the perspective of path finding, which is relevant for the implementation of the intelligent agent chasing the player. Finally, we present the findings of a set of tests done on the described solution, which was utilized in the implementation of a maze-chase game, featuring NPC navigation through the generated maze. These include both user and system performance tests, as well as a user satisfaction survey.

Author Keywords
maze generation algorithms; dynamic path computing; game development

DOI: 10.37789/ icusi.2024.11

INTRODUCTION

Ever since the beginning of the modern gaming industry in the early 1970s, games have encapsulated significant advance- ments in various aspects of technology, such as 3D rendering or artificial intelligence. Those left an impact not only on the area of computer science, but, as stated by Jakobsson and Carney [1] also on fields such as archaeology, psychology or medicine, where various tools have been developed to help facilitate processes required in that line of work, like 3D mod- elling tools for reconstruction or visualization.

Generally, games are one of the most demanding pieces of software, both in terms of complexity and of computing re- sources employed. Therefore, there has always been a need to improve the efficiency of various parts of the game: from rendering to decision making and content generation.

Conceptually, the research and development work described in this paper started from the idea of coming up with a time and resource efficient pseudo-random maze generation algorithm. This was to be employed to generate content in a *maze-chase* game engine, whose main elements are also briefly described in the present paper. The game engine features several state machines, used to orchestrate the application loop, as well as the player and non-player character (NPC). For controlling the NPC, the described solution relies on an accurate and effective path finding algorithm, which was chosen based on the partic- ularities of the selected maze generation algorithm. While we briefly describe the workings of the game engine, throughout this paper we will mainly focus on the maze generation and path finding components.

In regard to maze generation, the approach commonly adopted is to use graph theory algorithms for computing Minimum-Spanning Trees [2], [3]. This way it chooses the shortest number of edges needed for connecting all nodes in a graph, and, when applied to maze generation, it would translate into a result that has no cycles and where all nodes are within reach from each other.

Although it is a good generation method, depending on the strategies that the algorithms are based upon, the difficulty of solving the maze ranges from easy for a human agent and rather difficult for an intelligent agent to easy for an intelligent agent, but difficult for a human one [4], [3]. Because of this, one of the main issues aimed to be solved is balancing the difficulty for all agents involved in solving the maze.

Another issue followed is the scalability potential. As some traditional algorithms used in maze generation were shown to have an exponential time complexity, another question raised that seeks to be answered is if there is a possibility of find- ing an approach in which the dimensions of the maze will not dramatically affect the time performance of the creation process.

The main contribution of our work was to experiment and integrate various efficient methods of maze generation, path finding, and rendering to create the main components of the game engine described above. We focused on improving the maze generation and path finding aspects of the game engine. We validated its design by creating a *maze-chase* game and evaluating the game in terms of usage of computing resources and player satisfaction.

In the following sections, the paper will provide an analysis on works and solutions related to various aspects of the project, an overview of the implementation, the experimental and

performance testing results and final conclusions of the work.

RELATED WORKS

Maze generation

In a 2015 article published by Kozlova et al. [4], three algorithms used for maze generation are described and compared based on the visual aspect and the average path lengths of the results. By comparing Prim, Depth-First Search (DFS) and Recursive Division (RD) they came to the conclusion that, in terms of maze complexity and aspect, those generated by Prim's algorithm are defined by recurring short corridors and those generated through RD have longer straight passages, which makes them easier to solve by a user with a top down view over the mazes. Meanwhile, those generated by using DFS also have longer passages, but because of the random nature of choosing the expansion direction, they are also more sinuous which makes it harder to visualise the path to the exit of the maze. The study's concluded that the use of DFS algorithm is advised for maze generation as the result is de- fined by its long and winding passages, which makes it more challenging to solve by a human agent.

Gabrovšek [3] comes with an extension of the previous work. This analyzes and compares three pairs of algorithms used for MST computation, each pair based on a different strategy. These are *Wilson* and *Aldous Broder*, both using the *Random Walk* strategy, *Recursive Backtracking* and *Hunt and Kill*, both based on DFS, and *Prim* and *Kruskal*, both greedy algorithms. These six algorithms have been examined on three different grounds: running time, average number of intersections and dead-ends, and agent performance.

The study has shown that, experimentally in terms of time needed for generating a 100x100 maze, *Recursive Backtrack- ing* was the fastest algorithm, with a time of around 0.1 sec- onds, followed by *Prim* and *Aldous Broder*, with times of around 0.25 seconds. Meanwhile, on the opposite side of the spectrum, *Kruskal* and *Wilson* are clearly the slowest ones, with execution times that grew exponentially.

Taking into consideration the average number of intersections that each resulted maze contains, by comparing them, *Prim* and *Kruskal* take the first two places, followed relatively close by *Wilson* and *Aldous Broder*, as the last two places are taken by *Hunt and Kill* and *Recursive Backtrack*. Analysing them based on their strategies, *Hunt and Kill* and *Recursive Back- track*, both being expanded using the DFS approach, their tendencies are to create mazes with longer passages—thus not many intersections—than the other four. This can also be seen in the table at Table 1, where n_i represents the average number of intersections in a generated 100x100 maze.

Another aspect featured in Figure 1 is the average number of dead-ends—noted as n_{de}—for each algorithm used for maze generation. The result is similar to the previously analyzed av- erage number of intersections. This again can be placed upon the algorithms' strategies, where *Hunt and Kill* and *Recursive Backtrack* being based on DFS have a bias of creating mazes with longer passages and, by extension, fewer and deeper dead-end corridors.

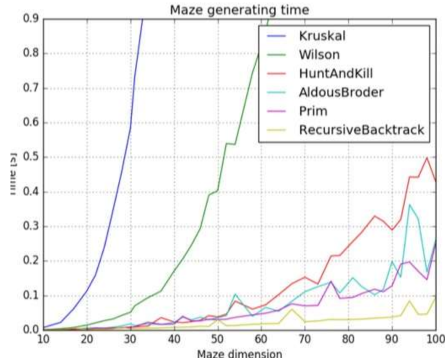


Figure 1: Running times for the algorithms with respect to the maze dimensions [3]

algorithm	n_i	n_{de}	rank
Prim	2946	3559	1
Kruskal	2654	3058	2
Aldous-Broder	2577	2933	3
Wilson	2576	2932	4
Hunt and Kill	920	939	5
Recursive Backtrack	869	898	6

Table 1: The average number of intersections and dead ends for each maze generation algorithm [3]

On the other hand, comparing the strategies on the cri- teria of agent performance, the conclusions reached are that the ones based on *Random Walk*—*Wilson* and *Aldous Broder*—achieved the best results as it is hypothesised that the agents had a much difficult time navigating their resulted mazes because of their unbiased pathways. These are followed by *Prim* and *Kruskal*—both using greedy approaches—that, even though after they were adapted for pseudo-random maze generation, their passages and dead ends are not as evenly distributed as in the previous examples, which makes them easier to solve by an intelligent agent.

The strategy that proved to create the easiest mazes to solve are the ones based on DFS—*Recursive Backtrack* and *Hunt and Kill*. For an uninformed intelligent agent, such as the ones briefly described in this paper, the larger number of intersections means a larger number of decisions that have to be made to continue the navigation.

The work of Dagaev et al. [5], captures a more diverse se- lection of maze generation algorithms besides the ones tradi- tionally used for MST computation. Among these, one of the algorithms mentioned—*Eller*—is characterised by the authors as a special one as it doesn't depend on having the graph read- ily available, but rather on generating it row by row. Because of this, the complexity of the algorithm is linear and easily scalable, and, according to Jamis Buck [6], he describes it as *'striking a nice balance between "long and winding" and "lots of cul-de-sacs".'*, which increases the difficulty to navigate for both the human and the intelligent agent.

Path finding

Beside the maze generation algorithms, Gabrovšek[3] also presents an evaluation on the efficiency and accuracy of four intelligent agents using different path finding strategies. These are Random Walk, Breadth-First Search (BFS), Depth-First Search (DFS) and Heuristic Depth-First Search (HDFS)—it uses Manhattan Distance as the heuristic of choice. Despite them being used as a way to further analyze the maze gen- eration algorithms, it also uncovers some details about their efficiency on various measured criterias. These are the average number of steps it takes to find the optimal path, the average number of visited intersections and the average number of visited dead-ends on generated mazes of 100x100 cells.

According to the average number of steps needed for finding the optimal path, the agent using random walk is the most inefficient one, situated at a very large margin from the second most inefficient one. On the other hand, the agent using BFS proved to be the most efficient, followed relatively close by DFS. When comparing the agents based on the average number of visited intersections, the one using random walk again proves to be the most inefficient, in contrast to the BFS agent who had the lowest numbers of visited intersections. The BFS agent is followed by DFS again relatively close, while Heuristic DFS is in third place.

As for the average number of visited dead-ends, the results are similar to the previous ones in the sense that Random Walk proved to be yet again the most inefficient agent for path finding. In contrast, the BFS agent is once again proving to be the most efficient one, being the one with the least number of visited dead-ends on average, but this time DFS is following really closely, in some scenarios—the ones where the maze was generated using a DFS approach—was even found to have a lower number of visited dead-ends than BFS.

From this, it can be concluded that BFS is the most efficient uninformed algorithm for path finding, followed closely and be even replaceable by DFS.

An alternative for the algorithms showed in the previous work is the A* algorithm, which is the most widely used and most efficient path finding strategy, whose result is always the optimal path given that the chosen heuristic is both complete and consistent, as stated in [7]. Despite being the most efficient and widely used algorithm, its biggest problem is represented by its high usage of process memory. For this, the authors of the book recommend various modifications for reducing the needed memory by limiting the searchable area for the agent, such as by using Beam Search.

IMPLEMENTATION DETAILS

The design of the game was heavily centered on its long-term flexibility and scalability. Hence the chosen architecture is focused mainly on modularity, where each main functionality is separated into a different subsystem, as seen in Figure 2. Therefore, if follows the classical structure of a video game, with three main components, the rendering component, the game engine and the intelligent agent component, and three auxiliary ones, the input decoder, the game surface generator and the one comprised of the representative classes for the game and UI objects.

The communication between them is provided through flags and state management. This way, when the result of a state's routine after the user's input triggers another event—such as losing the game, which triggers the initialization of the lose screen—the intermediary state will be processed before continuing reading the user's input.

For the better orchestration of the events used for state processing, all modules are invoked in a main component, which doubles as an aggregator and as a controller. So, all user input and results from the subsystems are collected in this component, which later on caters the information to the appropriate structures for further processing.

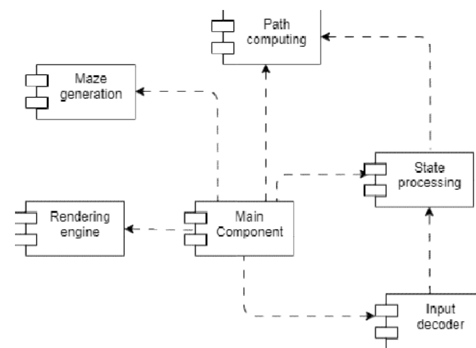


Figure 2: The communication between the modules of the system

Maze Generation

Following the conclusions of the research work summarized in the previous section, the algorithm of choice for generating the maze is *Eller*.

Instead of employing traditional algorithms used for com- puting the Minimum-Spanning Trees, the reasoning behind choosing this one is based on two main factors: its scalability potential and the more balanced results in terms of solving difficulty for both human users and intelligent agents.

There are two possible implementations for the algorithm—starting with an empty matrix followed by building the walls step by step, or starting with each cell surrounded by walls and then creating the pathways. For this project the latter was chosen so that the generation starts with each node separated in its own disjoint set that will later be merged into a singular connected component.

The pseudocode for *Eller*'s algorithm used for the implementation is the one defined by Jamis Buck [8]:

1. Initialize of the cells on the first row in individual sets.
2. Randomly concatenate neighboring cells which are not part of the same set.
3. Initialize the nodes in the next row.

4. Descend by randomly concatenating the nodes in the current row with the corresponding ones in the next row.
5. Repeat steps 2-4 until the execution reaches the last row of the maze.
6. In the last row concatenate all neighboring nodes that are not a part of the same disjoint sets.

The adaptation brought to this algorithm was to switch steps three and four so that the result can be stored for further usage in the main component.

A maze can be illustrated as an undirected acyclic graph, where the cells represent the nodes and the edges of the graph repre- sent the pathways. This way the cells' initializations can be equivalently interpreted as the nodes' initializations in a graph. These nodes store its coordinates in the graph/maze, the nodes it's connected to, and the parent node.

As stated in the second step, the random union with the neighboring cells is performed on nodes that are not part of the same connected component. For this, the first action after choosing whether or not to unite the current node to a nearby node is to check the direction in which the union will take place—will it be linked to the left or the right neighbor. After the direction is chosen, it's checked that the two nodes are not already indi- rectly connected, in which case the process is stopped and the execution continues with the next node. Otherwise, the two are finally coupled by storing their addresses in each others' neighbors lists.

As the algorithm is based upon the generation of the result row by row, a vital step is represented by expanding the connected components to the next rows. This action, similarly to the previous one, first chooses whether or not to link the current node with the one right under it. If the answer is yes, it starts the process directly, as they are by definition part of different disjoint sets. For guaranteeing that the result is correct, there are some cases in which it is highly important to descend to the node in the next row. These are characterised by the scenario in which the current node is the last one in the row that is part of the current connected component. If no previous nodes decided to go down to the next row, the current one must do so for the continuation of the set further in the maze, until it's united with another one.

After the nodes of the current row have been processed, the next row now becomes the current row. For that to happen, the addresses stored in the current row have to be updated with the ones currently in the next row, so that new sets of nodes can be initialized later on.

When reaching the last row, if there are more than one disjoint set remaining, it is its duty to make sure that the final result constitutes a single connected component that contains all nodes in the graph. For this, all nodes in the last row are analyzed and, if the neighboring ones are part of different sets, a link between them is set up so that the components are now merged.

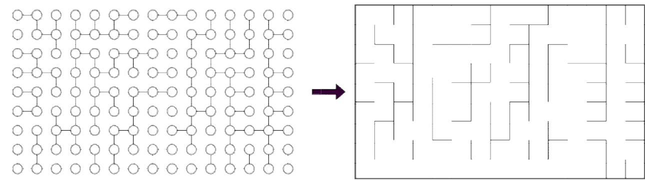


Figure 3: The generated maze using Eller's algorithm

The final result is an undirected graph with no cycles that can then be translated in the traditional maze visualization, just like in Figure 3.

Maze Validation

To make sure that the result is indeed correct—it is represented by a single connected component that contains all elements in the maze, so that any two nodes can be connected through one singular path—another process is developed for this step. According to Cormen et al. [9], in case of a static graph, a way of computing the connected components is by using the Depth-First Search algorithm.

Because of this, DFS has been chosen as the algorithm implemented for the validation process in which, instead of count- ing the time when each node is visited and then processed, it counts the number of visited nodes, that is later compared to the total number of nodes in the maze, and checks to see if the current node is neighboring a node (different from the parent node) that has already been visited, so that the maze will not have cycles.

Path finding

Generally, the clear choice for the intelligent agent after studying the existing solutions would be using the A* algorithm. In spite of its effectiveness, its biggest shortcoming is represented by its high usage of process memory, which would increase the workload for the processor in a dynamic scenario just like this one, where the path has to be computed after each iteration. Because of this aspect and the fact that the resulted maze is essentially a tree, Depth-First Search seems suited for the task of path finding, but with some slight adaptations to make it more time efficient, by avoiding to recompute the entire path.

Given that in a tree there is a single path between any two nodes, it means that the only existing path is already the optimal one between those two nodes two and also between any two nodes covered by the path. Thus, by extending this to the current maze, it means that after computing a path the first step would be to check if the new destination node is on the path or not. In that case, instead of recomputing the path, it would just provide the next coordinates to the intelligent agent. If, instead, the node is not part of the already computed path, analogous, the optimal path should intersect the already existing one, which means that part of the result is already computed and the only unknown part is from the destination to the current path.

The pseudocode for the path finding algorithm is based upon the one presented in Introduction to Algorithms [9], with the slight adaptations mentioned previously added to it, and can be observed in Figure 4.

The worst case scenario for this implementation is that in which the path from the new destination node and the existing path

intersect in the root node. In that case it would just follow the same steps as if there is no existing path stored, but with the extra steps

```
DFS_node(u, g, path)
    u.color <- GRAY
    if  u = g then
        return true
    for each v in u.neighbors do
        if v.color = WHITE then
            if DFS(v, g, path) is true then
                insert_first(path, v)
                return true
    u.color <- BLACK
    return false

DFS_list(u, crtPath, newPath)
    u.color <- GRAY
    if u is in crtPath then
        return true
    for each v in u.neighbors do
        if v.color = WHITE then
            if DFS(v, crtPath, newPath) is true then
                insert_last(newPath, v)
                return true
    u.color <- BLACK
    return false

DFS(u, g, path)
    if path is null then
        DFS_node(u, g, path)
    else if g is in path then
        delete the elements in the path after g
    else
        DFS_list(g, path, newPath)
        concat path and new path after the shared node
    return the newly computed path
```

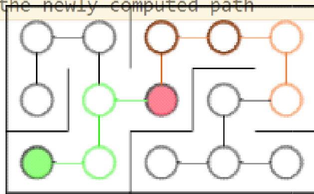


Figure 4: The updated DFS pseudocode

Figure 5: The worst case scenario for the current path finding algorithm

State orchestration

For the system to work accordingly, it has been designed as a state machine in which each state has a set of routines that can be triggered depending on the internal and external events.

For this, several decision trees have been drawn, one for the entire application, one for each main state, and two for the characters—one for the player character and one for the non-player character.

When talking about the entire system, the three states it can be found on are the *menu* state, the *game* state, and the *exit* state.

- *menu* state represents those moments in the execution of which the program has one of the menus as the active screen. They can be either the title screen, the win screen, the lose screen, or the pause screen, each comprising of a backdrop and two buttons that make up the menu.

of searching the nodes in the list used for storing the existing path. This can be easily visualized in Figure 5, where the green node is the destination, the red node is the root and the orange path is the already existing path

.

- *game* state is used for when the application has a game in progress. In this case, the input is mostly used to compute the state of the main character, but it can also trigger the *pause* menu, which would send the program back to the *menu* state. Naturally, this can progress to the *menu* state in three ways, by pausing, by losing or by winning the game.
- *exit* state is triggered when the corresponding button is pressed in the title screen. This one starts the processes needed for cleaning the allocated memory and destroying the window before the application stops its execution.

Going deeper, each one of them has a set of substates that define their progress, that can be visualized in Figure 6.

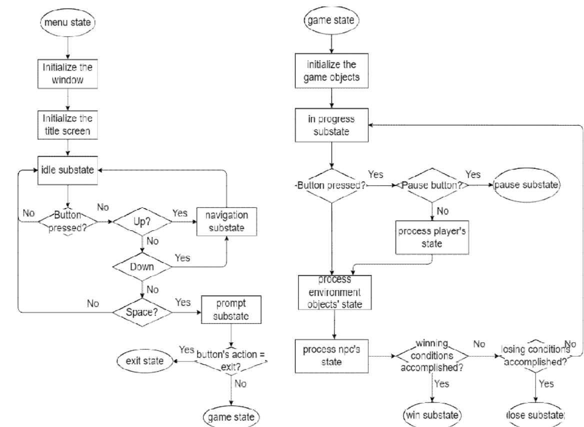


Figure 6: The state diagrams for the title screen (left) and for the game (right)

When talking about the characters, their states are similar, but with variations depending on their specific actions, which can be seen in the flowcharts depicted in Figure 7.

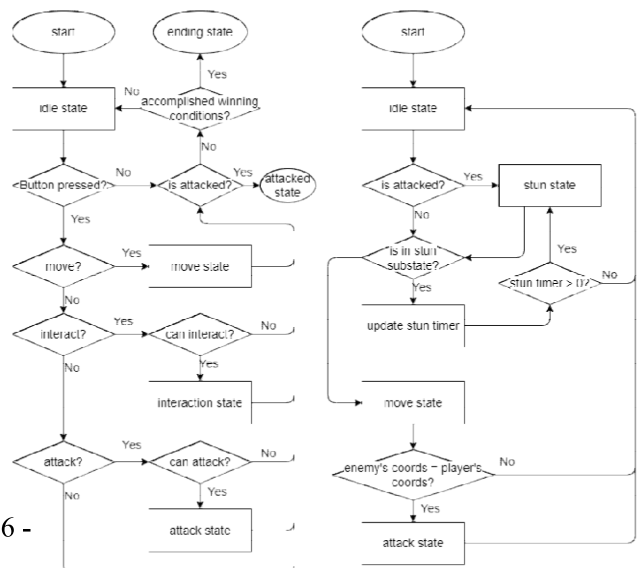
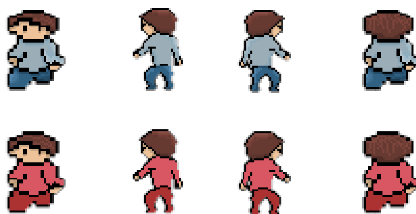


Figure 7: The characters' flowcharts: player (left), enemy (right)

Display and input capture

For rendering the game elements accordingly in a 2D top-down manner, the engine responsible with this task has been devel- oped using the second iteration of SDL (Simple Directmedia Layer) library. SDL offers support not only for displaying the objects on the screen, but also with the communication with the other hardware devices used for input or output [10].

For portraying the various objects on the screen, for each of them have been defined sprite sheets that the rendering engine can process and then display on the screen. The characters both have four frames that are defined, one for each navigation direction: up, down, left, right, that can be visualized in Figure



8.

Figure 8: Sprite sheets—player (top), enemy (bottom)

For the maze cells, there have been drawn 15 frames, for each combination of walls, and two additional ones for the destination cell, one for when it is locked, and one for when it is unlocked, which can be seen in Figure 9.
So, additionally to the rendering engine the input decoder has also been developed with the help of the library for its keyboard support.

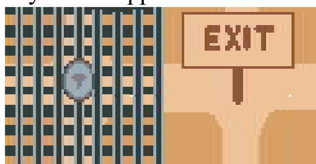


Figure 9: Destination cell's frames—left: locked, right: unlocked

TESTING AND VALIDATION

Experimental Testing

The game has been tested on a number of 8 participants, each playing 5 rounds after the warm-up ones used to understand the controls and the pacing of the game. Each player chosen is part of the 22-25 age group, with and without video previous gaming experience.

For each of them we computed the win to lose ratio, the average time spent in a game round, and the average time spent in a winning round. The statistics have shown that for inexperienced players the average time spent in game is higher than for the ones with previous gaming experience, but the difference is relatively small, the longest time being 24.83 seconds, followed in second place by 17.82 seconds, and the fastest time being 12.87 seconds. More details can be seen in Table 2.

No.	Avg. time <s>	Avg. time (Win) <s>	W/L ratio	Gaming experience
1	24.83	27.46	3/2	No
2	13.44	19.4	3/2	Yes
3	13.29	14.83	4/1	Yes
4	17.82	17.82	5/0	Yes
5	16.07	24.45	4/1	No
6	13.45	18.1	3/2	No
7	12.87	13.17	4/1	Yes
8	14.19	16.29	4/1	Yes

Table 2: Players' statistics after the experimental testing of the application

Another aspect that can be observed from this table is the difference between the average time spent in winning rounds and the average time overall. This can be explained by the shorter losing rounds, courtesy of the enemy catching the player's character during the beginning of the rounds, before it has time to navigate through the maze properly. Here the average longest registered times for winning rounds is 27.46 seconds and the shortest 13.17 seconds. What is also interesting to see is that, beside the longest time being acquired by an inexperienced user, the shortest registered time by an inexperienced user is 18.1 seconds, which is comparable to the times obtained by the more experienced users.

Regarding the win to lose ratio, the results are similar among all users, where the majority obtained a 3/2 ratio (3 participants), and four obtaining a 4/1 ratio. The outcome shows that the game is balanced, as it has a medium to easy difficulty, and it's easy to understand both by seasoned players and newcomers.

In terms of player satisfaction, each participant has been asked to rate the game based on seven criteria, ranging from the aspect of the menu and of the game, to the menu navigation and control of the character, to the maze's and enemy's dif- ficulty and to the user's experience overall. The consensus has been that the participants were pleased with the aspect of the application and the game's controls. In terms of the maze and the enemy, they rated them at a medium difficulty. Overall, all players claimed they were satisfied by the gaming experience. Their feedback on what can be improved centered around the player experience, by integrating the timers in the window so that the player can properly see when they can use an ability again, and increasing the competitiveness of the game by introducing a scoring system and a hall of fame view in which the users can see who has the highest scores.

Performance testing

For the purpose of performance testing, the application had been tested on a Lenovo laptop with the following

specifications:

- Intel Core i7 2.30 GHz CPU
- 32GB RAM memory
- NVIDIA GeForce RTX 3050 GPU.
- Windows 11 OS

Maze generator

For measuring the performance of the implemented maze generation algorithm, the process has been called to create a 100x100 cells maze, similarly to the algorithms included in the analysis contained in [3].

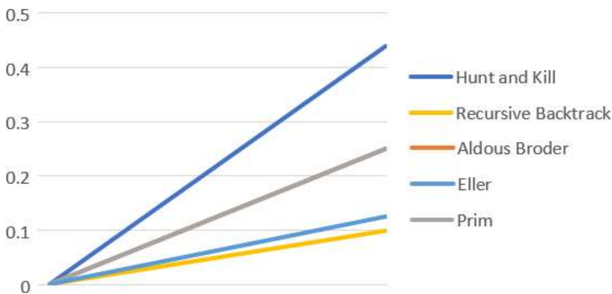


Figure 10: Top 5 performing algorithms for generating a 100x100 maze

By analyzing the profiler, after the call of the function at 8.437, the task was in execution until 8.563, which meant that the algorithm's implementation needed 0.125 seconds to generate the result. Comparing this result to the ones in Fig. 1 (encompassed in Fig. 10), this time places Eller's algorithm in the second place behind Recursive Backtrack—who has a time of around 0.1 seconds—and ahead of Prim and Aldous Broder—both with times over 0.2 seconds.

System's performance

For measuring system's performance, the process memory needed during the running of the application has been followed in the profiler. For covering all bases, the test consisted of opening the application, playing a full game round, reaching the end title card, starting another game, pausing, resuming the game, then pausing again and exiting the game from the pause menu, then closing the application from the title screen.

For this analysis, 8 snapshots have been taken in various moments of the execution of the program:

1. Opening the application, in the title screen, the process memory reached and stayed at 183 MB.
2. During the first game round, when in the beginning it was observed a spike in the process memory, jumping from 183 to 208 MB, then decreasing and staying at 202 MB. The spike could have been caused by the initialization routines of the game, with the generation of the game surface and resetting the game objects. Once those routines ended, the memory evened at 202 MB during the entire game.
3. When the game reached the end state and the winning title card was initialised, the process memory decreased again to 183-184 MB.
4. When starting the second round another spike in the mem- ory could be seen at 208 MB, then followed again by the decrease and stabilization at 202 MB.
5. Opening the pause menu, when a spike to 210 MB was registered followed by the return and stabilization at 202 MB.
6. Closing the pause menu and resuming the game, when after a hiccup to 205 MB it returned to 202 MB.
7. During the second opening of the game menu, when nothing changed in terms of process memory.
8. Going back to the title screen, when after a spike to 210 MB it immediately decreases to 184 MB.

When exiting the program it's captured a constant decrease that reaches 173 MB when the profiler stops running.

This shows that, in terms of performance, the allocations and deallocations are done accordingly, with little to no residual allocated memory remaining and no leaks registered during the execution of the program.

CONCLUSIONS

The research and development work presented within this paper focused on identifying and implementing time and resource-efficient pseudo-random maze generation and path finding methods to be used as key components for a game engine meant to facilitate the development of maze-chase type games. We explored various maze generation and path finding methodologies to enhance the gameplay experience for both human players and intelligent agents.

We selected Eller's algorithm for maze generation due to its scalability and balanced difficulty. This proved to be highly competitive when compared to the traditional methods for computing MST, by also generating a result that balances the long and winding roads with the larger number of intersections, making it more challenging for both human and intelligent agents, results that can be seen in the section focused on Testing.

While A* is widely regarded as the most efficient path finding strategy, its high memory usage posed a challenge in our dynamic scenario. Therefore, for path finding through the resulted maze, we adapted Depth-First Search to leverage previously computed paths, reducing the need for re-computation.

With these maze generation and path finding components as the basis, we described a game engine that can be used for maze-chase game development, which we piloted it by developing a game and testing it for user and system performance, as well as user satisfaction.

As for future improvements, there are two main concerns: the algorithmic nature of the project and the potential game engine developments.

For the former, these improvements are related to the further

optimization of the used algorithms—treating the resource efficiency problem in the worst case scenario for the enhanced DFS implementation, and reducing the process time for maze generation by using multithreading.

Acknowledgment

This work was supported by the project AITECH - "Excellence research in the field of artificial intelligence and big data", 38PFE/2021.

REFERENCES

[1] M. Jakobsson and L. Carney, "Games innovation: The role of games in societal innovation," 2023. [Online]. Available: https://gamelab.mit.edu/research/games-innovation/

[2] B. Courtehoute and D. Plump, "A fast graph program for computing minimum spanning trees," *arXiv preprint arXiv:2012.02193*, 2020. [Online]. Available: https://arxiv.org/abs/2012.02193

[3] P. Gabrovšek, "Analysis of maze generating algorithms," *IPSI Transactions on Internet Research*, vol. 15, pp. 23–30, Jan 2019.

[4] A. Kozlova, J. Brown, and E. Reading, "Examination of representational expression in maze generation algorithms," in *IEEE Conference on Computational Intelligence and Games*. IEEE, Aug 2015.

[5] A. Dagaev, A. Sorokin, R. Kovalenko, and E. Yakovleva, "Mazes creation for further study of swarm intelligence," *IOP Conference Series: Materials Science and Engineering*, vol. 919, p. 052058, Sep 2020.

[6] J. Buck, "Maze generation: Algorithm recap," Feb 2011. [Online]. Available: https://weblog.jamisbuck.org/2011/ 2/7/maze-generation-algorithm-recap

[7] S. J. Russell and P. Norvig, *Artificial Intelligence: a modern approach*, 4th ed. Pearson, 2021.

[8] J. Buck, "Maze generation: Eller's algorithm," Dec 2010. [Online]. Available: https://weblog.jamisbuck.org/ 2010/12/29/maze-generation-eller-s-algorithm.html

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms ed. 3*. Cambridge, Massachusetts: MIT Press, 2009.

[10] S. D. Layer, "Simple directmedia layer - homepage," 2024. [Online]. Available: https://www.libsdl.org/