Project 1.2
Department of Advanced Computing Sciences
Maastricht University
Academic Year 2023/2024
Maastricht, Netherlands

Crazy Putting!

Group KEN17: Alexandra Afanasiuc, Anna Balañá Zemanaj, Jakub Bujak, Olaf Deckers, Przemysław Grudka, Jagoda Jakuczun and Nina Żórawska

Table of contents

Abstract

The goal of this report is to show how we can implement a golf game simulation that combines physics, mathematical modelling, and artificial intelligence. This report addresses difficulties of accurately predicting the motion of a golf ball's over sloping terrains and developing AI models for making decisions on the terrain. To simulate the physics of golf balls, differential equations are solved using Euler and 4th order Runge-Kutta methods to create equations of motion. AI bots developed with efficiency and pathfinding algorithms were used to navigate the golf courses and score hole-in-one shots and to complete maze-like courses. The primary findings show how the RK4 technique offers more precision but is computationally demanding, whereas Euler's method is

quicker but not as precise. AI bots operated far more effectively and accurately compared to rule-based bots. Also, for maze-like courses, the complexity and layout of the maze has a significant impact on the efficiency of the algorithm. These results show that artificial intelligence can work effectively in game production and it can hold the potential for developing independent decision-making systems.

1. Introduction

The purpose of this project is to develop a simulation environment and artificial intelligence models for a golf game. The problem is relevant within the scientific field as it integrates principles from physics, numerical analysis and AI-based decision making, which are key areas in data science and artificial intelligence fields. Accurately modelling the motion of the golf ball on different surfaces and terrains with various obstacles is an interesting problem, because of complex interactions between forces like gravity and friction. Additionally, incorporating an artificial intelligence model showcases the practical application of AI in game development and contributes to advancements in autonomous decision-making systems. During our research we used Java programming language and LibGDX library for our Graphical User Interface (GUI).

The research questions this project aspires to answer include:

1. How can we accurately simulate the physics of a golf ball on a sloping course with obstacles using differential equations?
2. What are the optimal numerical methods for solving these ordinary differential equations in the context of this problem?
3. How can we design and implement an AI system capable of navigating the simulated courses to reach the goal in one shot and handling complex maze-like courses?

Differential equations have been widely used to model physical systems. For example the Lotka-Volterra equations [1] for imitating population dynamics and the SIR-model [2] for predicting the spread of infectious diseases. But, the application of differential equations in golf simulation is less explored and they often lack the integration of advanced numerical systems for the most accurate simulations. We would like to incorporate artificial intelligence in our system.

The primary elements of this research are numerical methods for solving differential equations, that have complicated analytical solutions or are impossible to solve analytically. The equations chosen by us include the Euler method and the Runge-Kutta method of 4^{th} order [3]. To develop AI bots to complete the courses, different algorithms were used, involving a rule-based systems, optimization algorithms like Gradient descent [4] and pathfinding algorithms like A* [5]. These algorithms were chosen for their computational efficiency and accuracy for solving the golf in the least number of steps.

The structure of this report is intended to provide an extensive overview of the methodology and findings. Following that, the report delves into implementation of a physics system, technical development of the differential equation solvers and bots. It is followed by a series of experiments and their evaluations to assess performance and accuracy of different numerical methods and various bots. The last part looks at the impact of those findings and proposes future research topics.

2. Physics

The physics implemented in the simulation of golf ball motion is derived from Newton's laws of motion. The primary forces considered are gravitational force, normal force, and frictional forces. The following sections elaborate on these forces, their effects, and the equations governing the ball's motion.

2.1. Primary Forces

1. **Gravitational force:** This constant force acts downward, toward the centre of the Earth, and is given by: $G = -mg$, where m is the mass of the ball (0.5 kg) and g is the acceleration due to gravity (9.80665 m/s^2).
2. **Normal force:** This force is exerted by the green surface on the ball and is perpendicular to it. The magnitude and direction of the normal force change dynamically as the ball moves across different slopes, altering its acceleration along the surface. This is done by the use of the unit vector. The unit vector normal to the surface is:
$$\hat{n} = \frac{\nabla h}{|\nabla h|}.$$
where $h(x, y)$ is the height of the terrain at point (x, y) and ∇h represents the gradient of the height function.
3. **Force friction:** friction opposes the motion of the golf ball and is dependent on the normal force. The are two types of frictional forces to consider:
    a. **Kinetic friction:** Occurs when the ball is in motion. It acts opposite to the direction of movement and is proportional to the normal force, which is assumed to be equal to the component of the gravitational force perpendicular to the surface.
    $$\vec{F}_{\text{friction}} = -\mu_k mg\frac{\vec{v}}{|\vec{v}|}$$
    where $\mu_k$is the kinetic friction coefficient and $\vec{v}$ is the velocity of the ball.
    b. **Static friction:** Acts when the ball is at rest and prevents it from starting to move under the influence of gravitational and normal forces, especially on a slope. The maximum static friction is given by:
    $$\vec{F}_{\text{friction}} = -\mu_s mg\frac{\nabla h}{|\nabla h|}$$
    where $\mu_s$is the static friction coefficient.

2.2. Equations of Motion

The motion of the ball across the course is governed by the following differential equations, assuming the ball is always in contact with the terrain and neglecting any rolling or flying motion. The horizontal accelerations $a_x$and a_z are given by:

$$a_x = \frac{F_{\text{gravity},x} + F_{\text{friction},x}}{m} = -g \cdot \frac{\partial h}{\partial x} - \mu_k g\frac{v_x}{|\vec{v}|}$$
$$a_z = \frac{F_{\text{gravity},z} + F_{\text{friction},z}}{m} = -g \cdot \frac{\partial h}{\partial z} - \mu_k g\frac{v_z}{|\vec{v}|}$$

2.3. Approximations

- The terrain slopes are assumed to be gentle, allowing the use of linear approximations.
- The rolling and flying motions of the ball are neglected.
- The normal force is assumed to be equal to the component of the gravitational force perpendicular to the surface.

The most important parameters used in the simulation can be adjusted by the user through a settings menu, allowing changes to values such as kinetic and static friction coefficients on different surfaces (grass and sand) and the target radius. These adjustments are made within the bounds provided in the project manual to ensure realistic simulation results. In table 2.1 you can find an overview of all the parameters and restrictions.

Quantity	Symbol	value/range	restrictions
Course profile	$h(x,y)$	-10m - 10m	$\left\lvert\frac{\partial h}{\partial x}\right\rvert, \left\lvert\frac{\partial h}{\partial y}\right\rvert \leq 0.15, \left\lvert\frac{\partial^2 h}{\partial x^2}\right\rvert, \left\lvert\frac{\partial^2 h}{\partial x \partial y}\right\rvert, \left\lvert\frac{\partial^2 h}{\partial y^2}\right\rvert \leq 0.1/\text{m}$
Mass of the golf ball	m	0.0459 kg	
Gravitational constant	g	9.81 m/s^2	
Kinetic friction (grass)	μ_K	0.05-0.1	
Static friction (grass)	μ_S	0.1-0.2	$\mu_S > \mu_K > 0$
Kinetic friction (sand)	$\mu_{K,s}$		$0 < \mu_K < \mu_{K,s} < 1$
Static friction (sand)	$\mu_{S,s}$		$0 < \mu_{K,s}, \mu_S < \mu_{S,s} < 1$
Maximum speed	$v_{\max}$	5 m/s	
Target radius	r	0.05m - 0.15m	

Table 1. Physics parameters and restrictions.

3. ODE Solvers

Understanding and implementing the physical forces on a golf ball for realistic motion involves using numerical solvers to tackle complex differential equations. To accurately simulate the movement of a golf ball, we implemented two different numerical algorithms. These algorithms are crucial for solving the system of ordinary differential equations derived from the forces acting on the golf ball. Methods like Euler's and Runge-Kutta of 4$^{\text{th}}$ order allow us to approximate solutions with desired accuracy by adjusting the step size.

State Vector

The state vector is a fundamental component used in both Euler and the Runge-Kutta methods. It represents the system at a given time and includes all the relevant state variables. The state vector is used in the physics engine for simulating the ball movement and forces on the ball. In vector terms, let's consider the state vector $x(t)$ representing the system at time t:

$$\vec{x}(t) = \begin{pmatrix} x(t) \\ z(t) \\ v_x(t) \\ v_z(t) \end{pmatrix}$$

3.1. Euler's Method

Euler's method is a straightforward numerical procedure for solving ODEs with a given initial value. This method is particularly useful for providing approximate solutions to ODEs where analytical solutions are difficult or impossible to obtain and the solutions do not have to be precise. The method is based on the premise of using the slope of a function to predict its values at subsequent points.

System of ODEs

The system of ordinary differential equations (ODEs) describes the evolution of the state vector over time:

$$\frac{d\vec{x}}{dt} = \vec{f}(\vec{x}(t))$$

- $\vec{x}(t)$: State vector at time t.
- $\frac{d\vec{x}}{dt}$: Time derivative of the state vector, representing the rate of change of the state.
- $\vec{f}(\vec{x}(t))$: Function that defines dynamics of the system, taking state vectors as input.

Update rule

$$\vec{x}(t+h) = \vec{x}(t) + h \cdot \vec{f}(\vec{x}(t))$$

- h: Step size
- $\vec{x}(t+h)$: State vector at the next time step.
- $\vec{x}(t)$: State vector at the current time step.
- $\vec{f}(\vec{x}(t))$: Rate of change of the state vector at the current time step.

Euler's method is simple and intuitive and its accuracy depends on the size of the step. We expect smaller steps to lead to more accurate results and we test that in our experiments described in Section 5 of the paper. The method is also prone to accumulating errors, especially over large intervals, because each step is based on the previous step's approximate value.

3.2. The Runge-Kutta Method

The Runge-Kutta method extends the concept of Euler's method by incorporating multiple slope estimates to find the approximation of the solution. Specifically, the 4th-order Runge-Kutta method (RK4) calculates the next value by averaging several intermediate slope estimates, enhancing the stability and precision of the solution at each step.

Formula:

$$\mathbf{k}_1 = h \cdot \mathbf{f}(\mathbf{x}(t))$$
$$\mathbf{k}_2 = h \cdot \mathbf{f}(\mathbf{x}(t) + \frac{1}{2}\mathbf{k}_1)$$
$$\mathbf{k}_3 = h \cdot \mathbf{f}(\mathbf{x}(t) + \frac{1}{2}\mathbf{k}_2)$$
$$\mathbf{k}_4 = h \cdot \mathbf{f}(\mathbf{x}(t) + \mathbf{k}_3)$$
$$\mathbf{x}(t+h) = \mathbf{x}(t) + \frac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4)$$

Variables Explained:

- $x(t)$: State vector at time t.
- h: Step size.
- $f(x(t)$: Rate of change of the state vector at the current time step.
- k_1, k_2, k_3, k_4 : Intermediate slope calculations in the RK4 method, each providing an estimate of the slope of x(t) at different points within the interval $[t, t + h]$.

The Runge-Kutta method provides a balanced approach between computational effort and solution refinement. It involves multiple slope estimates within each step, allowing for a detailed approximation of the solution. This method is applicable to a wide range of problems, requiring several function evaluations per step to achieve its estimates. The process includes averaging intermediate slopes to enhance the solution's accuracy without necessarily decreasing the step size.

4. Bot Design

For the purposes of our research we want to design three types of bots for a golf simulation: a rule based bot, an artificial intelligence bot and an improved AI bot for maze-like structures. The three bots differ fundamentally in their operational logic: one follows a set of predefined rules, and the others employ advanced artificial intelligence techniques. These bots interact with a simulated golf game that includes varied terrain and slopes. The primary objective for the rule based and AI bot is to consistently achieve a hole in one when possible, contrasting with the basic bot's goal to reach the target without paying attention to the number of moves used. The primary objective for the A* algorithm is to reach the target of a maze-like structure.

4.1. Rule Based Bot Design

The Rule-Based Bot is designed to follow a series of rules that determine its behaviour. Our bot is equipped with a full knowledge about the golf course. The decisions it makes will be based on the shape and slope of the terrain as well as the positions of both the golf ball and the target. The bot's decision-making mechanism focuses on identifying the best shot direction and power based on the distance to the hole and the slope of the terrain. This entails retrieving parameters such as height and slope from a data structure describing the terrain, as well as conducting geometric and trigonometric computations to calculate the optimal direction for the shot.

Here we can find an overview of how our algorithm works:

- **Initialization:** Initialise the bot with the golf ball, target position, target radius, physics engine, game rules and retrieve the height and slope at the ball's current position.
- **Establish direction:** This is done by the following steps:

- Calculate vector from the target to the ball with the following formula:

$$\text{vectorBT} = \begin{pmatrix} \text{ballPosition.x} - \text{targetPosition.x} \\ \text{ballPosition.z} - \text{targetPosition.z} \end{pmatrix}$$

- Compute the vector's magnitude, which is the length or size of the vector by:

$$\text{magnitude} = \sqrt{(\text{vectorBT.x})^2 + (\text{vectorBT.y})^2}$$

- Normalise the vector:

$$\text{direction} = \begin{pmatrix} \frac{-\text{vectorBT.x}}{\text{magnitude}} \\ \frac{-\text{vectorBT.y}}{\text{magnitude}} \end{pmatrix}$$

- The final direction vector has a magnitude of 1 and points from the ball to the target

- **Rules followed by the bot to make the shot:**
  - If distance ≤ 1, use a force equal to magnitude $+ 1$.
  - If distance > 1 and < 7, use a force equal to magnitude $\square$ 1.5.
  - If distance > 7, use a force equal to magnitude $\square$ 0.75.
  - If slope > 0.2, apply additional force of a size 10 $\square$ slope.
- **Execution**: Update the ball's position and velocity using the physics engine and adjust its vertical position according to the terrain height.
- **Target Check**: Determine if the ball has reached the target within the specified radius by checking its proximity in x and y directions.

We defined the 4 rules followed by our bot by a trial and error method. The first set of rules takes into account how far away from the target the ball is. We decided to divide the possible distance into 3 categories: small (0,1], medium (1,7) and big [7,∞), arriving at those specific thresholds, by testing how the bot behaves on different terrains. When the distance is small we enlarge the force by 1 unit, so to get closer to the target, but not overshoot. When the distance is greater, then the bot uses a smaller force with respect to the distance. This is because on simple, not steep terrains it would cause the bot to overshoot and possibly fall out of the map. For the medium distance we arrived at the middle ground and decided to increase the calculated force by 50%. The last rule helps us deal with steeper hills. When a slope is big enough, the applied force needs to be big enough to overcome the hill.

4.2. AI Bot Design

The AI bot uses an algorithm called Adam optimization [6], which constantly adjusts the golf ball's velocity to minimise the distance between the target and the golf ball after each shot. Adam improves convergence speed and accuracy by combining adaptive learning rates with moment estimates. The bot tries to find the global minimum by iteratively searching for the lowest distance from the target position, treating the terrain as a cone and moving downward. If it finds the global minimum or reaches the maximum number of iterations, it stops, having found the optimal solution. This strategy results in a sophisticated and precise system for guiding the golf ball to the target position.

Here we can find an algorithm overview:

- **Initialisation:** Assign initial variables, such as moments for the Adam optimizer, learning rates, and epsilon values for gradient descent.

- **Deviation Calculation:** Evaluate how far the ball lands from the target (deviation) for a given velocity using the Euclidean distance formula. This is done by performing simulations of the shoots (to get the position of the ball with assigned velocities), and then calculating the deviation from the target position. The following formula is used:

$$d(q,p) = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}$$

- **Gradient Approximation:** Estimate the gradient of the deviation function with respect to the velocity. The gradient indicates how much the deviation would change if the velocity is slightly adjusted. We do not calculate the exact derivative of the function we are minimising because this would not consider terrain changes. Instead, we add small perturbations (ϵ) to the velocities (x and z) separately and simulate a shot with the given velocities to calculate a valid outcome. This is done using the following formula:

$$\nabla f(v) \approx \frac{\|\text{deviation}(v+\epsilon)\| - \|\text{originalDeviation}\|}{\epsilon}$$

- **Moment and Velocity Update:** Update velocity using Adam's adaptive learning rates. This is done with the following formulas:
  - **First moment estimate:**

$$m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t$$

This represents an average of the past gradients at iteration t, where β_1 is the decay rate for the first moment estimate, m_{t-1} is the previous first moment estimate, and g_t is the gradient at iteration t. It helps smooth out the gradients to avoid drastic changes in velocity.

  - **Second moment estimate:**

$$v_t = \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t^2$$

This represents the average of the squared gradients at iteration t, with β_2 as the decay rate for the second moment estimate, v_{t-1} as the previous second moment estimate, and g_t^2 as the square of the gradient at iteration t. It captures the variability of the gradient, helping to adjust the learning rate adaptively.

  - **Bias Correction:**
    Since m and v are initialised to zero at the beginning, they are biassed towards that value in the initial iterations. We correct this by performing the following calculations:

    - **bias-corrected first moment:**

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

This corrects the bias in m_t by dividing by $1-\beta_1^t$, which accounts for the decay rate β_1 raised to the power of the current iteration t. The correction ensures that the moment estimates are unbiased, leading to more accurate updates.

    - **bias-corrected second moment:**

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Similarly, this corrects the bias in v_t by dividing by $1 - \beta_2^t$, with $\beta_2$being the decay rate raised to the power of iteration t. This ensures that the second moment estimate accurately represents the variability of the gradient.

- **Velocity update:**

$$v_{t+1} = v_t - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

This formula adjusts the velocity v_{t+1} at iteration $t + 1$ by subtracting α, the learning rate, multiplied by the bias-corrected first moment $\hat{m}_t$divided by the square root of the bias-corrected second moment $\hat{v}_t$ plus a small constant ϵ to prevent division by zero. This ensures faster convergence to the target.

- **Shot Execution**: Ensure the velocity does not exceed a maximum value. If it does not exceed, then the shot is executed.

4.3. A Star Search Algorithm for Improved AI Bot

The A* search algorithm is integrated to help the AI bot navigate maze-like courses effectively. It finds the shortest path from a starting point to the target using a combination of the actual cost from the start to a given node and a heuristic estimate of the cost from that node to the goal. The function is defined as:

$$f(n) = g(n) + h(n),$$

where g(n) is the cost from the start node to node n, h(n) is the heuristic estimate of the cost from node n to the goal and f(n) is the total estimated cost of the cheapest solution through node n. The heuristic function helps prioritise nodes that are closer to the goal:

$$h(a, b) = |x_a - x_b| + |y_a - y_b|,$$

where x_a and y_a are the coordinates of node a, and x_b and y_b are the coordinates of node b.

The course is represented as a grid of nodes, each node corresponding to a point on the course and storing its coordinates and associated costs. The node with the lowest f(n) is selected from an open set and the algorithm examines each selected node's neighbours, calculating g scores and updating them if a cheaper path is found. Updated neighbours are reinserted into the open set. Once the goal node is reached, the algorithm reconstructs the path by tracing back from the target to the start using stored parent references.

When the player initiates a shot, the game uses the A* algorithm to calculate the best path from the ball's current position to the target and segments said path to determine intermediate positions for the ball's movement. By overriding the target position of the AI bot, we make it follow the desired path until it reaches the end.

5. Experiments

5.1. ODE Solvers Experiments

Testing different numerical methods is necessary, as they are a fundamental base of the rest of our research. With this section of experiments we hope to answer our second research question: "What are the optimal numerical methods for solving these ordinary differential equations in the context of

this problem?". Since our goal is to create an accurate golf simulator and efficient bots, we decided to compare the Euler and RK4 methods in terms of computational time and errors in approximation. Both experiments were conducted using the following system of differential equations:

$$da/dt = b$$
$$db/dt = 6a - b$$

with initial conditions: $a(0.0) = 1, b(0.0) = 0$
on the time interval $t = [0.0,1.0]$
for different step sizes: $h1 = 0.1, h2 = 0.01, h3 = 0.001$

Conducting the experiments with different step sizes is important, because we can correlate changes in step sizes with changes in time and accuracy. This can help us make informed decisions, when adjusting the solvers for the next part of our project.

5.1.1. Experiment - Computation Time of Euler and RK4 methods

Because we want our game to run smoothly, we need to have a fast method to solve complicated differential equations. To test how both approaches perform and to compare them we created new experiment methods for both solvers. In them we call the particular solver multiple times with the same system of equations. We do it three times, one time for a different step size. As our solvers calculate the solution, we record how long it takes them to finish, by using a System.nanoTime() Java function [7].

5.1.2. Experiment - Root Mean Squared Error of Euler and RK4 methods

The other aspect that we need to take into account, when comparing those two numerical methods is how accurate they can be. The objective of the simulation is not only to be fast, but it also needs to closely follow real-life physics. To assess a method's accuracy, we first created a method that calculates an analytical solution for this particular system of differential equations. We solved the system by hand to get a function, which could be used for obtaining a precise solution:

$$a(t) = C_1 e^{2t} + C_2 e^{-3t}$$

with $C1 = 0.6$ and $C2 = 0.4$.

We test the accuracy by calculating a root mean squared error (RMSE) of the entire evolution of the first variable, in the system, "a" in time. RMSE is calculated using the following formula:

$$\mathrm{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

where n - number of points, yi - actual values, y^i - approximated values.

5.2. AI Bot VS Rule-Based Bot Experiments

5.2.1. Experiment - "AI Bot vs Rule-Based Bot - Number of Shots Taken to Reach Target"

It is important to know which bots are the most accurate for our golf simulation, because it impacts the overall effectiveness and efficiency of the game experience. To determine this. We compare the two bots that we have created by computing how many shots they require to reach the target. These bots are the rule-based bot and the AI bot. 4 terrains were chosen to represent different difficulty

levels. Starting from a flat surface and ending on a terrain covered with hills. The four different conditions are stated in Table 2. We conducted this experiment by counting every instance the bots make a shot and stop when the target is reached.

5.2.2. Experiment - "Comparison of Time to Reach Target for Different Bots"

In this experiment we decided to compare how our two bots perform, when tested on different terrains. To ensure clarity and consistency, we conducted the experiment on the settings we used in Experiment 1. The four different conditions we tested are stated in Table 2. To obtain the results we created two variables: "startTime" and "endTime". We save the "startTime" once the bot starts looking for the solution and save the "endTime" once the ball reaches the target. In the end we calculate the difference between both time stamps. Because performance of our AI Bot can vary slightly, even on the same terrain, we decided to repeat its shot 3 times, under the same conditions, and obtain an average out of those results.

Test number	Terrain function	X coordinate target	Y coordinate target	Radius target	Initial position of the ball on the x-axis	Initial position of the ball on the y-axis
1	5	0	0	0.5	25	25
2	sqrt ((sin (0.1 * x) + cos (0.1 * y)) ^ 2) + 0.5 * sin (0.3 * x) * cos (0.3 * y)	4	1	0.5	15	20
3	0.4 * cos x + 10	0	0	0.5	10	10
4	0.4 * cos x + 0.6 * sin y + 10	0	0	0.5	10	10

Table 2. Test conditions for Experiments 5.2.1 and 5.2.2

5.2.3. Experiment - "AI Bot vs Rule-Based Bot - Time to Reach Target from Different Distances"

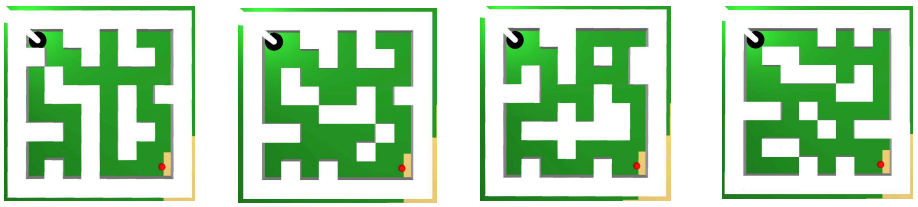
In the last experiment that focuses on differences between our bots, we decided to look at how the performance time changes, when we change the initial distance from the ball to the target. To conduct the experiment, we set the target to be at (0,0) position with radius equal to 0.5 and performed 4 tests. All of them were executed under the exact same conditions, on a flat surface (function used – "5"). The only factor that changed was the starting position of the ball. We begin at (10,10), increase the coordinate values by 10 each time and end at (40,40). To measure the time we used the same approach as in Experiment 2.

5.3. A Star Algorithm experiments

5.3.1. Experiment - "A Star Algorithm - Time to Reach Target in Different Mazes"

To test our approach for designing an AI bot capable of solving a maze-like course we decided to compare the time of reaching the target by the A Star Bot, in different mazes. To conduct the

experiment, we set the coordinates of the target to (1,1) with the radius equal to 0.5 and calculated the average of three tests in each maze. The mazes differ in the number of walls and their positioning, whilst the flat surface (function used - "1") and the initial position of the golf ball (8,8) stay the same. The mazes are constructed by dividing the map in a grid-like pattern. Each entry in an array is a small 1x1 square. 0 stands for no wall, whilst 1 for a wall object. Mazes 1, 2, 3 and 4 we tested the algorithm on were defined as follows:



1. 2. 3. 4.

5.3.2. Experiment - "Performance of Different Bots on Maze-Like Courses"

At the end we wanted to compare the performance of all the bots we have created to this point under the most complicated conditions - in mazes. We used the mazes from Experiment 5.3.1. and applied identical conditions. We want to see which bots will be able to reach the target and which will fail.

6. Results

6.1. Results of ODE Solvers Experiments

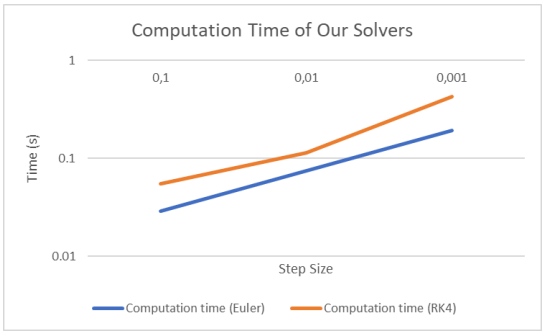


Figure 1. Computation time on both methods.

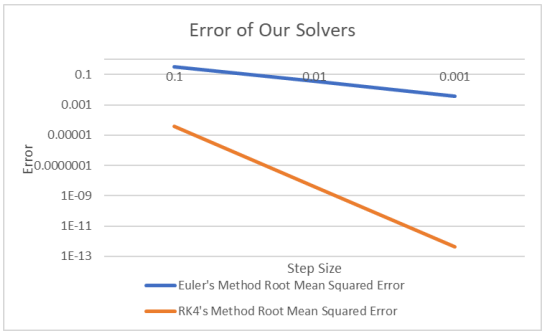


Figure 2. RMSE in both methods with respect to the step size.

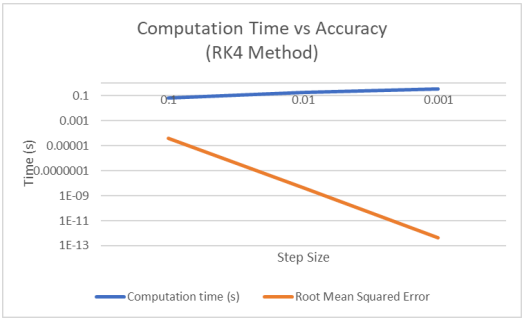


Figure 3. Computation time of RK4 method.

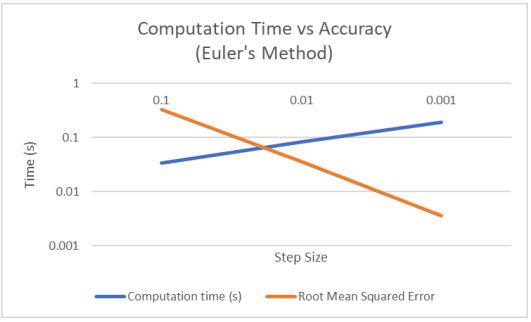


Figure 4. Computation time of Euler's method.

6.2. Results of AI Bot VS Rule-Based Bot Experiments

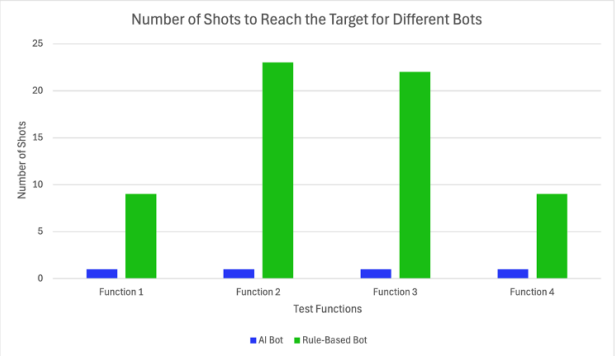


Figure 5. Experiment - “AI Bot vs Rule-Based Bot - Number of Shots Taken to Reach Target”

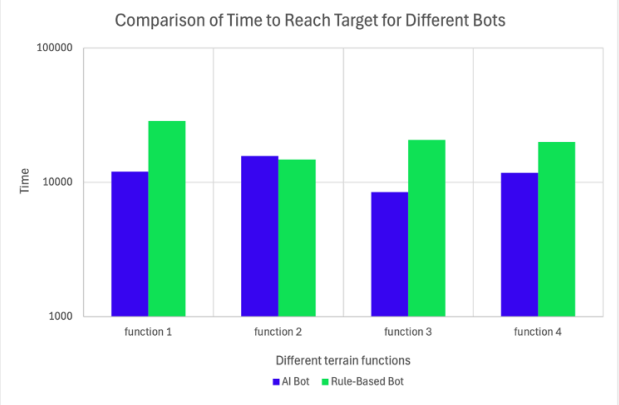


Figure 6. Experiment - “Comparison of Time to Reach Target for Different Bots”

Exact Results for the bots for Function 2

Number of the Test	AI Bot Time in Milliseconds	Rule-Based Bot Time in Milliseconds
Test 1	13235	14787
Test 2	19905	-
Test 3	14104	-

Table 3. Experiment -“Comparison of Time to Reach Target for Different Bots” - Exact results for function 2

Figure 7. Experiment - “AI Bot vs Rule-Based Bot - Time to Reach Target from Different Distances”

6.3. Results of the A Star Algorithm Experiments

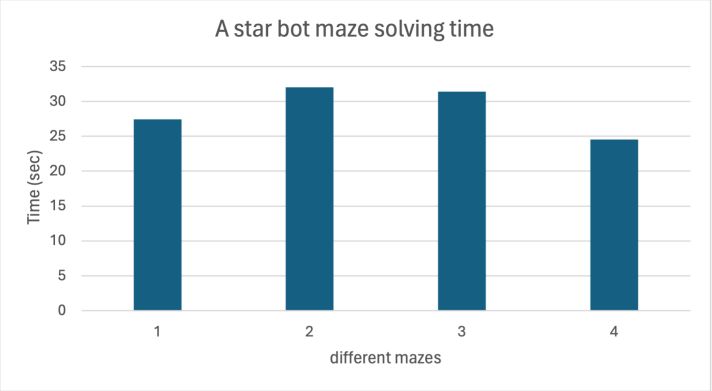


Figure 8. Experiment - “A Star Algorithm - Time to Reach Target in Different Mazes”

Maze	Rule Based Bot	AI Bot	A Star Bot
1.	Fail	Fail	Win
2.	Fail	Fail	Win
3.	Fail	Fail	Win
4.	Win	Fail	Win

Table 4. Experiment - “Performance of Different Bots on Maze-Like Courses”

7. Discussion

Analysing the results reveals that the RK4 method is generally more precise across varied step sizes, making it better suited for applications that demand accuracy. This is clear from the lower error levels found. Euler's method, on the other hand, could be used for rapid estimates in which high precision is not required. However, the RK4 approach is more computationally demanding than Euler's method, implying that while it provides greater precision, it consumes more processing power and time, particularly at smaller step sizes. Euler's approach is faster at bigger step sizes, but it loses accuracy more dramatically as step sizes decrease.

As our results show, our AI Bot generally needs less time to finish the game. On 3 out of 4 different terrains, it performed better. Not only did it manage to score the ball quicker, but it always achieved the goal in one shot. The Rule-Based Bot always needed at least 9 shots to complete the game and in 75% of cases needed more time to reach the target. The bad performance of the Rule-Based Bot may be a result of hard-defined rules, which are followed blindly by the bot, no matter the environment it is playing in. Although the magnitude of the force the Rule-Based Bot uses is influenced by the slope of the terrain surrounding the ball, this bot doesn’t take into consideration how the entire map looks.
In one case – Test Case 2 – the Rule-Based bot was faster. It may be due to the fact that sometimes the AI is not consistent. As we can see in Figure 6, in 66% of the tests the AI Bot was faster than the Rule-Based Bot. But because sometimes it can take the AI Bot more time to find the solutions and it lowers down its score, often it is going to perform better.
For both bots, the time needed to reach the target increases as we place the ball further from the target, but the AI Bot always shows better performance. The time for the AI Bot increases linearly from the beginning. For the Rule-Based Bot, there is a clear moment where the slope of the function changes, and from that point, the time starts to grow more rapidly with the distance. When

we arrive at the distance equal to 50, the Rule-Based Bot fails to score the target and loses the game.

Our results suggest several insights into the A* bot's performance. Firstly, the time taken by the bot to reach the target varied significantly across different mazes. Maze 4 had the shortest average time (24.56334667 sec), while Maze 2 had the longest (32.03258 sec). This shows that the complexity and layout of the maze have a significant impact on the efficiency of the algorithm. Secondly, the significant difference in times between the mazes highlights areas for potential optimization of the algorithm.
For instance, analysing why Maze 4 allowed for the fastest time could reveal strategies or heuristics that can be applied to improve performance in more complex mazes like Maze 2. Maze 4 was also the only maze that any of the first two bots were able to solve. One of the reasons for both of those things can be the fact that the shortest path to the target in Maze 4 is an almost perfect straight line. This works in favour of the Rule-Based Bot that makes each shot in the exact direction of the target. A short path also explains why the improved AI Bot required less time to solve this maze.

8. Conclusion

This study aimed to develop a thorough simulation environment for a golf game by integrating physics, mathematical modelling, and artificial intelligence. The initial research questions focused on accurately simulating the physics of a golf ball, identifying optimal numerical methods for solving the relevant ordinary differential equations, and designing an AI capable of navigating the simulated environment. Our findings demonstrated that the 4th order Runge-Kutta method (RK4) provided exceeding accuracy in predicting the golf ball's motion compared to Euler's method, although it required more time for computations. The AI bots, using advanced algorithms such as A*, significantly outperformed the Rule-Based Bots in terms of efficiency and accuracy in navigating the courses.

The implications of this study are numerous. Firstly, it verifies the effectiveness of AI and advanced numerical methods in simulating complex physical systems. The superior performance of AI bots over rule-based bots highlights the potential that AI methods have in improving decision-making processes in dynamic and changing conditions. This finding contributes to various fields, besides game development, that require autonomous decision-making systems. Additionally, the study confirms that the RK4 is an optimal method for solving differential equations in this problem and demonstrates the design of a physical system reflecting real-world conditions and an AI system capable of navigating that environment.

Future research could explore several possibilities to build on the findings of this project. Enhancing the RK4 method or developing an approach combining RK4's accuracy and Euler's speed could significantly improve our project. Further development of the AI bots, by making them adaptable to a wider variety of obstacles and terrains, as well as exploring machine learning techniques, could enhance their performance. Finally, expanding our simulation to incorporate more complex environmental factors and real-world physics, such as adding the rolling motion of the ball, would provide a more realistic gaming experience and have broader applications where accurate physics imitation is crucial.

9. References

[1] Ellner, S. P., & Guckenheimer, J. (2006). *Dynamic models in biology*. Princeton University Press.

[2] Diekmann, O., & Heesterbeek, J. A. P. (2000). Mathematical epidemiology of infectious diseases: Model building, analysis and interpretation. John Wiley & Sons.

[3] Yenesew Workineh, Habtamu Mekonnen, & Basaznew Belew. (2024). Numerical methods for solving second-order initial value problems of ordinary differential equations with Euler and Runge-Kutta fourth-order methods. *Frontiers in Applied Mathematics and Statistics*, *10*. https://doi.org/10.3389/fams.2024.1360628

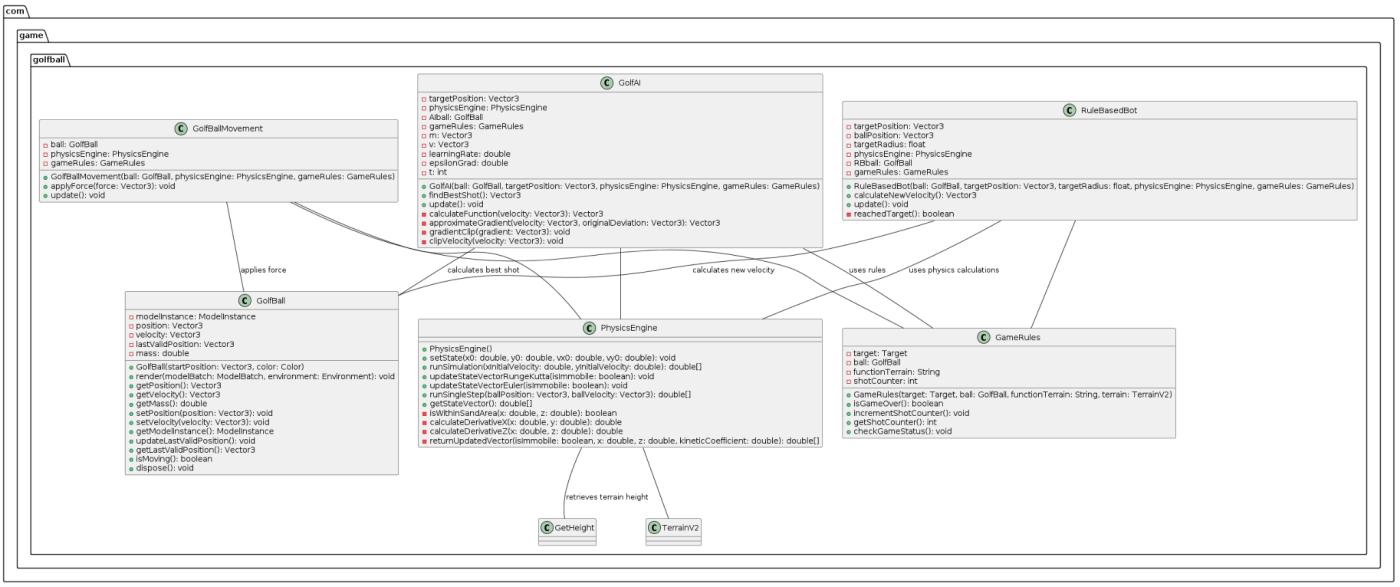
[4] Nonhoff, M., & Müller, M. A. (2020). Online Gradient Descent for Linear Dynamical Systems. *IFAC PapersOnLine*, *53*(2), 945–952. https://doi.org/10.1016/j.ifacol.2020.12.1258

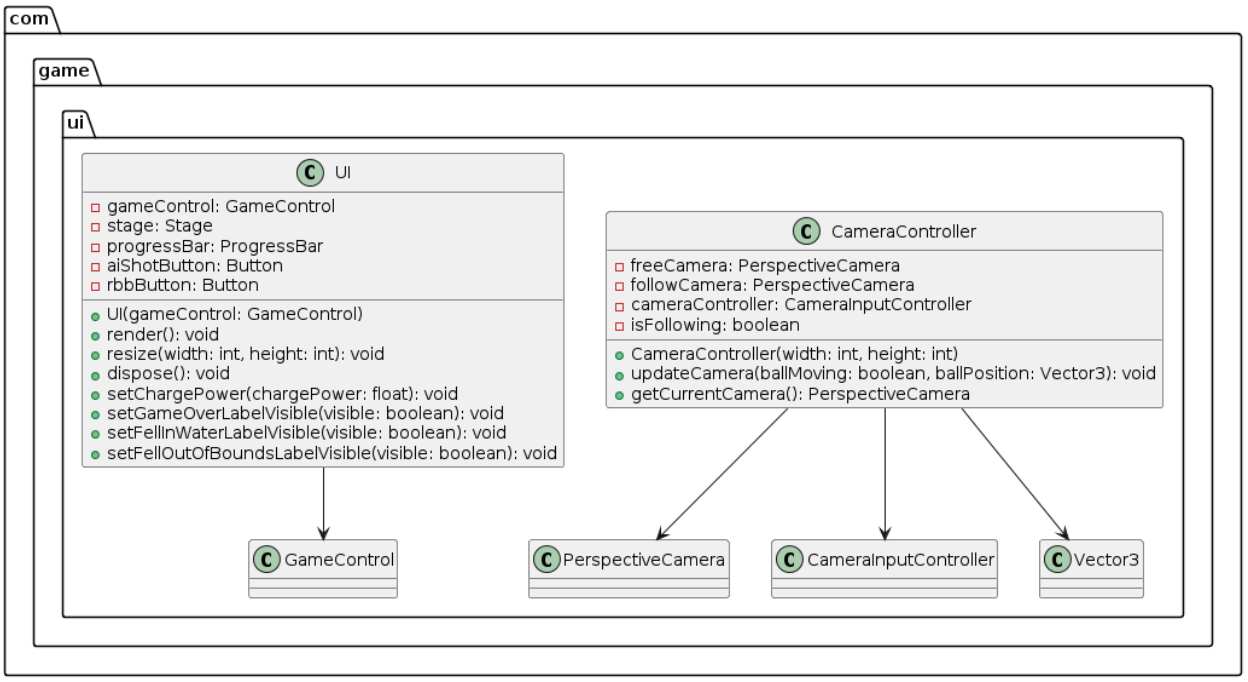
[5] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (Third edition). MIT Press.

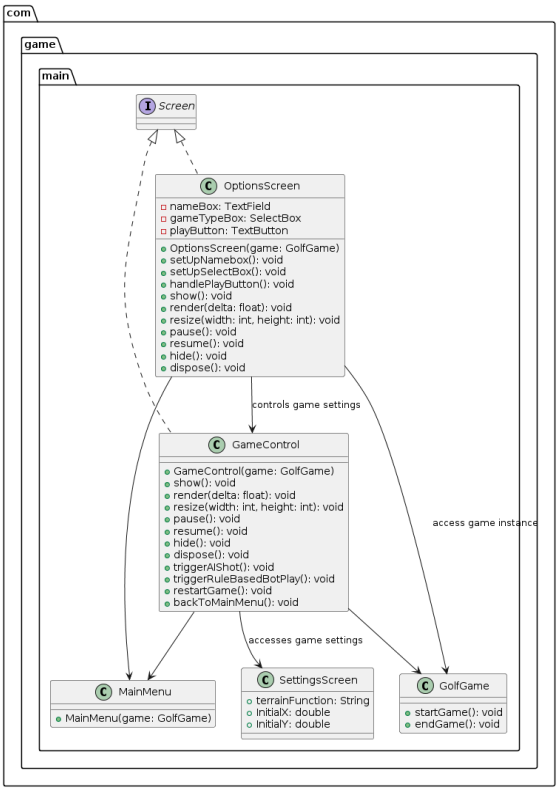
[6] Burden, R. L., & Faires, J. D. (2010). *Numerical analysis* (9th ed.). Brooks/Cole.

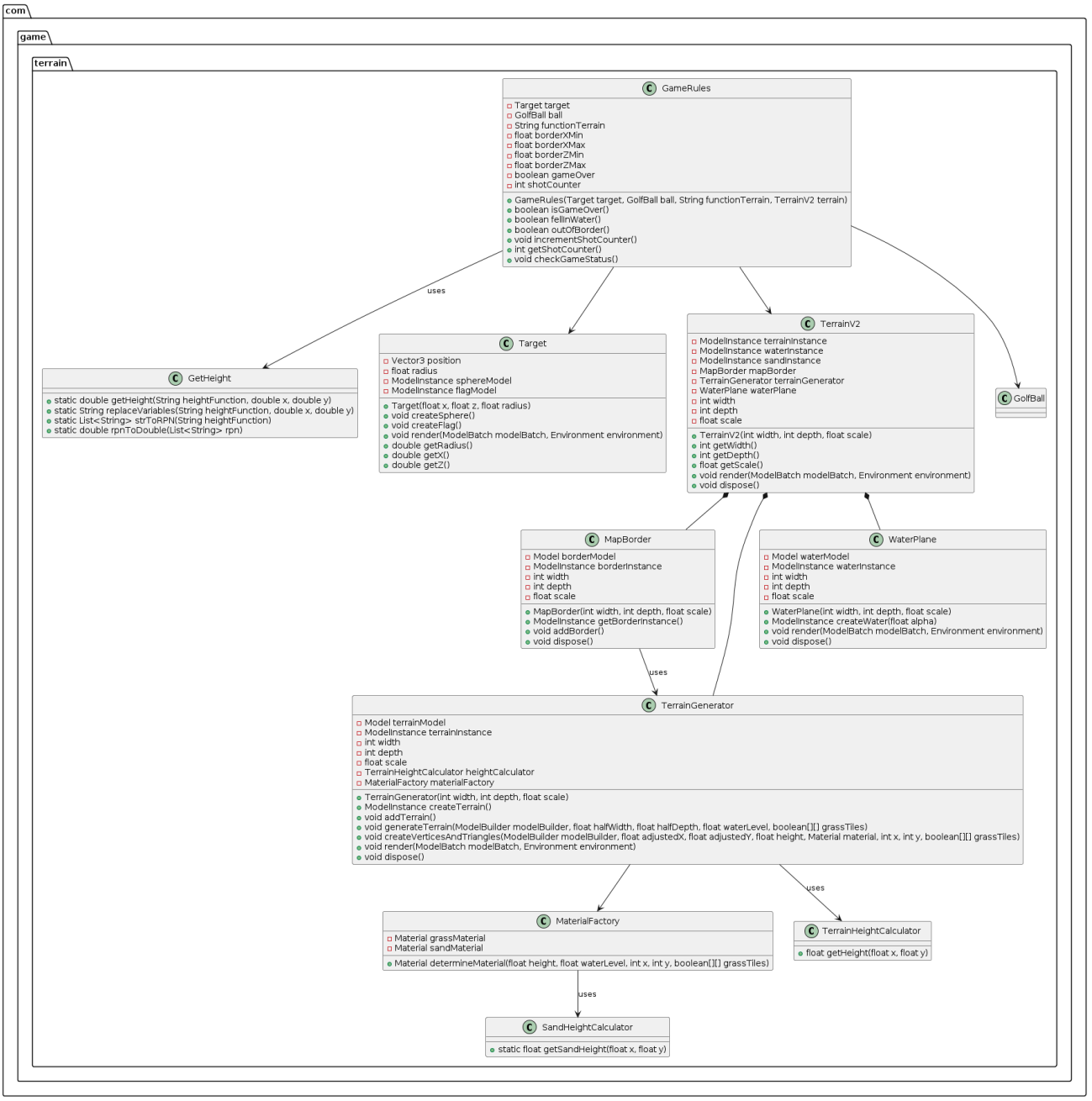
10. Appendix

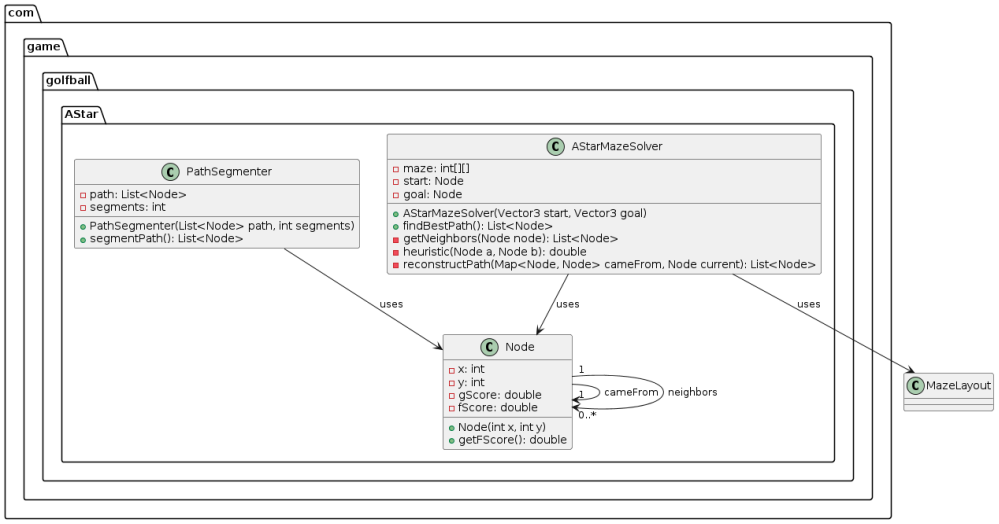
a) This appendix provides the Unified Modeling Language (UML) diagrams representing the architecture of the software system discussed in this document.

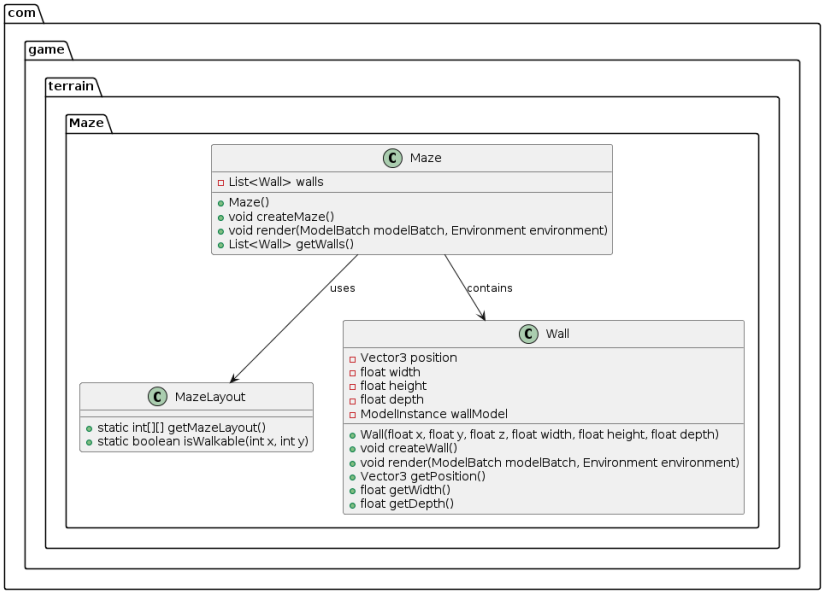

com
game
golfball
GolfAI
- targetPosition: Vector3
- physicsEngine: PhysicsEngine
- Aball: GolfBall
- gameRules: GameRules
- m: Vector3
- v: Vector3
- learningRate: double
- epsilonGrad: double
- t: int
+ GolfAI(ball: GolfBall, targetPosition: Vector3, physicsEngine: PhysicsEngine, gameRules: GameRules)
+ findBestShot(): Vector3
+ update(): void
- calculateFunction(velocity: Vector3): Vector3
- approximateGradient(velocity: Vector3, originalDeviation: Vector3): Vector3
- gradientClip(gradient: Vector3): void
- clipVelocity(velocity: Vector3): void
GolfBallMovement
- ball: GolfBall
- physicsEngine: PhysicsEngine
- gameRules: GameRules
+ GolfBallMovement(ball: GolfBall, physicsEngine: PhysicsEngine, gameRules: GameRules)
+ applyForce(force: Vector3): void
+ update(): void
RuleBasedBot
- targetPosition: Vector3
- ballPosition: Vector3
- targetRadius: float
- physicsEngine: PhysicsEngine
- RBBall: GolfBall
- gameRules: GameRules
+ RuleBasedBot(ball: GolfBall, targetPosition: Vector3, targetRadius: float, physicsEngine: PhysicsEngine, gameRules: GameRules)
+ calculateNewVelocity(): Vector3
+ update(): void
- reachedTarget(): boolean
applies force
calculates best shot
calculates new velocity
uses rules
uses physics calculations
GolfBall
- modelInstance: ModelInstance
- position: Vector3
- velocity: Vector3
- lastValidPosition: Vector3
- mass: double
+ GolfBall(startPosition: Vector3, color: Color)
+ render(modelBatch: ModelBatch, environment: Environment): void
+ getPosition(): Vector3
+ getVelocity(): Vector3
+ getMass(): double
+ setPosition(position: Vector3): void
+ setVelocity(velocity: Vector3): void
+ getModelInstance(): ModelInstance
+ updateLastValidPosition(): void
+ getLastValidPosition(): Vector3
+ isMoving(): boolean
+ dispose(): void
PhysicsEngine
+ PhysicsEngine()
+ setState(x0: double, y0: double, vx0: double, vy0: double): void
+ runSimulation(xInitialVelocity: double, yInitialVelocity: double): double[]
+ updateStateVectorRungeKutta(isImmobile: boolean): void
+ updateStateVectorEuler(isImmobile: boolean): void
+ runSingleStep(ballPosition: Vector3, ballVelocity: Vector3): double[]
+ getStateVector(): double[]
- isWithinSandArea(x: double, z: double): boolean
- calculateDerivativeX(x: double, y: double): double
- calculateDerivativeZ(x: double, z: double): double
- returnUpdatedVector(isImmobile: boolean, x: double, z: double, kineticCoefficient: double): double[]
GameRules
- target: Target
- ball: GolfBall
- functionTerrain: String
- shotCounter: int
+ GameRules(target: Target, ball: GolfBall, functionTerrain: String, terrain: TerrainV2)
+ isGameOver(): boolean
+ incrementShotCounter(): void
+ getShotCounter(): int
+ checkGameStatus(): void
retrieves terrain height
GetHeight
TerrainV2


com
game
ui
UI
- gameControl: GameControl
- stage: Stage
- progressBar: ProgressBar
- aiShotButton: Button
- rbbButton: Button
+ UI(gameControl: GameControl)
+ render(): void
+ resize(width: int, height: int): void
+ dispose(): void
+ setChargePower(chargePower: float): void
+ setGameOverLabelVisible(visible: boolean): void
+ setFellInWaterLabelVisible(visible: boolean): void
+ setFellOutOfBoundsLabelVisible(visible: boolean): void
CameraController
- freeCamera: PerspectiveCamera
- followCamera: PerspectiveCamera
- cameraController: CameraInputController
- isFollowing: boolean
+ CameraController(width: int, height: int)
+ updateCamera(ballMoving: boolean, ballPosition: Vector3): void
+ getCurrentCamera(): PerspectiveCamera
GameControl
PerspectiveCamera
CameraInputController
Vector3

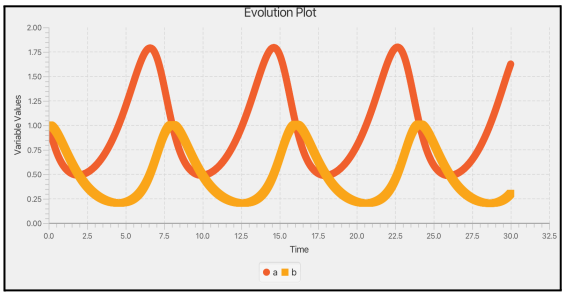

com
game
main
Screen
OptionsScreen
- nameBox: TextField
- gameTypeBox: SelectBox
- playButton: TextButton
+ OptionsScreen(game: GolfGame)
+ setUpNamebox(): void
+ setUpSelectBox(): void
+ handlePlayButton(): void
+ show(): void
+ render(delta: float): void
+ resize(width: int, height: int): void
+ pause(): void
+ resume(): void
+ hide(): void
+ dispose(): void
controls game settings
GameControl
+ GameControl(game: GolfGame)
+ show(): void
+ render(delta: float): void
+ resize(width: int, height: int): void
+ pause(): void
+ resume(): void
+ hide(): void
+ dispose(): void
+ triggerAIShot(): void
+ triggerRuleBasedBotPlay(): void
+ restartGame(): void
+ backToMainMenu(): void
access game instance
accesses game settings
MainMenu
+ MainMenu(game: GolfGame)
SettingsScreen
+ terrainFunction: String
+ initialX: double
+ initialY: double
GolfGame
+ startGame(): void
+ endGame(): void

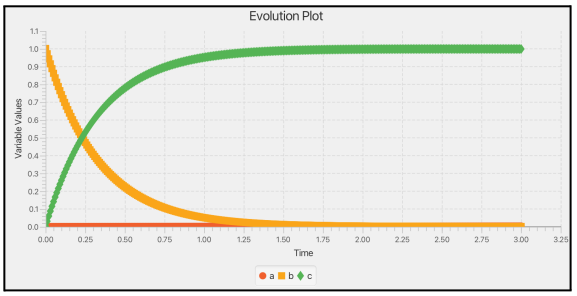

com
game
terrain
GameRules
- Target target
- GolfBall ball
- String functionTerrain
- float borderXMin
- float borderXMax
- float borderZMin
- float borderZMax
- boolean gameOver
- int shotCounter
+ GameRules(Target target, GolfBall ball, String functionTerrain, TerrainV2 terrain)
+ boolean isGameOver()
+ boolean fellInWater()
+ boolean outOfBorder()
+ void incrementShotCounter()
+ int getShotCounter()
+ void checkGameStatus()
uses
GetHeight
+ static double getHeight(String heightFunction, double x, double y)
+ static String replaceVariables(String heightFunction, double x, double y)
+ static List<String> strToRPN(String heightFunction)
+ static double rpnToDouble(List<String> rpn)
Target
- Vector3 position
- float radius
- ModelInstance sphereModel
- ModelInstance flagModel
+ Target(float x, float z, float radius)
+ void createSphere()
+ void createFlag()
+ void render(ModelBatch modelBatch, Environment environment)
+ double getRadius()
+ double getX()
+ double getZ()
TerrainV2
- ModelInstance terrainInstance
- ModelInstance waterInstance
- ModelInstance sandInstance
- MapBorder mapBorder
- TerrainGenerator terrainGenerator
- WaterPlane waterPlane
- int width
- int depth
- float scale
+ TerrainV2(int width, int depth, float scale)
+ int getWidth()
+ int getDepth()
+ float getScale()
+ void render(ModelBatch modelBatch, Environment environment)
+ void dispose()
GolfBall
MapBorder
- Model borderModel
- ModelInstance borderInstance
- int width
- int depth
- float scale
+ MapBorder(int width, int depth, float scale)
+ ModelInstance getBorderInstance()
+ void addBorder()
+ void dispose()
WaterPlane
- Model waterModel
- ModelInstance waterInstance
- int width
- int depth
- float scale
+ WaterPlane(int width, int depth, float scale)
+ ModelInstance createWater(float alpha)
+ void render(ModelBatch modelBatch, Environment environment)
+ void dispose()
uses
TerrainGenerator
- Model terrainModel
- ModelInstance terrainInstance
- int width
- int depth
- float scale
- TerrainHeightCalculator heightCalculator
- MaterialFactory materialFactory
+ TerrainGenerator(int width, int depth, float scale)
+ ModelInstance createTerrain()
+ void addTerrain()
+ void generateTerrain(ModelBuilder modelBuilder, float halfWidth, float halfDepth, float waterLevel, boolean[][] grassTiles)
+ void createVerticesAndTriangles(ModelBuilder modelBuilder, float adjustedX, float adjustedY, float height, Material material, int x, int y, boolean[][] grassTiles)
+ void render(ModelBatch modelBatch, Environment environment)
+ void dispose()
uses
MaterialFactory
- Material grassMaterial
- Material sandMaterial
+ Material determineMaterial(float height, float waterLevel, int x, int y, boolean[][] grassTiles)
TerrainHeightCalculator
+ float getHeight(float x, float y)
uses
SandHeightCalculator
+ static float getSandHeight(float x, float y)


com
game
golfball
AStar
PathSegmenter
- path: List<Node>
- segments: int
+ PathSegmenter(List<Node> path, int segments)
+ segmentPath(): List<Node>
AStarMazeSolver
- maze: int[][]
- start: Node
- goal: Node
+ AStarMazeSolver(Vector3 start, Vector3 goal)
+ findBestPath(): List<Node>
- getNeighbors(Node node): List<Node>
- heuristic(Node a, Node b): double
- reconstructPath(Map<Node, Node> cameFrom, Node current): List<Node>
uses
uses
uses
Node
- x: int
- y: int
- gScore: double
- fScore: double
+ Node(int x, int y)
+ getFScore(): double
1
1
cameFrom
neighbors
0..*
MazeLayout



b) Visualisation of Lotka-Volterra and SIR model variable evolutions through time. Graphs created with the help of our ODE Solvers.





Figures 9 and 10. Evolution plot of Lotka-Volterra and SIR model (left to right).