Impact of algorithmic and data structure implementation to game development

Yuzhe Xie[1,4,7], **Junjie Zheng**[2,5], **Kerui Chen**[3,6]

[1]College of Letters and Science, University of Wisconsin-Madison, Madison, 53706, United States
[2]Electrical Engineering and Computer Science, Syracuse University, Syracuse, 13210, United State
[3]College of Arts and Sciences, University of Miami, Miami, 33146, United States

[4]xie97@wisc.edu
[5]jzheng07@syr.edu
[6]kxc1006@miami.edu
[7]Correspondence author

Abstract. Many game developers tend to build their proprietary game engines with various algorithmic and data structure implementations. This work discusses a game made using the Pygame library that utilizes a maze generation algorithm, collision detection algorithm, and 2d shadow drawing algorithm to analyze how programming implementation could impact the game's representation, optimization, and future development. In this paper, various algorithms and data structures are compared based on their characteristic and performance, which leads to an analysis of how the game would differ from its current state if alternative implementations are used. It also addresses the issues raised by algorithms and data structures based on their interdependence relationship.

Keywords: Game Development, Framework, Algorithms, Data structures, Optimization.

1. Introduction

In making a game, game programmers would use various algorithms to implement game ideas, such as game arts or game mechanisms. In typical cases, the quality of a game solely depends on the game's concept. In contrast, sometimes the algorithmic approach to the game becomes a second determinant in the final representation of the game. In making a 2D game using the Pygame library, we experimented with alternative approaches for coding the game from multiple aspects, including graphics, level procedural generation, and collision detection. We will discuss how data structures and algorithms impact the game's performance and how they would influence its future development by composing the core framework.

2. Method

We divided our game into a series of steps to build the game's framework. Initially, the method of representing all game objects (obstacles, bullets, and characters) determines the difficulty of handling game data in future development.

2.1. Game Object Representation

The method of representing game objects can be crucial to the game's speed. Complex data structures can lead to difficult access to game data and slow execution time. Considering that Python can be inefficient in handling geometrical calculations with polygons (Calculating intersections, etc) as an interpreted language, we used only rectangle shapes to simplify the analysis of shadows and ray casting on collision with edges. Specifically, we utilized the Rect class from the Pygame library to reduce the time for constructing classes and methods for base structures representing sprites.

2.1.1. Geometry Calculation

To implement the ray casting function, we created a class that starts from a Point with 2-dimensional coordinates up to polygons that consist of multiple segments. These geometrical data serve as mathematical representations for all in-game objects handled in other functions requiring ray calculation.

2.2. Maze Generation

2.2.1. Aldous-Broder

The Aldous-Broder algorithm is a unique maze generation algorithm that can create a uniformly random spanning tree. This means that given an empty grid, the algorithm has an equal probability of generating any valid maze. The Aldous-Broder algorithm starts from a random cell on the grid and performs a random walk, which means it goes to a random neighboring cell at each step. When it encounters an unvisited cell, it carves a path to this new cell. This process continues until all cells have been visited. The distinctive characteristic of this algorithm is that it treats all possible mazes equally likely. This results in a uniform distribution of all possible mazes generated from a given grid, a property that most other maze generation algorithms lack.

2.2.2. Storing Data Structure

The basic unit for maze generation is a cell, represented using the ‘Cell‘class. Each cell knows its row and column, its four neighbors (north, south, east, west), and which of those neighbors it is linked to. The ‘Grid‘ class holds all cells. It consists of a two-dimensional array representing the maze's cells and contains methods for generating and manipulating the maze. The data structures are designed to easily manage and manipulate the maze, allowing for the clear and concise implementation of the Aldous-Broder algorithm. The Aldous-Broder algorithm is implemented in the” Aldous _broder()” function. It starts with a random cell and performs a random walk across the grid, linking unvisited cells until all cells have been visited. The implementation is simple and intuitive, capturing the essence of the algorithm’s description.

2.3. 2D Visibility

The 2D visibility adopted for our game slightly differs from the standard ray collision method. The standard algorithm emits a constant number of rays with fixed lengths from the point of a light source and constructs mini triangles between each intersection with obstacles [1]. During our game-making, we drew the shadow of all obstacles in the camera by creating a bounding box centered at the light source. Only rays start at the light source, and vertices of obstacles would be cast. They would form polygons or triangles with edges of the bounding box. The time complexity of this algorithm scales by the number of obstacles in the camera.

2.3.1. Segment Shadowing

We started experimenting with the shadow drawing process by drawing the shadow of a single segment from a light source. In the game prototype, the piece stands for a wall with no width.

2.3.2. Rectangle Shadowing
To draw shadows for all rectangle objects, the shadow function would loop through each rectangle and draw every shadow polygon for the edges of each rectangle. A rectangle instance marks the obstacle shadow as a unit.

2.3.3. Truncating Edges
Edges of intersecting rectangles are truncated due to redundant shadow polygon produced by this ray casting algorithm. Each rectangle has four arrays of segments that correspond to existing features on their edges. After the pre-processing, the shadow function calculates shadow polygons from all existing features on the rectangles' edges.

2.4. Collision Detection
We employ three methods to detect collision: Pygame's built-in collide detection, predictive collide detection, and stepped divide collide detection. Each technique includes distinct characteristics and weak points in resource consumption, time efficiency, and detection accuracy. We use them for tasks such as bullet collisions, enemy and player movement collisions, and item collision detection.

2.4.1. Pygame's Built-in Collision Detection
The built-in collision detection provided by Pygame is easy to use. It checks the collisions between two items at each tick. If two things overlap within one tick, their clash will be detected. However, this method is based on ticks, meaning it only checks for collisions at specific frames in the game loop. If an object moves too fast, it will pass through other things in one tick without overlapping with it, so the collision detection method will miss this collision. Because of this limitation, this collide detection is used for more straightforward and slower collisions, such as collisions between the player and stationary objects like items and walls.

2.4.2. The Predictive Collision Detection
Predictive collide detection, instead of checking for collisions at each frame, will ”predict” the movement of an object: it draws an extension line from the object's initial position to its projected final position based on the object's velocity and direction. And then, check if it collides with another object or the extension line of the other object [2]. If an intersection is found, then it means a collision happened. This method is more reliable for detecting collisions between fast objects. However, it requires more complex calculations and more resources to work smoothly. Therefore, we use this collide detection for fast items like bullets colliding with stationary objects like walls.

2.4.3. Stepped Collision Detection
Stepped collision detection enhances Pygame's built-in method by dividing each tick into smaller "steps." These steps can check collisions individually; the more steps you use, the more frequently collisions are checked [3]. So that there will be less ratio of the collision being missed. The number of steps can be adjusted based on available hardware resources, making it adaptable to simple and complex situations. However, extensive testing is needed when applying it in different areas. We found this method can be used for almost all collision situations we met during the game design.

2.5. Enemy AI
A finite machine system determines the behaviors of enemy characters based on the player's actions. The machine includes four states, each with its actions and conditions for transitioning to another state. We separate the functions in which AI performs activities and the controlling AI to facilitate adding new enemies with the same AI or unique AI in future development. Each enemy has a specific class that inherits an Enemy class, specifying generic enemy behaviors such as piloting to a certain point on the map. An AI class defines a primary finite state machine with states PATROL, CHASE, ATTACK, and

INVESTIGATE, prototyping a ranged or melee enemy AI that investigates noises the player makes and chases them down once they gain a direct line of sight to the player.

2.5.1. Enemy Waypoints

Since the game's map consists of a randomly generated maze denoted by an array of rectangles, the enemy must navigate corridors without colliding with walls. To simplify the problem, waypoints are transformed into Cell instances by their coordinates. Then, we perform A* in the maze from the starting to the destination cell, generating the path in a Cell maze [4]. To transform the path back to waypoints with coordinates, we convert each cell in the path to a point in the corresponding cell. The programmer passes the destination coordinates into a function that performs an A* search; it returns an instance of the Waypoint class that encapsulates an array of coordinates.

2.5.2. Stop Condition

We experimented with two ways of implementing the stopping condition for a moving enemy. The first method is to compare two coordinates directly. The second method is to create a small square box centered at the destination point, which collides with the moving enemy.

3. Discussions

3.1. Game Effect

3.1.1. Rectangular Objects

Rectangular object representation limits the game experience and visuals of the game. Adopting rectangles in representing sprites converts the problem of collision to rectangle collision, which can cause the issue of inaccurate sprite display. Failing to achieve pixel-perfect collision may show more significant overlap between sprites during their clash. The visual of the map tends to lack smoothness as a maze would be visually presented as a room made out of rectangular walls. The textures of the floor have to be displayed as squares to fit into every room, creating a sense of constraint in a game that allows the player to move in all directions freely.

3.1.2. Maze Difficulty

The maze-solving performance is examined using the Breadth-First Search (BFS) and Dijkstra's algorithm. Each algorithm's time complexity and path length are compared across various maze sizes. In addition, the percentage of dead-ends linked in each maze can influence the difficulty of the maze, which in turn impacts the performance of each algorithm. We can determine which algorithm best suits different mazes by analyzing these results. The use of the Aldous-Broder algorithm dramatically influences the game outcomes. The unpredictability and complexity of the mazes generated are directly tied to the player's experience and challenge level. For instance, a player might encounter a maze with several dead ends, increasing difficulty and longer gameplay time. At the same time, in another session, they may face a more straightforward path. This variability in results keeps the game fresh and engaging, prompting players to adapt their strategies continually.

3.1.3. Customized Shadows

Drawing shadows based on segments brings flexibility in shadow control. Obstacle shadows in the player's visibility range can be rough if no parts of the edges are removed. When components are over-truncated so that the cloud is turned off on both sides of a rectangle, the player may see through a partial obstacle, which becomes an unintended result. Since the programmer may exert customized control during segment truncation, drawing shadows by segments enables game developers to try different ways of displaying the cloud. For example, the game developer may choose to truncate only parts helped such that the shadow would not cover the obstacles, allowing the player to see the shape of the challenges.

3.1.4. Maze Algorithm Choice

The most direct application of the Aldous-Broder algorithm is maze generation, particularly when an unbiased maze is required. This uniform randomness can be essential in game design, art, and computer simulations. In each case, the algorithm's ability to create any possible maze with equal likelihood can lead to more varied and less predictable outcomes. Additionally, as the algorithm is based on the concept of a random walk, its study can have implications in fields where random walks are relevant, such as physics, economics, and computational bi- biology. While the Aldous-Broder algorithm might not be the most efficient, its unique characteristic of creating unbiased mazes and its ease of implementation make it a significant algorithm to study and understand. Another well-known approach is Prim's algorithm, a greedy method that finds a minimum spanning tree for a weighted undirected graph, thus creating a ”perfect” maze without loops or isolated walls. The algorithm is simple to understand and implement, and it establishes mazes with a complex ````appearance. Nevertheless, similar to DFS, it produces mazes with a distinct bias, leading to predictable patterns. Kruskal's algorithm, similar to Prim's, is a minimum-spanning-tree algorithm. It treats the maze as a disjoint set of cells and progressively combines them to form the maze. The algorithm is unbiased and creates mazes with a pleasing organic feel. However, implementing Kruskal's algorithm is more complex, requiring extra memory to store the disjoint sets. In contrast to the algorithms above, the Aldous-Broder algorithm generates unbi- ased mazes, i.e., it has an equal probability of developing any valid maze. This characteristic makes the Aldous-Broder algorithm unique and desirable for applications where an unbiased generation is crucial. Despite this advantage, the Aldous-Broder algorithm is not as widely studied, possibly due to its longer expected runtime than other methods.

3.1.5. Pathfinding

Although BFS and Dijkstra's algorithms can find the shortest path in a maze, their efficiencies can vary depending on its characteristics. For mazes with uniform path costs, such as the ones generated by our implementation, BFS tends to perform well due to its simplicity and speed. On the other hand, Dijkstra's algorithm would be more effective in scenarios where path costs vary, such as weighted mazes.

3.1.6. Collision Detection Method Influence

The choice of collision detection method impacts the game mechanics and performance significantly. While Pygame's built-in collision detection can handle slow-moving objects well, it's hard to deal with fast-moving objects, especially in scenarios involving bullets and rapidly moving objects. On the other hand, predictive collision de-detection does well with fast-moving objects and contributes to a more fluid gameplay when bullet-object interactions are critical. Stepped collision detection offers a balance between the two, improving the game's overall consistency in a collision and making gameplay smoother and more predictable.

3.2. Optimization

3.2.1. Shadow pre-processing

Controlling the level of detail segments that need to be truncated can impact the game's loading speed. While each rectangle has four edges, increasing the scale of the maze would exponentially increase the number of the rectangle, thus requiring longer pre-processing time for segments. Furthermore, the more detailed the details are, the more polygons will be yielded when calculating shadows. For example, a single edge of a rectangle may be divided into subsegments that demand more calls to the shadow function, slowing down the execution of the game loop.

3.2.2. Random Maze Generation

Implementing the Aldous-Broder algorithm has also played a significant role in optimizing the game. The efficiency of the al- algorithm enables the rapid generation of mazes, ensuring seamless gameplay without lag or delays. The diversity it provides has eliminated the need for manually designing maze patterns, thus reducing the development workload. Furthermore, it has improved the user experience by providing unique and diverse gameplay situations driven by maze patterns.

3.2.3. Optimized for Rectangles
Since the game's framework was built on rectangles, calculations in shadow and collision only consider the situation of a four-sided polygon and perform the corresponding optimization. Collision detection and shadow calculation only need to handle each object's four edges instead of generic polygon calculation. The Pygame library also contains built-in functions that rely on rectangle calculations in graphic rendering and geometric calculations.

3.2.4. Collision Detection Efficiency
The efficiency of each collision detection method dramatically affects the game's performance. Pygame's built-in process, while being the simplest, can become a performance limitation in games with many slow-moving objects due to its tick-based checking. Predictive collision tection, while providing a solution for fast-moving objects, demands additional computational resources for complex calculations. Although adaptable to different scenarios, she-stepped collision detection requires intensive testing and might lead to other CPU usage for increased stepping. Balancing collision accuracy and performance is essential to ensure a smooth game-play experience.

3.3. Future Development

3.3.1. Finite Machine Decision
By establishing a prototype of the enemy, we can conveniently add more enemies as long as they need to attack. Enemies that require navigation can be designed and created by inheriting the Enemy class. By defining the unique attack function, enemies with different attack methods can increase the game content. Enemies that require other AI logic can inherit the base AI class and redefine their behavior. As a result, the separation between the Enemy class and their behavior mechanic AI class significantly decreases the coding work in the game's future development by adding more enemies.

3.3.2. Automated Optimization of Stepped Collision Detection
In the future, we plan to incorporate an automated function for optimizing stepped collision detection. This function will dynamically adjust the number of steps based on game scenarios and hardware resources, improving game performance and accuracy. It will decrease the manual work in testing, making the game engine more adaptable and efficient across varying hardware conditions.

3.3.3. Enhanced Collision Detection
The current collision detection methods serve the game well but have their limitations. Future development can focus on introducing advanced collision detection algorithms that can handle both slow and fast-moving objects without relying on stepping or predictive strategies. This will make the gameplay experience more realistic and fluid.

4. Limitation
Since the data structures and algorithms, we choose to have their limitations compared to alternative solutions, the impact we observed during our game development may not represent common issues or benefits of algorithms in game development. This can be caused by the time complexity of specific algorithms' limitations on algorithmic and data structural implementations based on it. For example, without implementing the binary space partitioning algorithm, which partitions the space into multiple sub-spaces that limit the collision detection to subspace, reducing the number of objects needs to be checked [5]. It potentially improves the efficiency of collision detection and shadow drawing; the game only runs smoothly when the shadow and collision algorithm reduces its precision. Furthermore, some alternative algorithms perform significantly differently than the algorithms of our choice. The shadow

drawing al-algorithm can be implemented using the ray casting algorithm mentioned. Hence, it generates a stable performance in displaying the player's view, which we have yet to experiment with its performance in the environment of Pygame. Consequentially, the final representation of the game can vastly differ from the current competition in terms of optimization and game effects with unexplored algorithms and relevant data structures.

5. Conclusion

Building a game framework requires careful decisions on base data structures representing objects and corresponding algorithms. Base data structures determine optimizations for relevant algorithms available to the game. Using rectangles to represent game objects in a 2D game may limit the game content and visual effects due to the lack of art, which requires attention to extra graphical details and work. Graphic work for displaying the map in a smooth style. The second most impact algorithm and data structures can have on a video game's development falls to overall performance and optimization for the game. Algorithms such as binary space division are expected to significantly increase the performance of other algorithms in the game that can increase the overall refresh rate. Apart from the profound influence of base structures and algorithms, the choice of algorithms may incline towards algorithms that exhibit constant time complexities so that the game remains in a stable performance by the increase of complexity of game contents. While it is crucial to select a proper algorithm from its alternatives based on the optimization, game developers also prioritize the characteristics of the algorithm to support the game mechanic. The maze generation algorithm and corresponding path-finding algorithms must match the game's map design. At the same time, the step-by-step precision collision detection enables the game's mechanic of the bullet to be implemented. During the game's development, a game developer must also consider how specific implementation of the game may affect the difficulty of enlarging the game's content. The framework of enemy AI ensures that a diversity of new enemies can be implemented with a small amount of work based on existing data structures and classes. Some algorithms may need to be replaced with efficient substitutes to optimize these. As a result, the selection of data structures and algorithms can impact game development from perspectives of game effects they support, optimization be- tween relevant algorithms, and future development that amplifies the game's content and optimizes the game.

References

[1] Red Blob Games.(2012). 2d visibility. https://www.redblobgames.com/articles/visibility/.
[2] BrendanL.K. (2013). Swept AABB Collision Detection and Response. https://www.gamedev.net/articles/programming/general-and-gameplay-programming/
[3] Continuous collision detection. https://nphysics.org/ continuous_collision_detection/
[4] Red Blob Game. [n. d.]. Introduction to A*. https://theory. stanford.edu/~amitp/GameProgramming/AStarComparison.html
[5] Ryan Frazier. (2021). 2D Binary Spacing Partitioning with Python and NetworkX. https://www.fotonixx.com/posts/ 2d-binary-spacing-partitioning-with-python-and-networkx/
[6] Berglund(2017) Carl Berglund. 2017. Designing a simple game AI using Fi- nite State Machines. designing-a-simple-game-ai-using-finite-state-machines. https://www.gamedeveloper.com/programming/swept-aabb-collision-detection-and-response-r3084/
[7] Buck. (2011). Maze Generation: Aldous- Broder algorithm. http://weblog.jamisbuck.org/2011/1/17/ maze-generation-aldous-broder-algorithm
[8] Edward Green. (2020). Procedural maze generator using python. https://edtg.co.uk/2020/04/16/procedural-maze-generator-using-python/
[9] Jon. (2015).Continuous Collision Detection (CCD). https://www.stencyl.com/help/view/continuous-collision-detection/
[10] Daemian Mack. (2019). Mazes for Programmers: Dijkstra's algorithm. https://daemianmack.org/posts/2019/12/mazes-for-programmers-dijkstras-algorithm.htmls

[11] Deepak Reddy. [n. d.]. Collision Detection in pygame susing Python. https://www.codespeedy.com/ collision-detection-in-pygame-using-python/