Course: KOU001EEC BSc (Hons) Informatics

Module Code: KOL304CR

Module Title: Games and AI

Module Leader: Lena Erbs

Hand out date: February 2025

Submitted by: PAUL MICKY D COSTA

Submission deadline: 26/03/2025

Github link: https://github.com/paulmicky1/portfolio_gamesandai

Portfolio: AI Techniques in Real-Time Games

Contents

1. A* Algorithm

Description of the Technique

A* (A-star) is a widely used algorithm for pathfinding in game development and artificial intelligence. It is an extension of Dijkstra's algorithm, where a heuristic is added to prioritize paths that seem more promising, leading to faster solutions. The algorithm works by maintaining two key values for each node:

- g(n): The cost from the start node to the current node.
- h(n): The estimated cost (heuristic) from the current node to the goal node.

The total cost f(n) is calculated as the sum of g(n) and h(n) (i.e., f(n) = g(n) + h(n)), and the algorithm always expands the node with the lowest f(n) value.

A* is widely used for real-time pathfinding, especially in games, where characters or NPCs need to find the shortest and most efficient route while avoiding obstacles. It is highly efficient, guarantees the shortest path if the heuristic is admissible, and is versatile enough to be applied to both grid-based and continuous spaces.

Implementation

The A* algorithm is implemented in Python using the pygame library to visualize the process. In this implementation, a grid-based world is used where each cell represents a traversable space or an obstacle. The key steps in the implementation are:

- **Grid Representation:** The world is divided into a grid, and each grid cell is either a free space or a barrier.
- **Nodes:** Each cell is represented as a Vertex, which contains information about its position, neighbors, and the state (open, closed, or barrier).
- **Heuristic Function:** The heuristic used in this implementation is the Manhattan distance (vertical distance) between the current node and the goal node.
- **Priority Queue:** The open_set is implemented using a priority queue, which ensures that nodes with the lowest f(n) values are expanded first.
- **Path Reconstruction:** Once the goal node is reached, the path is reconstructed by backtracking from the goal node to the start.

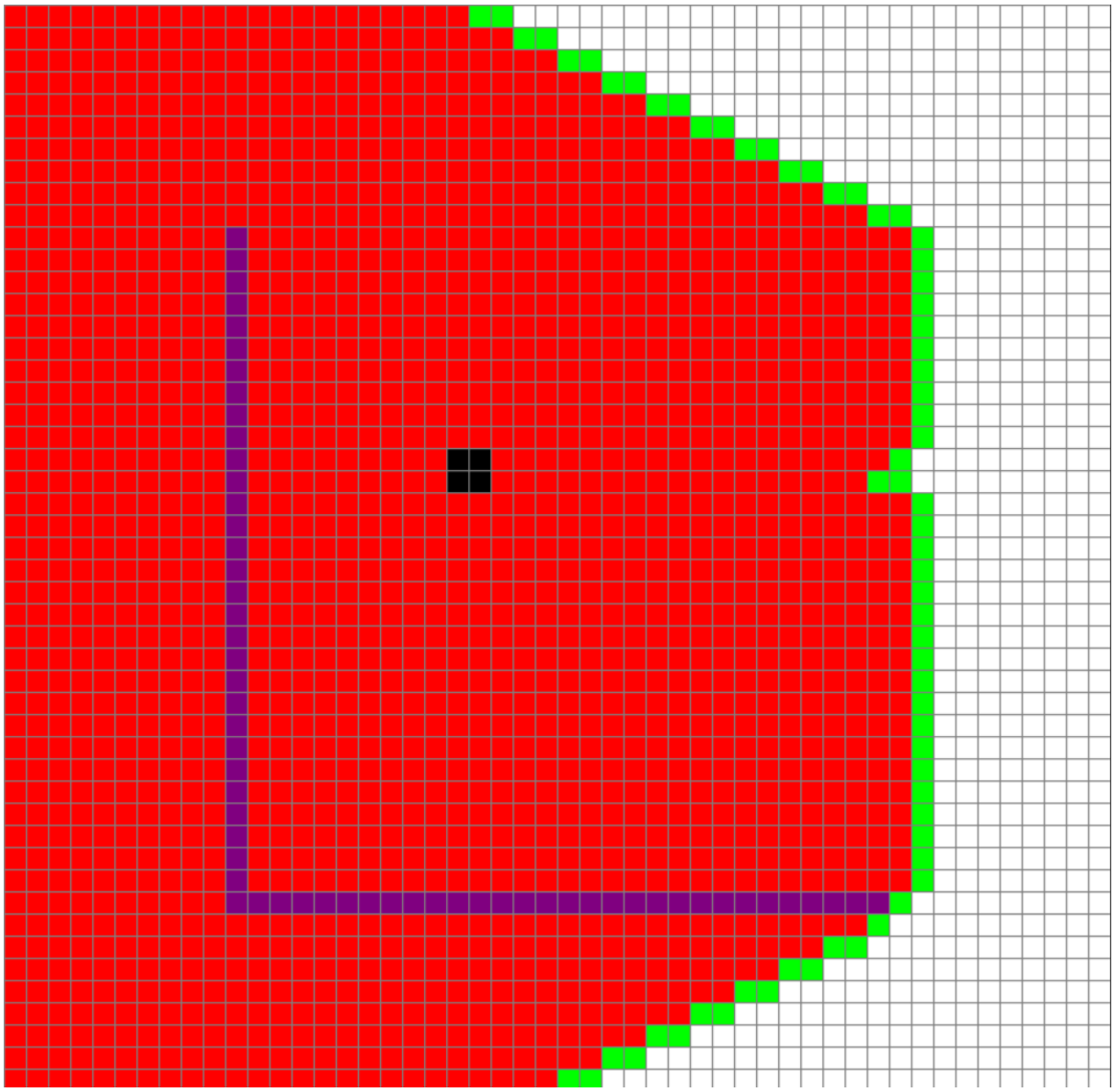
Here's a simplified version of the code implementation:

```python
import pygame
import math
from queue import PriorityQueue

# Define colors and grid setup
WIDTH = 800
WIN = pygame.display.set_mode((WIDTH, WIDTH))
pygame.display.set_caption("A* Path Finding Algorithm")

# Heuristic function (Manhattan distance)
def h(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return abs(y2 - y1)

# A* Algorithm Implementation
def astar(draw, grid, start, end):
    count = 0
    open_set = PriorityQueue()
    open_set.put((0, count, start))
    came_from = {}
    g_score = {vertex: float("inf") for row in grid for vertex in row}
    g_score[start] = 0
    f_score = {vertex: float("inf") for row in grid for vertex in row}
    f_score[start] = h(start.get_pos(), end.get_pos())
    open_set_hash = {start}

    while not open_set.empty():
        current = open_set.get()[2]
        open_set_hash.remove(current)

        if current == end:
            reconstruct_path(came_from, end, draw)
            end.make_end()
            return True

        for neighbor in current.neighbors:
            temp_g_score = g_score[current] + 1
            if temp_g_score < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = temp_g_score
                f_score[neighbor] = temp_g_score + h(neighbor.get_pos(), end.get_pos())
                if neighbor not in open_set_hash:
                    count += 1
                    open_set.put((f_score[neighbor], count, neighbor))
                    open_set_hash.add(neighbor)
                    neighbor.make_open()

        draw()
        if current != start:
            current.make_closed()

    return False

# Visualizing the grid and user interaction for start, end, and obstacles
def make_grid(rows, width):
    grid = []
    gap = width // rows
    for i in range(rows):
        grid.append([])
        for j in range(rows):
            vertex = Vertex(i, j, gap, rows)
            grid[i].append(vertex)
    return grid
```

Output Visualization:

The following image illustrates the result of the full implementation of the A* pathfinding algorithm. It shows how the algorithm dynamically calculates the shortest path from the start point to the end point while avoiding obstacles. The grid representation includes

the different states of nodes such as the start, end, barriers, and the final path, providing a clear visual of how A* operates in real-time.



The full implementation of the A* pathfinding algorithm can be accessed on GitHub
https://github.com/paulmicky1/portfolio_gamesandai/blob/main/Astaralgorithm.py

Reflection

Challenges Faced:

- Choosing the Heuristic: Different heuristics were tested, but the Manhattan distance heuristic was selected for its efficiency in grid-based worlds without diagonal movement.
- Real-Time Interaction: Handling dynamic user input (defining start and end points, placing obstacles) while continuously updating the visualization was a challenge. Ensuring smooth interaction was key.

- Large Grid Performance: On larger grids, the algorithm's performance can degrade. Optimizations such as hierarchical pathfinding or Jump Point Search (JPS) can improve efficiency.

Strengths and Weaknesses:

- **Strengths:**
    - Guarantees the shortest path as long as the heuristic is admissible.
    - Efficient for smaller to medium-sized grids, ideal for most game applications.
    - Flexible and works well with dynamic obstacles.
- **Weaknesses:**
    - Performance can be slow on large grids, especially when the heuristic is not optimized.
    - Computationally expensive, particularly when dealing with large, complex maps in real-time scenarios.
    - Can be outperformed by specialized algorithms in specific use cases (e.g., JPS for large open-world maps).

2. Dijkstra's Algorithm

Description

Dijkstra's algorithm is a well-known graph search algorithm that finds the shortest path between two nodes in a graph. In the context of game AI and pathfinding, it is commonly used to compute the optimal route from a starting point to a goal. It works by exploring all possible paths in increasing order of their distance from the starting node, ensuring that the shortest path is found before exploring longer paths.

In video games, Dijkstra's algorithm is particularly useful when the environment consists of fixed obstacles, and the goal is to find the most efficient route from one point to another. It is an important algorithm for static pathfinding scenarios, where the map or grid does not change during the pathfinding process.

Unlike the A* algorithm, which uses a heuristic to prioritize exploration, Dijkstra's algorithm explores all possible paths evenly, making it more exhaustive but slower in certain scenarios.

2. Implementation

The Dijkstra algorithm was implemented using Python and the pygame library for visualization. Below are the steps taken to implement the algorithm:

Key Components:

- **Vertex Class:** Represents each grid cell, including attributes such as its position, neighbors, color, distance from the start node, and a reference to its previous node in the shortest path.
- **Distance Calculation:** Each node maintains a distance score, which is initially set to infinity except for the start node, which has a distance of zero.
- **Priority Queue:** A priority queue is used to always explore the node with the smallest distance first, ensuring efficient exploration of the graph.
- **Path Reconstruction:** Once the end node is reached, the algorithm traces back through the nodes' previous references to reconstruct the shortest path.

The algorithm runs by initializing the start node’s distance as 0 and exploring its neighbors. The distance of neighboring nodes is updated as the algorithm continues, ensuring that each node's distance reflects the shortest path found so far. The process continues until the end node is reached.

Here is a code snippet that illustrates the Vertex class and how the algorithm processes each node:

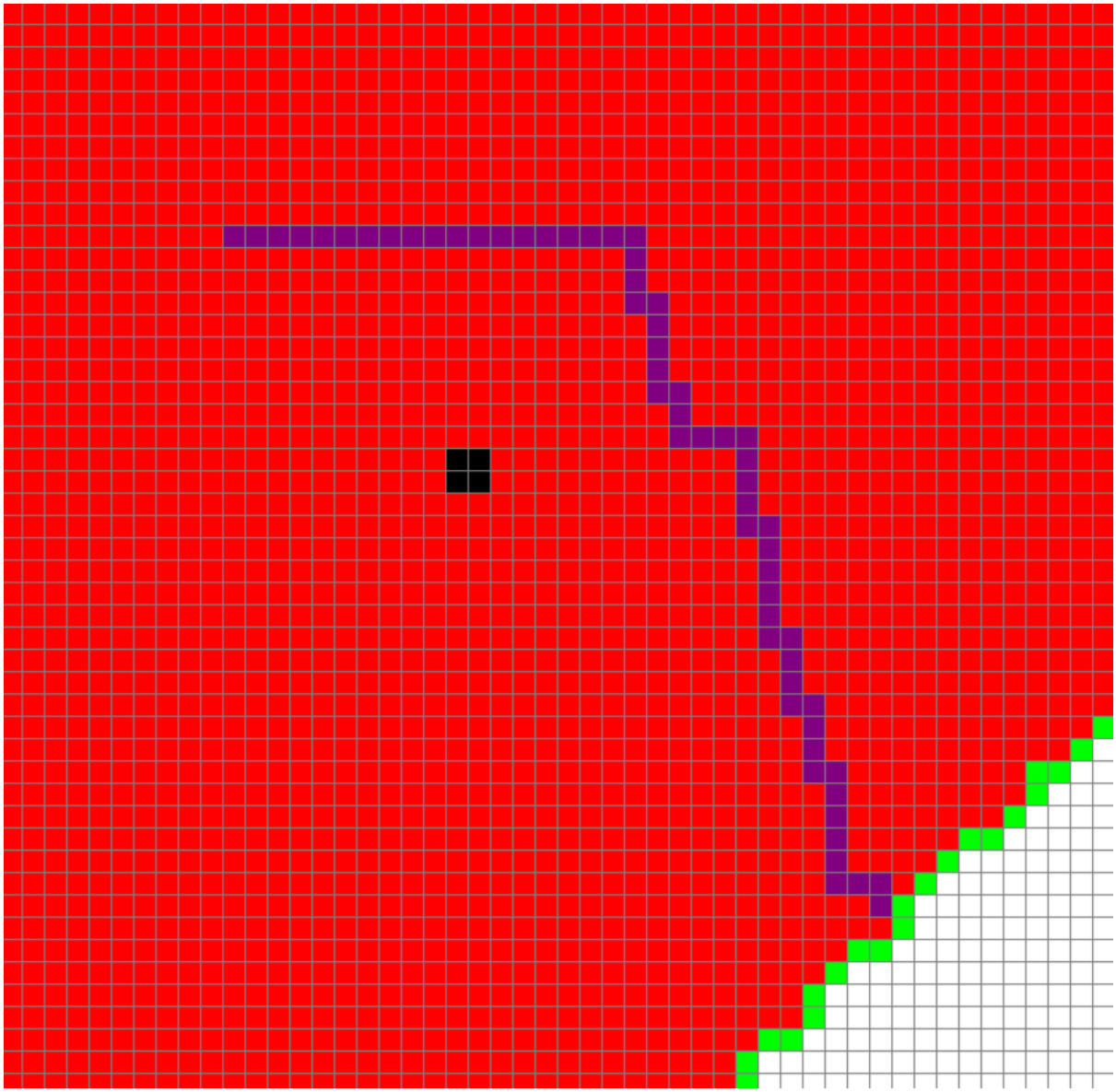
```python
class Vertex:
    def __init__(self, row, col, width, total_rows):
        self.row = row
        self.col = col
        self.x = row * width
        self.y = col * width
        self.color = WHITE
        self.neighbors = []
        self.width = width
        self.total_rows = total_rows
        self.distance = float("inf")  # Distance from start node
        self.previous = None  # Previous node in the shortest path

    def get_pos(self):
        return self.row, self.col

    def make_open(self):
        self.color = GREEN

    def make_closed(self):
        self.color = RED

    def make_path(self):
        self.color = PURPLE

    def update_neighbors(self, grid):
        self.neighbors = []
        if self.row < self.total_rows - 1 and not grid[self.row + 1][self.col].is_barrier():  # Down
            self.neighbors.append(grid[self.row + 1][self.col])
        if self.row > 0 and not grid[self.row - 1][self.col].is_barrier():  # Up
            self.neighbors.append(grid[self.row - 1][self.col])
        if self.col < self.total_rows - 1 and not grid[self.row][self.col + 1].is_barrier():  # Right
            self.neighbors.append(grid[self.row][self.col + 1])
        if self.col > 0 and not grid[self.row][self.col - 1].is_barrier():  # Left
            self.neighbors.append(grid[self.row][self.col - 1])

```

Algorithm Execution:

- **Initialization: The start node is initialized with a distance of 0, and all other nodes are set to infinity.**
- **Node Expansion: Each node's neighbors are explored, and if a shorter distance to a neighbor is found, the neighbor's distance and previous node reference are updated.**
- **Path Reconstruction: After reaching the end node, the algorithm backtracks from the end node to the start node using the previous references and marks the path in purple.**

Output Visualization:

The following image illustrates the result of the full implementation of the Dijkstra pathfinding algorithm. It shows how the algorithm computes the shortest path from the start point to the end point while avoiding obstacles. The grid representation highlights the different states of nodes, including the start, end, barriers, and the computed path, providing a clear visual of how Dijkstra operates in real-time. The visualization also demonstrates how Dijkstra's algorithm explores the grid by expanding nodes in order of increasing distance, ensuring that the shortest path is always found.



The full code implementation of the Dijkstra algorithm can be found on Github
https://github.com/paulmicky1/portfolio_gamesandai/blob/main/Dijkstraalgorithm.py

Reflection

Challenges Faced:

- Handling Large Grids: When using Dijkstra's algorithm on large grids, the algorithm can become computationally expensive. To mitigate this, optimization techniques such as reducing the grid size or using a more efficient priority queue structure can help improve performance.

- Correct Visualization: Ensuring that the nodes and their states (open, closed, path) are correctly visualized in real-time was crucial for debugging and understanding how the algorithm explores the grid.

Strengths and Weaknesses:

- **Strengths:**
    - Guarantees the shortest path from the start node to the end node.
    - Suitable for static maps where obstacles and start/end points are fixed.
    - Simple to implement and understand**.**
- **Weaknesses:**
    - Computationally expensive on large grids due to its exhaustive search.
    - Does not use a heuristic, meaning it explores all paths without prioritizing the most likely ones, which can make it slower compared to algorithms like A*.
    - Inefficient for dynamic environments where the grid changes frequently (e.g., moving obstacles).

3. Boids Flocking

Description

The Boids algorithm is a simulation technique used to model the flocking behavior of birds, fish, or other creatures that exhibit collective movement. It was first introduced by computer scientist Craig Reynolds in 1986 and is widely used in computer graphics and artificial intelligence to create natural and lifelike movements in groups of agents.

The algorithm is based on three simple steering behaviors that govern the movement of each agent (or "boid") in the flock:

1. **Separation:** Boids avoid crowding other nearby boids to maintain a comfortable distance.
2. **Alignment:** Boids align their velocity with the average velocity of their neighbors.
3. **Cohesion:** Boids move toward the average position of their neighbors to stay in the flock.

These behaviors, while simple, create complex and realistic group dynamics when applied to a large number of agents. The power of Boids comes from the fact that each

agent follows a set of local rules, and emergent behaviors arise from the interactions between these agents.

In real-time applications like video games or simulations, Boids is commonly used to simulate the movement of groups of animals, such as flocks of birds, schools of fish, or swarms of insects. The algorithm is effective because of its low computational cost, allowing for the simulation of large groups in real-time without heavy resource usage.

Implementation

The Boids algorithm can be implemented using basic vector math and a simulation loop. Here's a basic overview of how the algorithm is typically implemented:

- **Create a set of boids:** Each boid is an agent with properties such as position, velocity, and acceleration.
- **Calculate local forces:** For each boid, calculate the forces based on its neighbors. These include:
    - **Separation:** Avoid overcrowding by steering away from nearby boids.
    - **Alignment:** Align the boid's velocity with the average velocity of its neighbors.
    - **Cohesion:** Steer toward the center of mass of nearby boids.
- **Update the boid's position and velocity:** Apply the calculated forces to the boid's current velocity and update its position accordingly.
- **Render the simulation:** Each boid is drawn on the screen, and the simulation is updated on each frame.

Code Snippet:

```python
import pygame
import random
import math

# Define the Boid class
class Boid:
    def __init__(self, x, y):
        self.position = pygame.Vector2(x, y)
        self.velocity = pygame.Vector2(random.uniform(-1, 1), random.uniform(-1, 1))
        self.acceleration = pygame.Vector2(0, 0)
        self.max_speed = 4
        self.max_force = 0.1

    def apply_force(self, force):
        self.acceleration += force

    def align(self, boids):
        steer = pygame.Vector2(0, 0)
        total = 0
        for boid in boids:
            if boid != self and self.position.distance_to(boid.position) < 50:
                steer += boid.velocity
                total += 1
        if total > 0:
            steer /= total
            steer = steer.normalize() * self.max_speed
            steer -= self.velocity
            steer = self.limit(steer, self.max_force)
        return steer

    def cohesion(self, boids):
        steer = pygame.Vector2(0, 0)
        total = 0
        for boid in boids:
            if boid != self and self.position.distance_to(boid.position) < 50:
                steer += boid.position
                total += 1
        if total > 0:
            steer /= total
            steer -= self.position
            steer = steer.normalize() * self.max_speed
            steer -= self.velocity
            steer = self.limit(steer, self.max_force)
        return steer

    def separation(self, boids):
        steer = pygame.Vector2(0, 0)
        total = 0
        for boid in boids:
            distance = self.position.distance_to(boid.position)
            if boid != self and distance < 25:
                diff = self.position - boid.position
                diff /= distance
                steer += diff
                total += 1
        if total > 0:
            steer /= total
        if steer.length() > 0:
            steer = steer.normalize() * self.max_speed
            steer -= self.velocity
            steer = self.limit(steer, self.max_force)
        return steer

```

```
    def limit(self, vector, max_value):
        if vector.length() > max_value:
            vector = vector.normalize() * max_value
        return vector

    def update(self):
        self.velocity += self.acceleration
        self.velocity = self.limit(self.velocity, self.max_speed)
        self.position += self.velocity
        self.acceleration *= 0

    def edges(self, width, height):
        if self.position.x < 0:
            self.position.x = width
        if self.position.x > width:
            self.position.x = 0
        if self.position.y < 0:
            self.position.y = height
        if self.position.y > height:
            self.position.y = 0

    def flock(self, boids):
        separation = self.separation(boids)
        alignment = self.align(boids)
        cohesion = self.cohesion(boids)

        self.apply_force(separation)
        self.apply_force(alignment)
        self.apply_force(cohesion)

    def draw(self, screen):
        pygame.draw.circle(screen, (255, 255, 255), (int(self.position.x), int(self.position.y)), 3)


# Main function to initialize the simulation
def main():
    pygame.init()
    width, height = 800, 600
    screen = pygame.display.set_mode((width, height))
    pygame.display.set_caption("Boids Flocking Simulation")

    boids = [Boid(random.randint(0, width), random.randint(0, height)) for _ in range(100)]
    clock = pygame.time.Clock()

    run = True
    while run:
        screen.fill((0, 0, 0))

        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                run = False

        for boid in boids:
            boid.flock(boids)
            boid.update()
            boid.edges(width, height)
            boid.draw(screen)

        pygame.display.flip()
        clock.tick(60)

    pygame.quit()

if __name__ == "__main__":
    main()
```

Output Visualization:

The following image illustrates the result of the full implementation of the Boids flocking algorithm. It shows how the individual agents (boids) move together to simulate realistic flocking behavior. The boids dynamically adjust their velocity and position based on the separation, alignment, and cohesion rules, creating a fluid and lifelike movement pattern. The simulation includes the behavior of the boids as they interact with each other, maintaining a balance between staying close to the group (cohesion), aligning with the group's direction (alignment), and avoiding overcrowding (separation). The

result provides a clear visual of how the boids form a cohesive flock that moves as a unified entity in real-time, demonstrating emergent behaviors from simple local interactions.

The full code implementation of the boids flocking algorithm can be found on Github
https://github.com/paulmicky1/portfolio_gamesandai/blob/main/BoidsFlocking.py

Reflection

Challenges Faced:

- **Simulating realistic movement:** One challenge was ensuring that the movement of the boids felt natural and fluid. The parameters of the separation, alignment, and cohesion behaviors had to be fine-tuned to prevent the flock from either getting too spread out or too clustered.
- **Performance on large numbers of boids:** The performance of the simulation can degrade when simulating a large number of boids, especially with complex interactions. To mitigate this, optimizations like limiting the number of boids being considered at any given time (based on distance) were implemented.

Strengths and Weaknesses:

- **Strengths:**
    - **Realistic flocking behavior:** The algorithm produces lifelike flocking behavior with minimal computational resources.
    - **Scalability:** It can simulate a large number of agents in real-time with relatively low overhead.
    - **Emergent behavior:** The simplicity of the rules leads to complex, emergent behaviors that mimic real-world flocking.
- **Weaknesses:**
    - **Sensitivity to parameters:** The algorithm's performance is highly dependent on the chosen parameters for the steering behaviors. A slight change can drastically affect the behavior of the flock.
    - **Limited to 2D or 3D space:** While it's possible to extend Boids to three dimensions, the algorithm is typically implemented in 2D, limiting its applicability in more complex simulations or games.
    - **Collisions:** While the Boids algorithm models flocking behavior, it doesn't inherently handle collisions with obstacles, which would require additional logic to avoid crashes or more complex behaviors.

Conclusion

Implementing A*, Dijkstra's algorithm, and Boids flocking provided valuable insights into AI techniques for real-time games. Each method serves a unique role, making it essential to choose the right approach based on game requirements and computational constraints.

A* proved to be a highly efficient and scalable pathfinding algorithm, balancing accuracy and speed by using heuristics. However, its computational cost increases with larger maps, necessitating optimizations for extensive environments. In contrast, Dijkstra's algorithm guarantees the shortest path by systematically evaluating all possible routes, making it ideal for turn-based strategy games. Although slower, it excels in scenarios where precision is more important than speed.

Boids flocking, unlike pathfinding algorithms, focuses on emergent behaviors, enabling lifelike and adaptive group movements. This technique is widely used in simulating crowds, enemy formations, and NPC behavior, adding realism to game environments. Fine-tuning separation, alignment, and cohesion is crucial to maintaining fluid

movement, and optimization techniques such as spatial partitioning improve efficiency in large-scale simulations.

Beyond gaming, these AI techniques have broader applications in robotics, simulations, and real-world optimization. A* is commonly used in autonomous navigation, Dijkstra's algorithm aids network routing, and Boids flocking contributes to crowd dynamics and drone swarm coordination. Understanding these methods strengthens AI development skills and enhances problem-solving in various domains.

In conclusion, selecting the appropriate AI technique depends on game complexity, performance constraints, and desired realism. Mastering these algorithms allows developers to create intelligent, responsive, and engaging game experiences. As AI technology advances, integrating machine learning and adaptive AI models will further enhance game behavior, leading to more immersive and dynamic gameplay.