Crazy Putting

Report 1.2

Pie de Boer
Thomas Vroom
Laurent Bijman
Rafali Arfan
Tom Bakker
Mohammed Al-Azzani
Papuna Berdzulishvili

June 20, 2022

Abstract

This paper analyzes different numerical solvers and advanced optimization algorithms used in the context of a golf game. The physics engine of the golf game acts as a medium to examine the inner workings of the numerical solvers. Euler's method, Runge-Kutta 2 and Runge-Kutta 4 were used, of these the accuracy and precision were examined.

Different optimization algorithms were built to find the optimal way to beat the game. These include Simulated Annealing, Battle Royale Optimization, and Particle Swarm Optimization. These algorithms were able to run the game on three-dimensional, dynamic courses. The performance of the algorithms was investigated by varying the complexity of the courses, which was done by adding obstacles in the form of trees, mazes, or giving the course a more complex height profile. The algorithms were tested on both maze- and regular courses with varying complexity.

The main results showed that Runge-Kutta 4 had the best overall performance. Battle Royale Optimization proved to be the most capable optimization algorithm for all courses.

For future improvements, meta optimization of the existing algorithms would allow for improved performance. Moreover, exploring other more advanced numerical solvers could be of interest in obtaining a more accurate physics engine.

Contents

1 Introduction

This paper investigates the performance of numerical solvers and advanced optimization algorithms. This all is done in the context of a golf game.

The numerical solvers are responsible for solving ordinary differential equations (ODEs) that capture the mechanics of the physics engine. It is of relevance in many fields including chemistry, physics, mathematics and many engineering disciplines [1]. We can use these numerical solvers to simulate the movement of the ball within our course. Nowadays, Euler's method for solving ODEs is considered to be a popular and usable method for game development, but Runge-Kutta methods such as Runge-Kutta 2 (RK2) and Runge-Kutta 4 (RK4) are usually found to obtain higher accuracy [2].

Optimization algorithms are used to find the hole in the golf game using simulations where a pair of velocities is optimized. These types of algorithms serve of great importance to many engineering fields, but are mostly used in the field of artificial intelligence, for example in the training of deep learning models [3]. Classical problems utilizing optimization algorithms also include chess and timetabling [4].

We firstly implemented Euler as our sole ODE-solver, but soon after implemented more advanced numerical solvers such as RK2 and RK4 to achieve higher accuracy. We also explored relatively complex optimization algorithms, as simpler algorithms can struggle with finding a hole-in-one in more complex terrains because of the risk of getting stuck at a certain local minima [5]. The optimization algorithms implemented were Particle-Swarm Optimization (PSO), Battle-Royale Optimization (BRO) and Simulated Annealing (SA). We first implemented SA as it was the simplest algorithm of all three and its features include stochasticity, which can help avoid local minima. Secondly, PSO was implemented because of its reliability and popularity to optimize many problems [5]. Lastly, BRO was implemented because of its uniqueness compared to other algorithms and because it has shown more success over PSO according to recent research [6].

The research questions that we want to answer are:

- Which numerical solver is the most accurate for a fixed step-size?
- How does the complexity of a course influence the 'hole-in-one' performance of the optimization algorithms?
  - Complexity meaning adding more obstacles to the course
  - Complexity meaning giving the course a more complex height profile
- Which optimization algorithm performs best on mazes of varying complexity?

The golf game we built is focused on the putting of a golf ball [7]. In this process, the ball is only allowed to roll on the surface. Given some initial position, the ball has to be shot into the hole. If the ball gets shot into the water, the player or bot will be penalized (1 extra shot). The courses are created using a spline function to define the height profile and can contain obstacles like trees or mazes.

In this paper, an in-depth explanation of the theory behind the methods implemented is explained in the Methods section, this includes the intricacies behind the numerical solvers and the optimization algorithms implemented. Then, the way these methods were used within

the context of golf is explained in the Implementation section. In the Experiments section, we describe the experiments done to answer the research questions laid out previously and the results of those experiments can be found in the Results section. The Discussion section will contain the contextualization of the results gained in the previous section as well as their relation to our research questions. Finally, this will all be wrapped up in the Conclusion section which will concisely reiterate some of our previous sections and suggest possible improvements for the future.

2 Methods

2.1 Physics

2.1.1 Mechanics

The ball is brought into motion with an initial velocity $\vec{v}$, which can be decomposed into two separate velocity vectors $\vec{v_x}$ and $\vec{v_y}$. These are the only parameters influenced by the player or bots. On the terrain, the motion of the ball is governed by three forces [8]:

- Force of gravity G, constant downwards directed force
- Normal force N, exerted by surface of earth (normal to surface)
- Friction Force:
  - Kinetic friction μ_k, the force in the opposite direction of the movement of the ball, only exerted when the ball is moving
  - Static friction μ_s, when the ball is in rest

The following free body diagram visualizes the most important forces.

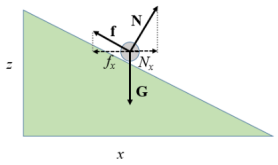


Figure 1: Free body diagram of golf ball on slope [8].

The terrain is described by a height function. It's a real function that takes in (x, y) coordinates to output a (z) coordinate. The acceleration of the ball in the x direction and y direction is based on a height function. This means the terrain can be captured in the following formulae [8]:

$$\ddot{x} = -g\frac{\partial h}{\partial x} - \mu_K g\frac{v_x}{\sqrt{v_x^2 + v_y^2}} \tag{1}$$

$$\ddot{y} = -g\frac{\partial h}{\partial y} - \mu_K g\frac{v_y}{\sqrt{v_x^2 + v_y^2}} \tag{2}$$

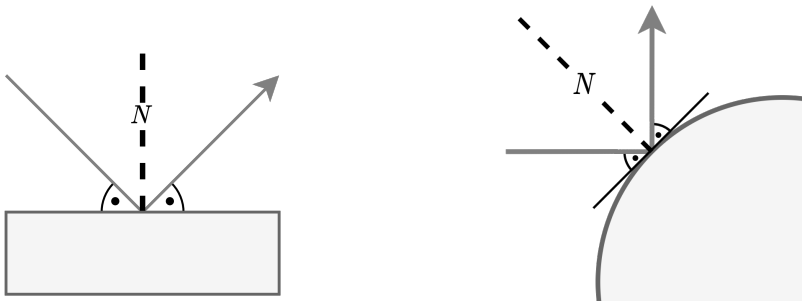
Both equations contain two terms. In both cases, the first term captures the downhill force (due to gravity and normal force). The kinetic friction is captured in the second term, this is in the opposite direction of the motion of the ball. Besides kinetic friction, it is of great

importance to take the static friction into account. Sometimes the downhill force exerted on the ball might be higher than the static friction and the ball will get into motion again.

$$\mu_S > \sqrt{\left(\frac{\partial h}{\partial x}\right)^2 + \left(\frac{\partial h}{\partial y}\right)^2} \tag{3}$$

2.1.2 Collision

We also need to describe how objects interact with each other. When the ball comes across an obstacle like a tree or a wall, its velocity gets reflected in the vector perpendicular to the surface of the obstacle (the normal vector N) [9]. For simplicity, we assume all walls and trees are perfect rectangles and cylinders respectively.



(a) Flat surface (b) Round surface

Figure 2: Collision for a ball on flat/round surfaces.

Assuming no energy is lost during the collision, this interaction can be described using the following formula:

$$\vec{V} = \vec{v} - 2(\vec{v} \cdot \vec{N}) * \vec{N} \tag{4}$$

Where $\vec{V}$ is the velocity of the ball after hitting the object, $\vec{v}$ is the velocity of the ball before hitting the object and $\vec{N}$ is the normalized normal vector of the obstacle.

2.2 Numerical Solvers

The ordinary differential equations (ODEs) needed to describe the motion of the golf ball are most of the time too complicated to be solved analytically. Even if an analytical solution is available it is often so complicated that it is of little use, the numerical solution is sometimes the only way to obtain information about the system. We explored and implemented three different numerical solvers to solve the physics equations earlier mentioned.

see appendix A for further description

2.2.1 Euler

The use of elementary difference methods to obtain initial value problems approximate solution of differential equations was first reported in 1768 by Leonhard Euler [10] which gives a local error of $O(h^2)$ and a global error of $O(h)$ [11]:

$$y_{n+1} = y_n + h \cdot f(t_n, w_n) \tag{5}$$

Where the derivative is only approximated once and the next value is obtained by multiplying a given step size h with the derivative and adding to it, the initial value is given.

2.2.2 Runge-Kutta order 2

In order to achieve higher accuracy, more advanced numerical solvers were implemented. Runge-Kutta schema order 2 (RK2) was introduced which gives a local error of $O(h^3)$ and a global error of $O(h^4)$ [11]:

$$y_{n+1} = y_n + \frac{1}{2}h \cdot (k_1 + k_2) \tag{6}$$

$$k_1 = f(t_n, y_n) \tag{7}$$

$$k_2 = f(t_n + h, y_n + h \cdot k_1)[12] \tag{8}$$

2.2.3 Runge-Kutta order 4

To obtain even more accurate results, Runge-Kutta scheme of order 4 (RK4) was used which retains a local error of $O(h^5)$ and a global error of $O(h^4)$ [11]:

$$y_{n+1} = y_n + \frac{1}{6}h \cdot (k_1 + 2 \cdot k_2 + 2 \cdot k_3 + k_4) \tag{9}$$

$$k_1 = f(t_n, y_n) \tag{10}$$

$$k_2 = f(t_n + \frac{h}{2}, y_n + h \cdot \frac{k_1}{2}) \tag{11}$$

$$k_3 = f(t_n + \frac{h}{2}, y_n + h \cdot \frac{k_2}{2}) \tag{12}$$

$$k_4 = f(t_n + h, y_n + h \cdot k_3)[13] \tag{13}$$

It is similar to Runge-Kutta 2 where the derivative is calculated at multiple points and a weighted average is taken to approximate the next point, the difference here however is that instead of taking the derivative at two different points, it takes it at four different points.

2.3 Artificial Intelligence

During our research, we used different kinds of ai-optimization algorithms. In optimization, we want to find the best possible output (fitness) within a function and a given search space [14]. Two vital elements make up the success of most optimization algorithms: exploration and exploitation. Exploration allows the discovery of new areas in the search space which have not been explored yet while exploitation refers to the search for better solutions within the vicinity of the areas which have already been discovered. Different optimization algorithms use these elements in their own way. Successful optimization algorithms strive to find a balance between the two to ensure convergence towards a global best solution.

2.3.1 Simulated Annealing

Simulated annealing (SA) takes inspiration from thermodynamics. The annealing process is used to explore a wide search space [15]. To simulate the annealing process the algorithm makes use of a temperature-function. The temperature starts high, allowing the process to freely move about the search space, with the hope that in this phase the process will find a good region with the best local optima (exploration). The temperature is slowly brought down,

reducing the stochasticity and forcing the search to converge to a global optima (exploitation).

Simulated annealing works by iteratively picking a random neighbour relative to the current state and comparing it to the current state s. Updating the current state s is done by comparing a probability P with a random value $R \in [0, 1]$ [16]. The probability P is based on the fitness of the current random neighbour s_{new}, the fitness of the current state s and the temperature T. If the probability P is greater or equal to the random value R then the neighbour s_{new} is accepted as the current state s. After this, a new iteration starts. Once the maximum number of iterations has been reached or the current state s is valid the loop will stop and the current state s will be returned.

Due to the probability function, stochasticity is introduced in the algorithm [15]. This makes SA appropriate for objective functions where other local search algorithms do not perform well.

see appendix F for the detailed flowchart of SA

2.3.2 Particle Swarm Optimization

Particle Swarm Optimization(PSO) is an algorithm based on swarm behaviour seen in nature [17]. PSO is an evolutionary algorithm which works by searching the space of an objective function by adjusting the trajectory of individual agents, named particles. In PSO a number of particles are distributed over the search space. This swarm, being the collection of particles, is initialized randomly. After a particle has been placed in the search space, the particle evaluates the objective function at its current location.

In each iteration, the current position of the particle is compared to its own previous best-known position and the global best-known position in the search space. Based on these two components the particle is updated. Those two positions, in combination with a random factor, are used in calculating the updated velocity. A new point is chosen by updating the current position with that velocity. The particle is updated using the following formulae:

$$P_i^{t+1} = P_i^t + V_i^{t+1} \tag{14}$$

Where P_i^{t+1} is the new particle, P_i^t the old particle and the V_i^{t+1} is the velocity used for updating the current position. This velocity is calculated using the following formulae:

$$V_i^{t+1} = wV_i^t + c_1 r_1 (P_{best(i)} - P_i^t) + c_2 r_2 (P_{bestglobal}^t - P_i^t) wV_i^t \tag{15}$$

This part is called inertia, which is used to keep the current motion of the particle. The inertia weight w is used to scale this motion and has an impact on the global and local search ability. The larger this weight, the larger the step-size of the particle and therefore a higher global optimization ability [18]. The lower this weight is, the higher the local optimization ability is [18]. The inertia weight coefficient combined with the current velocity V_i^t is the inertia.

$$c_1 r_1 (P_{best(i)} - P_i^t) \tag{16}$$

This part is called the cognitive or social component. This component quantifies the performance of the particle P_i^t relative to the particles own previous best performance $P_{best(i)}$.

$$c_2 r_2 (P_{bestglobal}^t - P_i^t) \tag{17}$$

This is the social component that quantifies the performance of the particle P_i^t relative to global best performance $P_{bestglobal}$.

To introduce stochasticity in the algorithm, random coefficients $r_1, r_2 \in [0, 1]$ are added to the cognitive and social components [19]. Social coefficients and global coefficients (c_1, c_2) are used to express how confident a particle is in itself or its neighbours. The higher c_1 the more a particle is attracted to its own optima. The higher c_2 the more a particle is attracted to the global best particle. In order to achieve an optimal working PSO there should be a balance between these two coefficients. This balance however depends on the problem at hand. After all the particles have been moved the next iteration starts.

see appendix B for the detailed flowchart of PSO

2.3.3 Battle Royale Optimization

The Battle Royale Algorithm is inspired by a relatively new popular genre of video games [6]. Similar to PSO it has a randomly initialized population of agents (possible solutions) searching the space of an objective function. Iteratively the agents update their positions to find the optima of the respective function. Unlike PSO, the agents of BRO (referred to as soldiers) keep track of how many times they have been damaged with a damage counter and there is a dynamic upper and lower bound constant which shrinks the search space.

Iteratively, each soldier will be compared to the closest soldier based on their position in the search space using the Euclidean distance measure. The soldier deemed to be less fit according to their outputs to the objective function will be damaged and its damage counter will be incremented by one, as a result, its position in the search space will be updated. The damaged soldier will be referred to as *dam* and the non-damaged one will be referred to as *vic*. Notably, the mechanism in which *dam* position in the search space is updated will differ based on whether *dam* damage counter has surpassed (or is equal to) a certain customizable threshold [6]. In the case that the soldier's damage counter has not surpassed the threshold, its position will be updated in such a way exploitation is focused. This is done by moving the soldier's position towards a point somewhere between its current position and the position of the best soldier based on the objective function[6].

This is mathematically expressed as:

$$x_{dam.d} = x_{dam.d} + r(x_{best.d} - x_{dam.d}) \tag{18}$$

where $x_{dam.d}$ refers to position of the damaged soldier in dimension d, $x_{best.d}$ refers to the position of the best soldier in dimension d and r refers to a randomly generated number within the interval [0, 1].

In the case that the soldiers damage counter has surpassed the threshold, exploration will be focused, so the agent will randomly reposition itself within an upper and a lower bound specific for each dimension of the search space[6]. This is shown below:

$$x_{dam.d} = r(ub_d - lb_d) + lb_d \tag{19}$$

In this case, ub_d and lb_d represent the upper and lower bounds of dimension d in the search space. As mentioned previously, the algorithm implements a mix of exploitation and exploration by occasionally shrinking the lower and upper bounds based on the current position of

the best soldier [6]. This is done by the following:

$$ub_d = x_{best_d} + SD(\overline{x_d}) \tag{20}$$

$$lb_d = x_{best_d} - SD(\overline{x_d}) \tag{21}$$

Here, $SD(\overline{x_d})$ represents the standard deviation of the population relative to dimension d.

see appendix C for the detailed flowchart of BRO

2.4 Splines

We use a spline function to define the height profile of the course. It maps a set of input coordinates to an output value ($[x, y] \to \mathbb{R}$) and has a set of predetermined points with known (x, y, z)-coordinates, called knots. For any set of input coordinates $[x, y]$ within the bounds of the spline, a spline function interpolates the value of these coordinates using its knots. Depending on the interpolation method and the number of knots, a spline function can infinitely approach a height function $f(x, y)$ [20].

2.4.1 Bicubic Interpolation

Bicubic interpolation is an interpolation method used to interpolate data values on a two-dimensional rectangular grid. It is considered to be one of the best interpolation methods for creating smooth surfaces because it uses derivatives to estimate the slopes between the data points [21]. Suppose the function values f are known at the four corners $(0, 0), (1, 0), (0, 1), (1, 1)$ of the unit square and it is possible to calculate the derivative with respect to the x-axis f_x, the derivative with respect to the y-axis f_y and the mixed derivative f_{xy} using standard numerical methods. Then any pair of values $[x, y]$ within this unit square can be calculated using the following formulae:

$$p(x, y) = \begin{bmatrix} 1 & x & x^2 & x^3 \end{bmatrix} \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} 1 \\ y \\ y^2 \\ y^3 \end{bmatrix} \tag{22}$$

where

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} =$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -3 & 3 & -2 & -1 \\ 2 & -2 & 1 & 1 \end{bmatrix} \begin{bmatrix} f(0,0) & f(0,1) & f_y(0,0) & f_y(0,1) \\ f(1,0) & f(1,1) & f_y(1,0) & f_y(1,1) \\ f_x(0,0) & f_x(0,1) & f_{xy}(0,0) & f_{xy}(0,1) \\ f_x(1,0) & f_x(1,1) & f_{xy}(1,0) & f_{xy}(1,1) \end{bmatrix} \begin{bmatrix} 1 & 0 & -3 & 2 \\ 0 & 0 & 3 & -2 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & -1 & 1 \end{bmatrix} \tag{23}$$

Note: these coefficients can be precomputed for increased performance.

3 Implementation

3.1 Graphics Engine

Since we did not build a graphics engine from scratch, we utilized the libGDX game development framework and some of its internal libraries.

3.2 Level Generation

We have a few different tools for generating levels of varying complexity. First, we define the height profile of the level using a spline function, where any vertical coordinate below zero is considered a body of water. To limit the computational requirements we restrict the course to a 20 x 20 field, with the origin $(0, 0)$ in the centre. If the ball travels outside of these bounds it will also be considered as hitting a body of water. To allow for more added complexity we included the option to add tree obstacles to the course at random or pre-defined locations, or we can turn the course into a maze where multiple shots are guaranteed needed to reach the hole.

3.2.1 Terrain

The height profile of the terrain is defined by a spline function with 256 knots, using bicubic interpolation. Suppose we have a height function $f(x, y) \to \mathbb{R}$ and a 16 x 16 matrix of floating point values $b[i][j]$ with $i, j \in \{0, 1, ..., 15\}$. Then the value of knot k is defined as $k_z = f(k_x, k_y) + b[k_i][k_j]$. Whenever we want to know the height of a pair of coordinates (x, y) we first find the four knots that surround the coordinates, called the quadrant. Because the knots are in an evenly spaced grid, there will exist a quadrant for every pair of coordinates that are within the bounds of the course. We can then calculate the height of the pair of coordinates using the bicubic interpolation formula [22].

3.2.2 Trees

For simplicity, we assume trees are perfect cylinders and hitting them only causes the ball to lose 20% of its velocity. Every tree t is defined by a position (t_x, t_y) and a radius r. Trees can be randomly generated with the following constraints:

- for height profile h, $h(t_x, t_y) \geq 0.1$
- the distance from the starting position to $(t_x, t_y) \geq 1$
- the distance from the hole to $(t_x, t_y) \geq 1$
- $0.2 \leq r \leq 0.4$

3.2.3 Mazes

We use a search algorithm for creating mazes of varying complexity. First, we divide the course into an evenly spaced grid of cells, the length of this grid of cells is what is referred to as the complexity of a maze. The algorithm we use for creating the mazes is called a random depth-first search algorithm. Suppose that all cells start off with walls on all sides. Then we can create a maze from this grid using the following pseudo-code [22]:

Algorithm 1 Randomized Depth-First Search

```
procedure CreateMaze(cell)
    mark cell as visited
    for every unvisited neighbour n of cell do          ▷ in random order
        remove wall between cell and n
        CreateMaze(n)
```

This results in a grid of cells with walls in such a way that the maze is guaranteed solvable from any position. The maze is constructed with two different types of walls. Most of the walls

are regular walls with normal collision, but at the corners of the walls there are special 'death' walls that reset the ball to its previous position. These were added to make it impossible for any collision bugs to disturb the data. Only adding these to the edges where collision bugs will be more likely to occur also allows the AI to still bounce shots off the regular walls.

3.3 Physics

To represent the most important parameters of the ball at any moment in time we introduced a state vector. The state vector contains the x, y position and x, y velocity:

$$\mathbf{x}(t) = \begin{pmatrix} x(t) \\ y(t) \\ v_x(t) \\ v_y(t) \end{pmatrix} \tag{24}$$

Using our numerical solvers we constantly update the values based on the physics equations mentioned in the methods' section 1. The physics engine is based on the following restrictions provided in the appendix G.

3.4 Artificial Intelligence

3.4.1 Distance Measures

In non-maze courses, we used the Euclidean distance to evaluate the fitness of each shot [23].

$$d(p, q) = \sum_{i=1}^{n} \sqrt{(q_i - p_i)^2} \tag{25}$$

To make informed shots in maze courses we used an algorithm called flood fill, it is a common technique used in game programming [24]. We first represented the course as a 2-dimensional grid, then implemented Breadth-First Search (BFS) in such a way it creates a clear navigation path towards the hole. While at the same time avoiding obstacles such as bodies of water, trees, and walls. This was done by assigning each cell in the grid the amount of steps it would need to take for the flood fill BFS algorithm to reach it (starting from the hole) while avoiding obstacles. Then, we can assign each pair of coordinates a measured heuristic score relative to how close they are to the hole when taking obstacles into account.

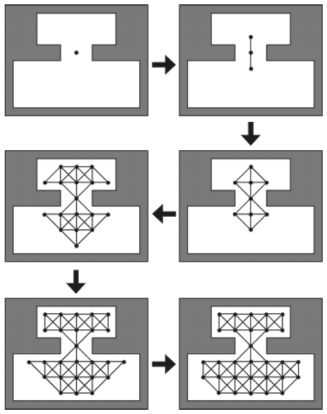


Figure 3: Flood Fill visualization taken from "Game AI by Example" [24].

3.4.2 Optimization Algorithms

Translated to the medium of golf we strive to search for the best pair of velocities (the search space) to either try to find a hole-in-one or to iteratively navigate through maze courses. For maze courses, we evaluate the fitness of a velocity pair by shooting the ball with it and getting the flood fill distance measure relative to the final position of the ball. For normal courses, we also evaluate the fitness of a velocity pair by applying it to the ball, however we instead use the shortest/smallest Euclidean distance between the ball and the hole in each time step to ensure the ball passes through the hole. Additionally, we added a "hill-climbing" aspect to BRO and PSO as we initiate a local search relative to the current best agent respectively at the beginning of each iteration of the algorithm.

For the parameters of PSO we made use of the following weights:

$$w = 0.4\left(\frac{i-(i_{max}+1)}{(i_{max}+1)^2}\right)+0.4 \tag{26}$$

$$c_1 = 3.5 \tag{27}$$

$$c_2 = 0.5 \tag{28}$$

$$r_1, r_2 = \in [0,1] \tag{29}$$

Where i refers to the current iteration number and i_{max} refers to the maximum possible allowed number of iterations.

For the SA parameters we made use of the following probability and temperature function:

$$P = e^{-1\left(\frac{s_{fitness}-s_{newfitness}}{T(i)}\right)} \tag{30}$$

$$T = 1-\frac{i+1}{i_{max}} \tag{31}$$

Where i refers to the current iteration number, i_{max} refers to the maximum possible allowed number of iterations and s refers to the current state.

3.5 Numerical Solvers

We implemented the different numerical solvers to solve the aforementioned equations [2.2], update the state vector of the ball, derive the velocities and accelerations at those states after which the next set of positions and velocities are calculated.

For Euler, the derivative is only taken once at its current state. We do not change the state vector, but use the current velocities (first derivative) and use the current state vector of the ball to approximate the accelerations (second derivative) needed to calculate the next state of the ball.

Using RK2, the derivative is taken twice at two different points, once at its current state and once at an altered state which we gained by temporarily altering the state vector of the ball according to another equation (8). Then, we can use the two different states to obtain the first and second derivatives and use them to calculate the next state of the ball.

Finally, we implemented RK4 in a very similar way to RK2. We temporarily alter the state vector three times according to these equations(11) (12) (13). Afterwards, we use the current state vector in addition to the three previously mentioned state vectors to calculate the first and second derivatives needed to use the last equation(10) to update the state of the ball.

4 Experiments

4.1 Accuracy of the Numerical Solvers

We wanted to measure the accuracy of our solvers with respect to different step sizes to ensure their correctness. To achieve this, we simulated the movement of the ball using the different numerical solvers with the same parameters. To avoid issues with our stopping condition, we decided to record the position of the ball at a certain time step. Subsequently, this also meant we had to use certain step sizes that were divisible by the time-step in which we record the position. For example, as we chose to record the state vector's position at $t = 1$, it would be pointless to use a step size of $h = 0.3$ as none of the implemented numerical solvers would be able to solve the position of the ball at $t = 1$. The height profile we chose to test the numerical solvers used the height function $f(x, y) = 0.2x + 5$. We chose this function because it is simple to analytically calculate a solution at $t = 1$. The full specifications can be found in appendix D.

4.2 Artificial Intelligence

4.2.1 Hole-in-one performance

The performance of the AI is measured through their running time, the number of simulations needed to find the hole and whether they can make a hole-in-one shot. To find out which AI is the fastest, the AI's compilation time was calculated from the start of the search until the solution was found. It is important to take the number of simulations into account, as less simulations taken by an AI means that this AI is more efficient. The most crucial indicator of the AI's performance is the number of missed (no hole-in-one) shots. This shows the ability of an AI to make one perfect shot to the target.

We defined multiple options to control the complexity of the golf course, for example by adding obstacles or modifying the smoothness of the height profile. Special experiments have been conducted to see how the AI's performance will be affected by different course complexities. The complexity of a golf course is determined by the number of obstacles it contains and how smooth the height profile is.

In the first experiment, a different number of obstacles in the form of trees were added to the course. The number of trees was incremented by ten until it reached 50. These trees were randomly generated each iteration. After generating the course, all the AIs were tested 40 times each on the same course with identical conditions to provide reliable results. This experiment was repeated for 60 different terrains per iteration. Data like the AI's runtime, the number of simulations and the number of missed shots was collected. The full specifications are available in the appendix E.

The second experiment was conducted to investigate how different height profiles can affect the performance of the AIs. To do this, we made use of the bicubic spline that describes the height profile. We only change a single parameter called $maxheight$. Based on this parameter, we define the height of each knot of the spline function as $k_z = 0.1 + r$, where r is a random number in the interval $[0, maxheight]$. Thus, a higher max height will create a more complex height profile. For each different max height we generate 60 different courses, where all AIs are tested 40 times on the same configuration. The full specifications are available in the appendix E.

4.2.2 Maze courses

To see how the AIs will perform on maze-like courses, we have conducted experiments for maze complexities from three to eight. All the AIs were tested 40 times in similar conditions for each complexity. The number of shots taken by the AIs to reach the target was recorded. Additionally, the number of simulations was collected since it will show how much computational resources each AI takes to find the hole. Similar to any classic maze, the ball and the target were placed on opposite corners of the maze. The full specifications are available in the appendix E.

5 Results

5.1 Numerical Solvers

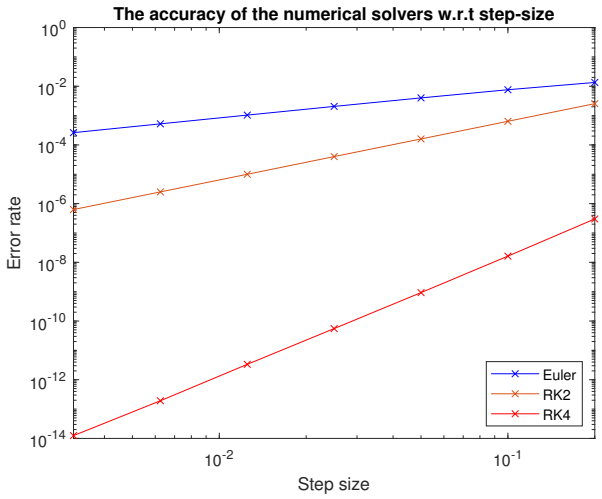


Figure 4: The accuracy of the numerical solvers.

5.2 Artificial Intelligence

5.2.1 Hole-in-one performance

The following data includes the average run-time, the average number of simulations and the number of missed (no hole-in-one) shots (per 100 shots). We calculated the sample standard deviation using the formula [25]:

$$\mathsf{S} = \sqrt{\frac{\sum(x_i - \bar{x})^2}{n-1}} \tag{32}$$

This will show how spread out our data samples are.

For the first experiment related to the number of obstacles:

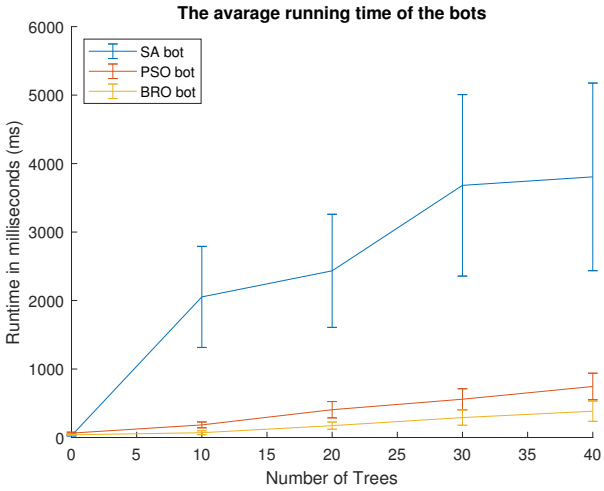


Figure 5: Average runtime of the AIs

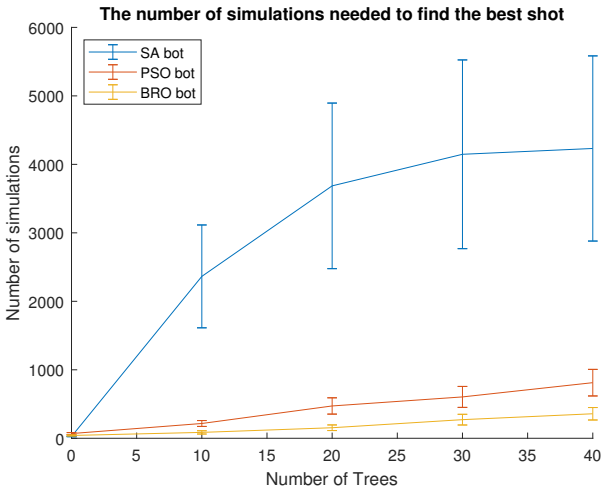


Figure 6: Average number of simulations

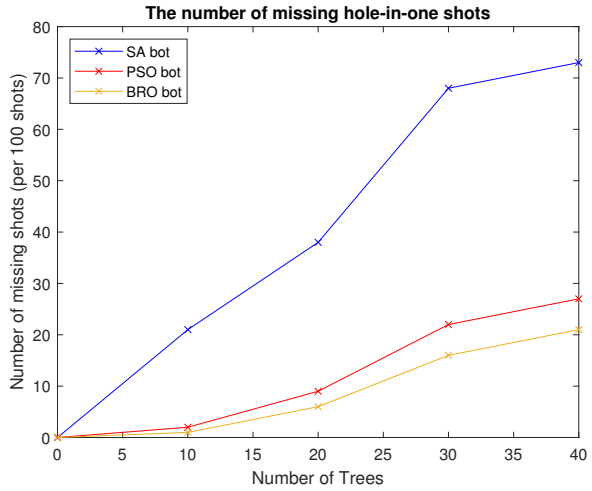


Figure 7: Number of missed hole-in-one shots (per 100)

For the second experiment related to the complexity of the height profile:

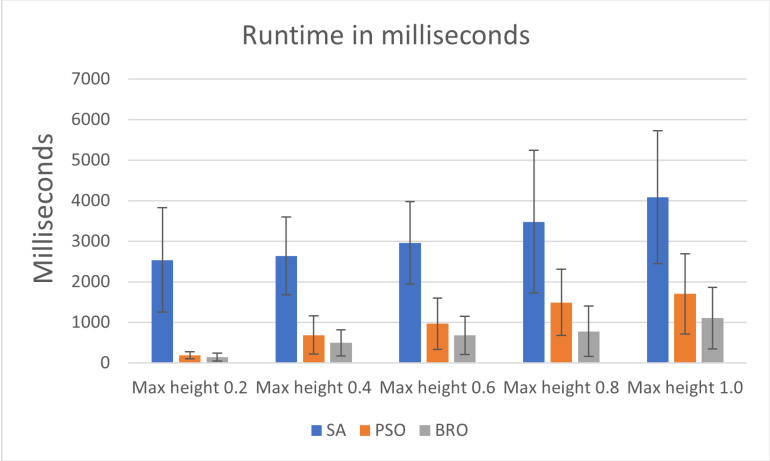


Figure 8: Average runtime of the AIs

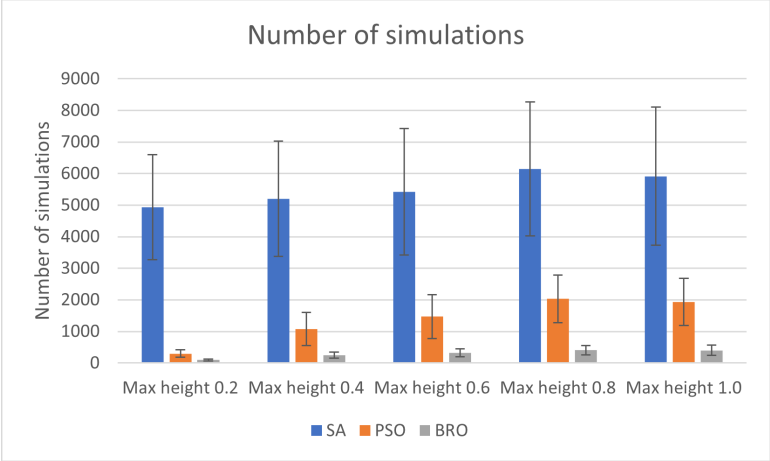


Figure 9: Average number of simulations

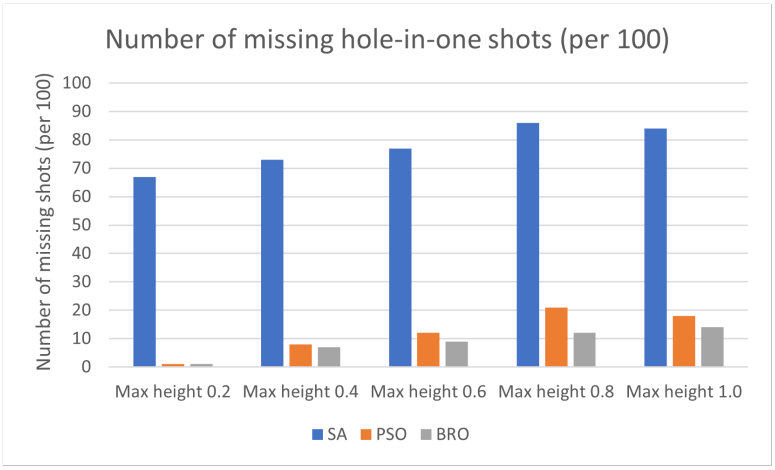


Figure 10: Number of missed hole-in-one shots (per 100)

5.2.2 Maze courses

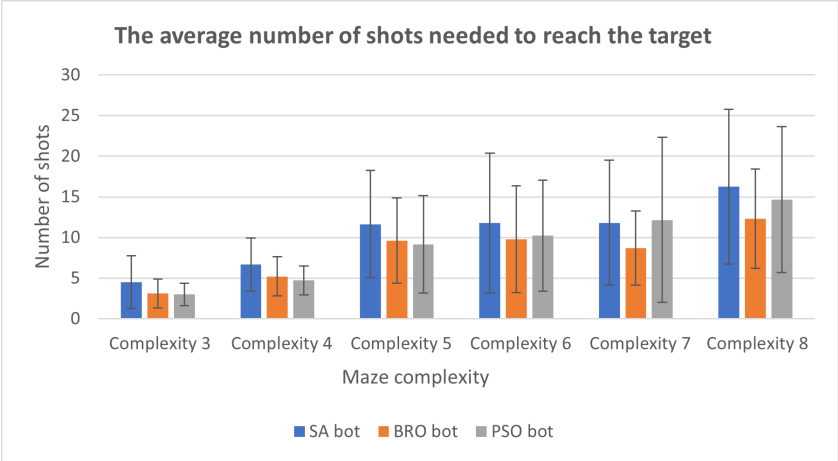


Figure 11: Average number of shots needed to reach the hole

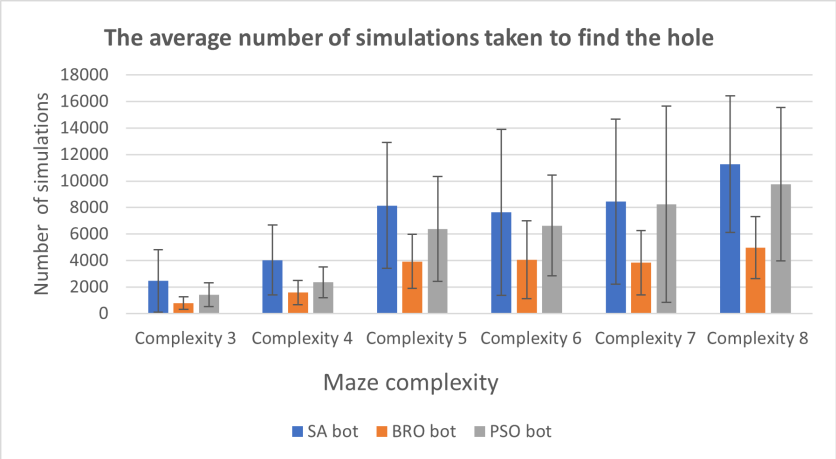


Figure 12: Average number of simulations

6 Discussion

The results demonstrate that a higher-order numerical solver proves to be more accurate. Previous research showed RK4 is a better method than Euler [2] and our research is in agreement with this. Additionally, the figure 4 showed expected behaviour. Decreasing the step-size by a factor of two results in improved accuracy of a factor of two for Euler, four (which is 2^2) for RK2 and 16 (which is 2^4) for RK4. This is in agreement with scientific literature because Euler, RK2 and RK4 respectively have global errors of $O(h)$, $O(h^2)$ and $O(h^4)$ [11].

Regarding the optimization algorithms implemented, BRO showed itself to be the most capable one out of the three implemented algorithms, while it was clear that SA performed the worst. For normal courses, even when taking standard deviation into account, BRO performed best in terms of average runtime in ms, average number of simulations and average number of missed hole-in-one shots when taking all terrain complexities into account (as seen in figures 5-10) with PSO being a close second and SA clearly being left behind. Something which can be of interest is that BRO and PSO have a similar rate of change in terms of performance with respect to an increase in terrain complexity while SA showed to possess a much higher rate of change relative to the other two when looking at the previous figures. Possibly because of the similarity of BRO and PSO algorithms when compared to SA.

For maze-like courses however, the results obtained showed a much higher standard deviation than expected. In figures 11 and 12, a general trend can be seen where BRO mostly performs the best, however there exist instances where BRO performed slightly worse than PSO such as in figure 11 when testing for mazes with complexities 3, 4 and 5. The high standard-deviation is most likely a result of the maze-generation algorithm being random, so mazes generated with the same complexities can have exceptionally varying difficulties. In retrospect, to get rid of the large standard deviations it was vital that we should have done much more runs per maze complexity. A common behaviour preserved in figures 11 and 12 was that again, SA performed the worst out of the three implemented algorithms. Further internal investigations implied that this is most likely a cause of its inability to escape local minima and its likelihood to prematurely converge. Perhaps, it would have been beneficial to further optimize and explore other variants of SA to make it more capable than it is now.

7 Conclusion

A physics engine was built from scratch in which RK4 was used as the primary numerical solver. The accuracy of the different numerical solvers was investigated. The optimization algorithms used the physics engine in the context of a golf game. We conducted experiments on golf courses of varying complexity to show which algorithm proved to be the most usable.

Based on our research RK4 clearly outperformed the other numerical solvers, while Euler performed the worst as expected. BRO demonstrated to be the best optimization algorithm throughout all the experiments, SA proved to be the worst. Something to keep in mind however is that the some of the results may be inadequate, seeing the high standard deviation.

Future research could be based on meta-optimization of our existing optimization algorithms, as better tweaked parameters could show enhanced performance. In addition, the further implementation of other optimization algorithms also come to mind when talking about possible future improvements. This can include a more robust investigation relating to the three different categories of optimization algorithms: evolution-based, physical-based

and swarm-based algorithms [26] and how algorithms based on those different categories differ in performance. As an alternative to flood fill, other path-finding algorithms such as the A* algorithm or Dijkstra is of interest to ensure the ball takes the shortest path possible. Finally, more advanced ODE solvers could be implemented to obtain higher accuracy when simulating the movement of the ball. This could include linear multistep methods such as the Adams-Moulton 4 method [11] and adaptive step-size methods such as the Bogacki-Shampine method [11].

Appendix

APPENDIX A: Ordinary Differential Equations

The term *æquatio differentialis* or differential equation was first used by Leibniz in 1676 to denote the relation between the differentials dx and dy of two variables x and y. Such relationship, involves the variables x and y together with other symbols a, b, c ... that represents constants [27]. The first order equation specifies the state of x and how the system itself changes in time as the function of x itself $\frac{dx(t)}{dt} = f(x(t))$ where $\frac{dx(t)}{dt} = \dot{x}$ Second order equation is most commonly used in $\ddot{x}$ which means they involve second derivatives, is most commonly used in physics $\ddot{x} = \frac{dx^2(t)}{dt^2}$. For example, if we have the differential equation

$$\ddot{x} = a,$$

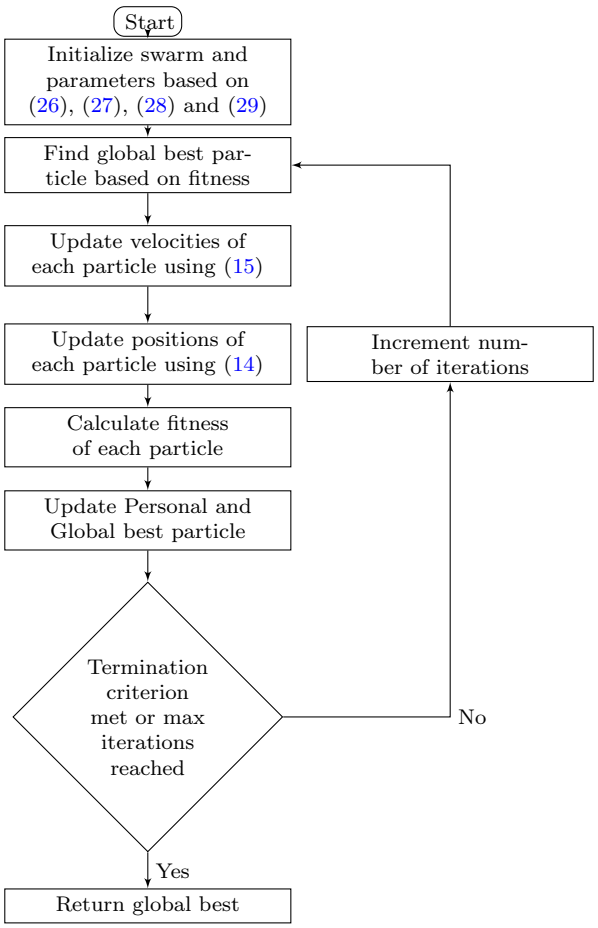
and as it models motion due to a constant acceleration a, it will have a solution:

$$x(t) = x_0 + v_0 t + \tfrac{1}{2} a t^2$$

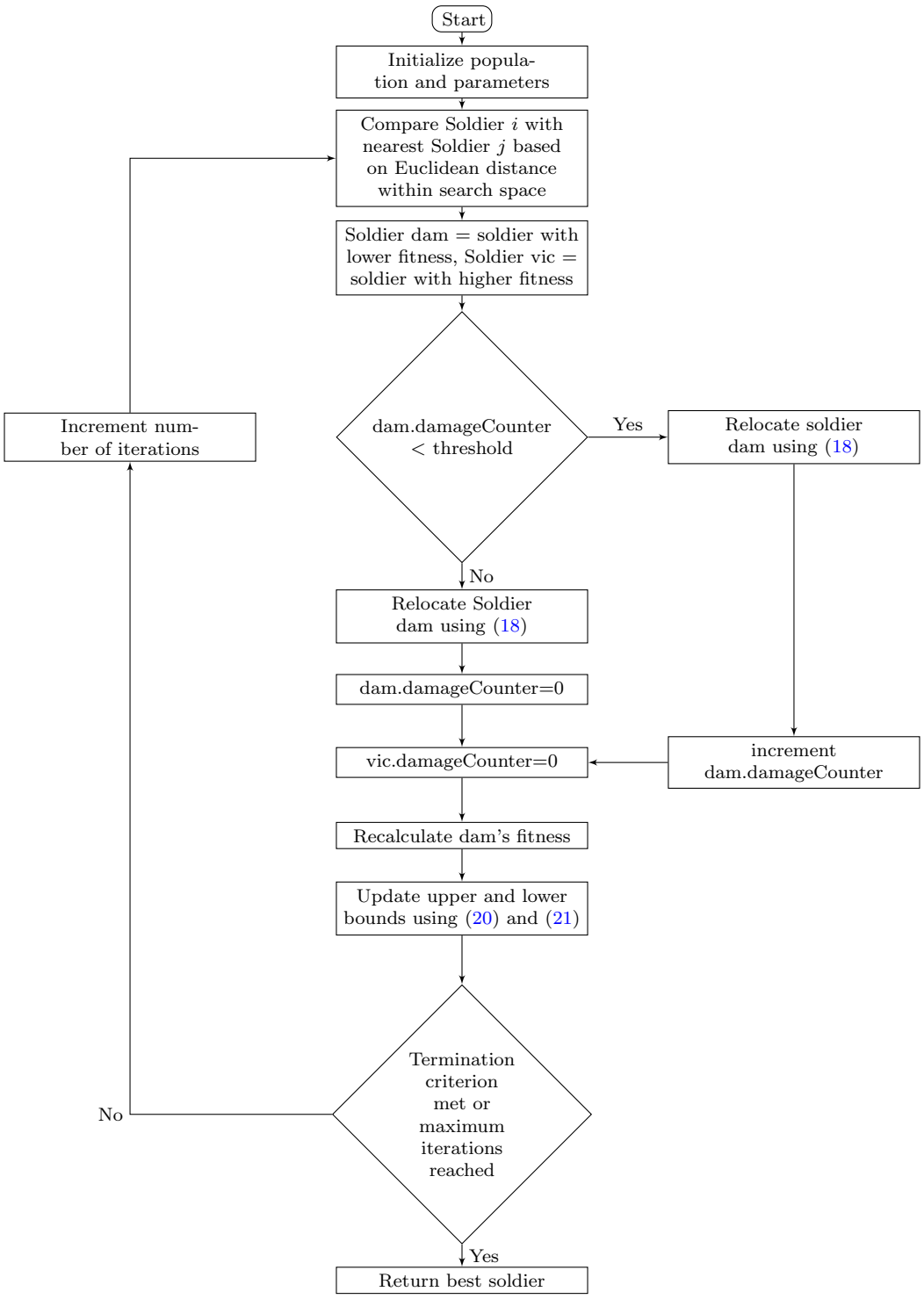
where x_0 is denoting initial position of $x(0)$ and v_0 is denoting the initial speed $\dot{x}(0)$. Also, it is important to note that there is a need for two initial conditions to determine $x(t)$ which itself is a general property for the second-order differential equation. If we compute the second derivative we can verify the solution:

$$\begin{aligned} \frac{d^2x}{dt^2} &= \frac{d^2}{dt^2}(x_0 + v_0 t + \frac{1}{2} a t^2) \\ &= \frac{d}{dt}(v_0 + at) \\ &= a \end{aligned}$$

APPENDIX B: Flowchart PSO



APPENDIX C: Flowchart BRO



APPENDIX D: Numerical Solvers' Experiments Specification Table

Specification	
initial x position	0
initial y position	0
initial x velocity	4
initial y velocity	0
kinetic friction	0.1
static friction	0.2
height function	$z = 0.2x + 5$
step-sizes used	0.2, 0.1, 0.05, 0.025, 0.0125, 0.00625, 0.003125

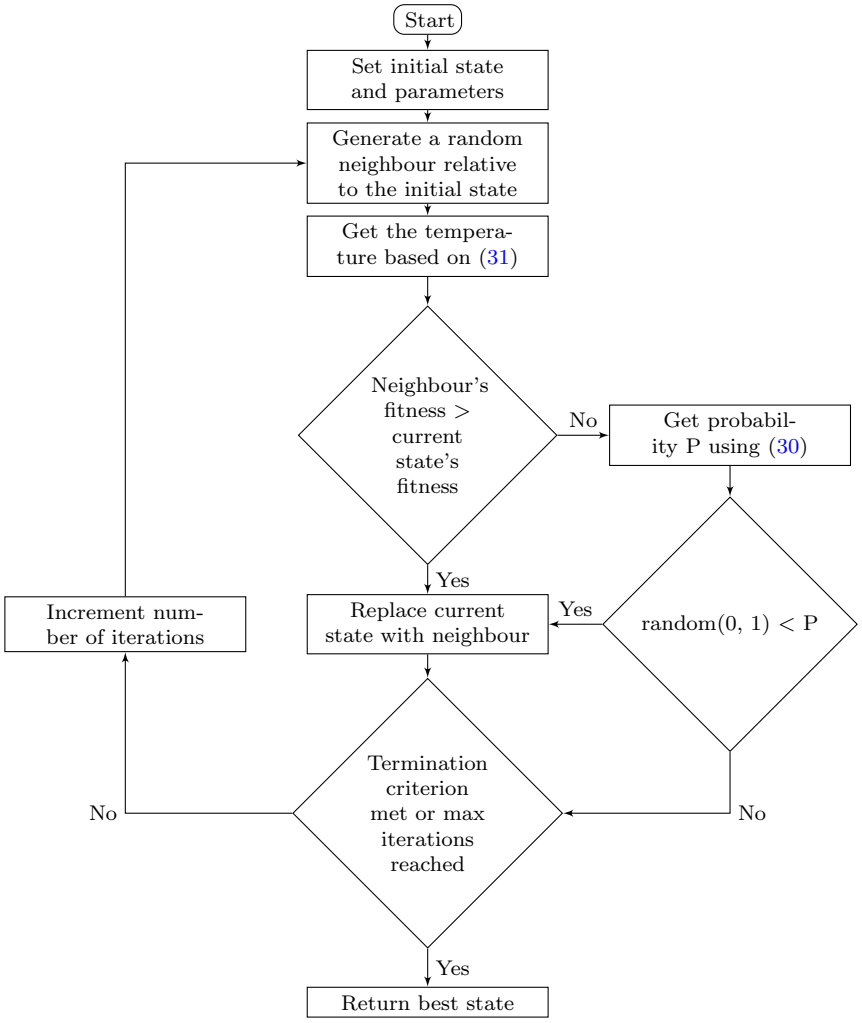
APPENDIX E: AI Experiments Specification Table

Obstacles experiments specification	
initial ball position (x,y)	(-4,0)
target position (x,y)	(4,1)
hole radius size	0.15
numerical solver	RK4
kinetic friction	0.08
static friction	0.2
number of trees used	0, 10, 20, 30, 40

Randomized height profiles experiments specification	
initial ball position (x,y)	(-3,0)
target position (x,y)	(4,0)
hole radius size	0.15
number of trees	0
numerical solver	RK4
kinetic friction	0.08
static friction	0.2
height function	0.1

Maze-like courses experiments specification	
hole radius size	0.15
number of trees	0
numerical solver	RK4
kinetic friction	0.08
static friction	0.2
height function	0.1
maze complexities used	3, 4, 5, 6, 7, 8

APPENDIX F: Flowchart SA



APPENDIX G: Physics Implementation

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	
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: Parameters, parameter ranges and restrictions [9].

APPENDIX H: UI screenshots

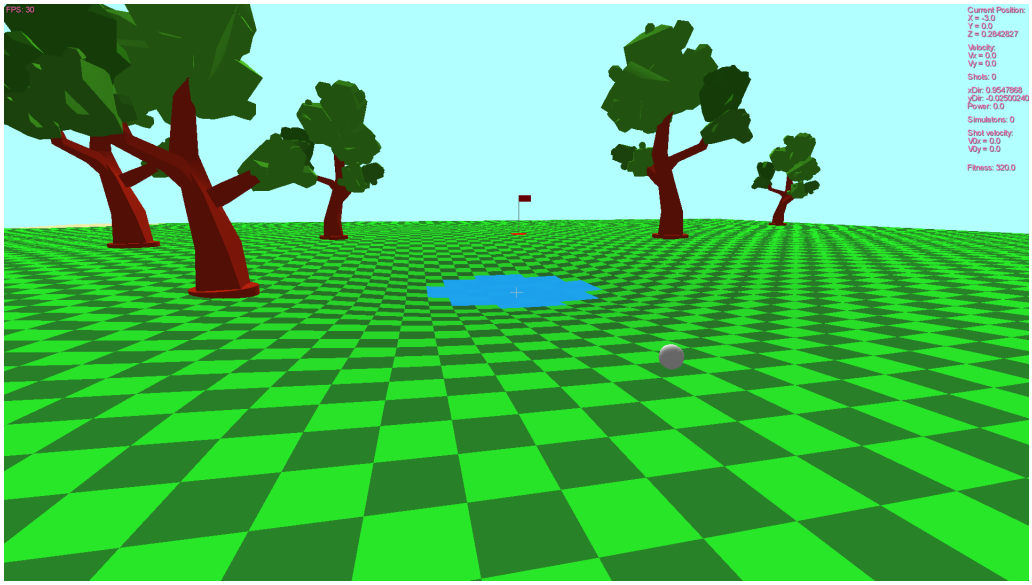


Figure 13: A screenshot that shows the golf course with some trees.

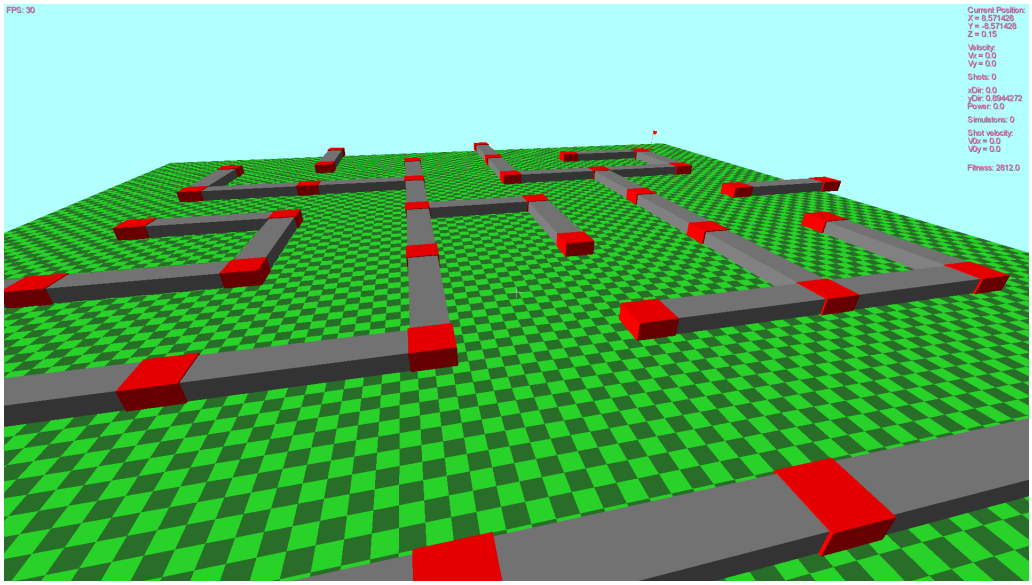
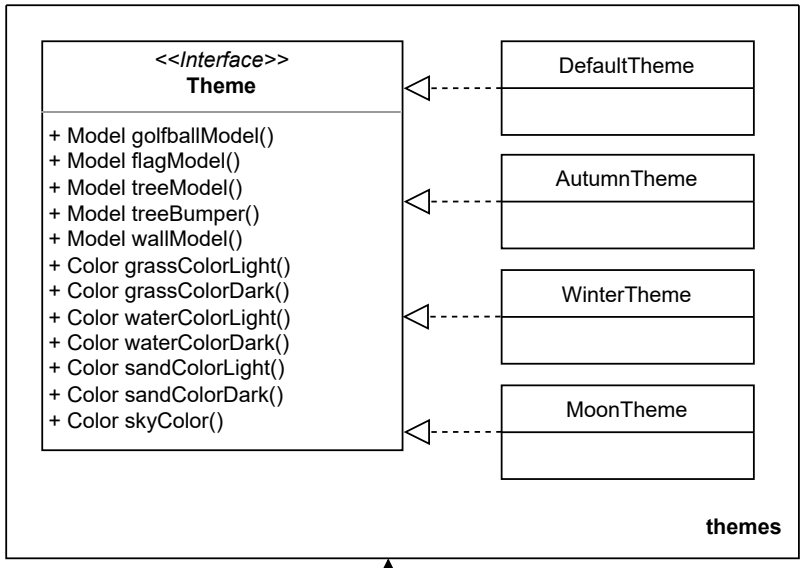


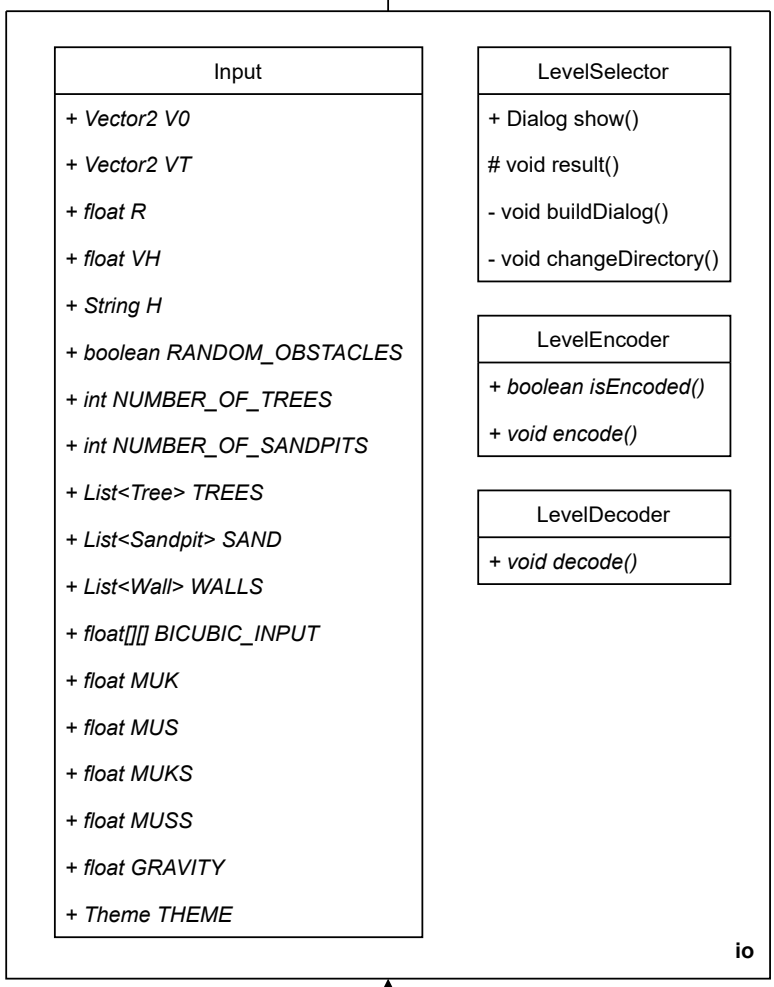
Figure 14: A screenshot that shows a maze course.

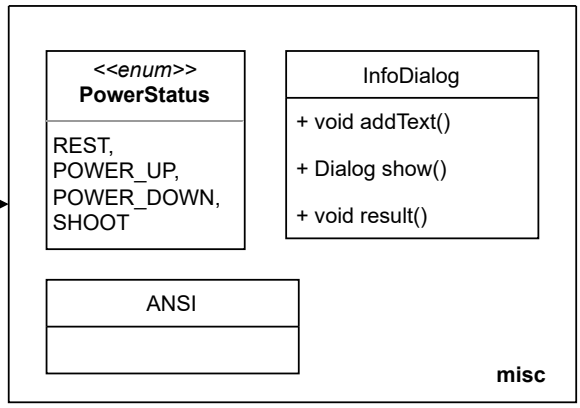
APPENDIX I: Full UML diagram

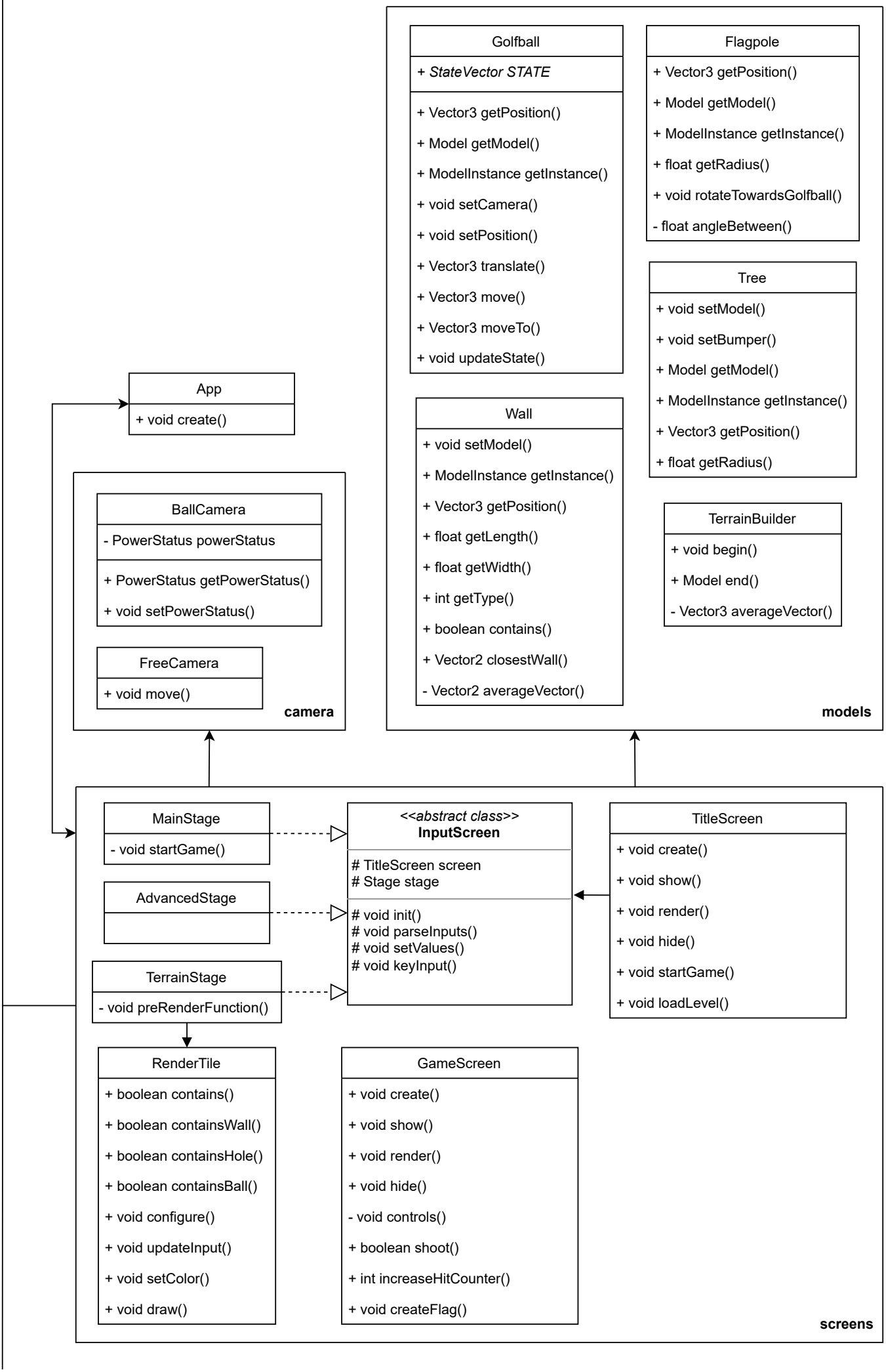


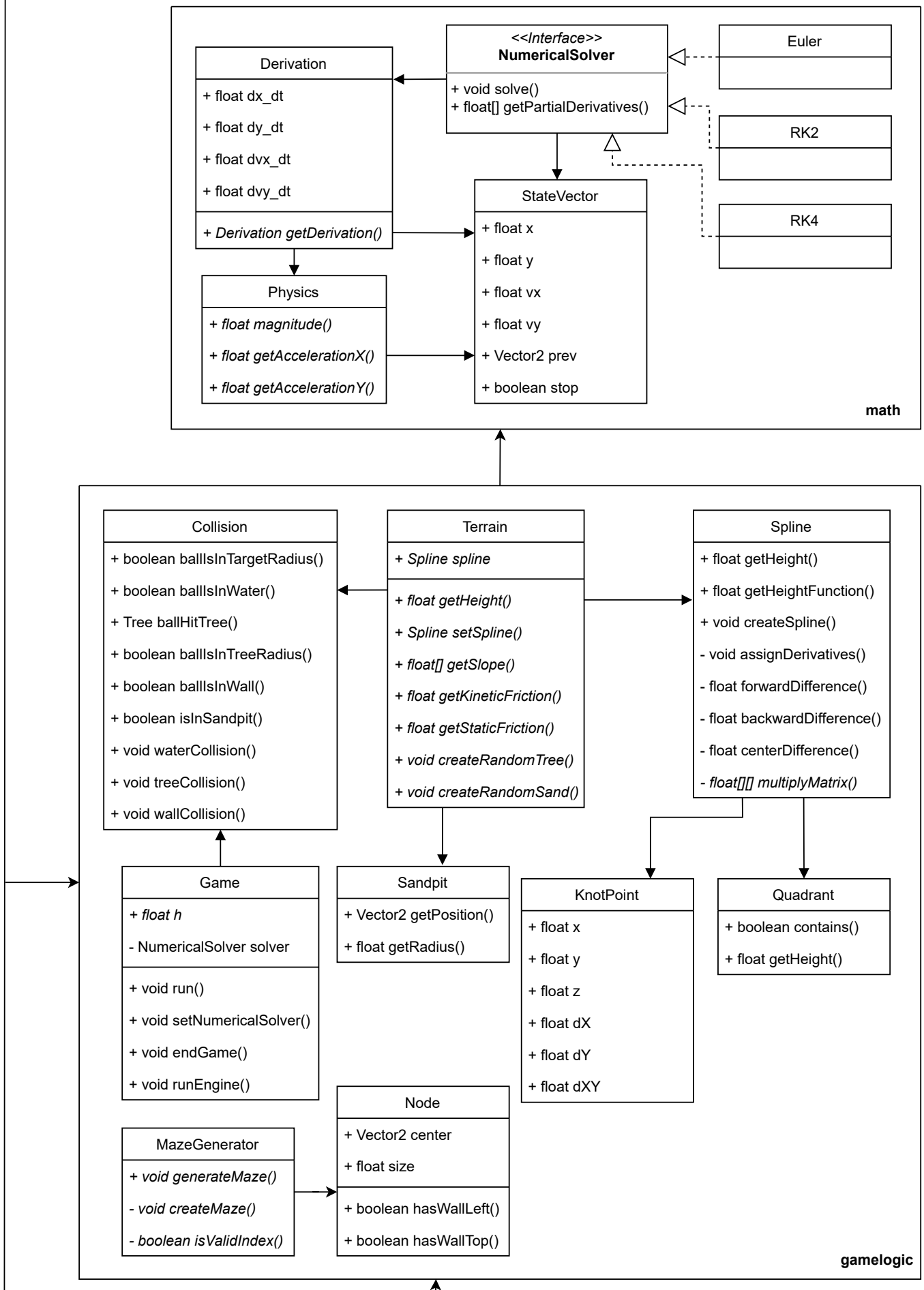
UML Diagram

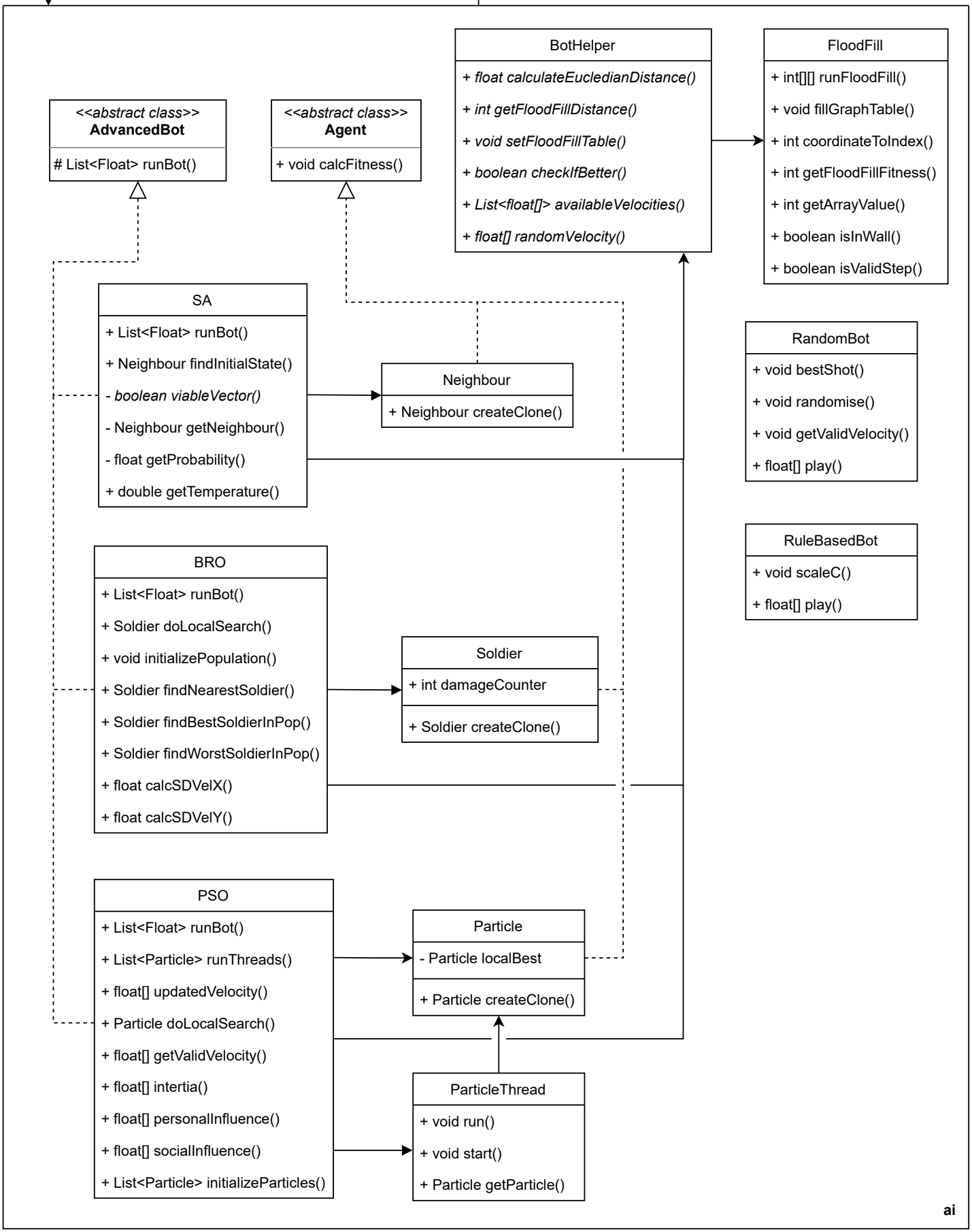
Pie de Boer
Thomas Vroom
Laurent Bijman
Rafali Arfan
Tom Bakker
Mohammed Al-Azzani
Papuna Berdzulishvili






Golfball
+ StateVector STATE
+ Vector3 getPosition()
+ Model getModel()
+ ModelInstance getInstance()
+ void setCamera()
+ void setPosition()
+ Vector3 translate()
+ Vector3 move()
+ Vector3 moveTo()
+ void updateState()
Flagpole
+ Vector3 getPosition()
+ Model getModel()
+ ModelInstance getInstance()
+ float getRadius()
+ void rotateTowardsGolfball()
- float angleBetween()
Tree
+ void setModel()
+ void setBumper()
+ Model getModel()
+ ModelInstance getInstance()
+ Vector3 getPosition()
+ float getRadius()
Wall
+ void setModel()
+ ModelInstance getInstance()
+ Vector3 getPosition()
+ float getLength()
+ float getWidth()
+ int getType()
+ boolean contains()
+ Vector2 closestWall()
- Vector2 averageVector()
TerrainBuilder
+ void begin()
+ Model end()
- Vector3 averageVector()
models
App
+ void create()
BallCamera
- PowerStatus powerStatus
+ PowerStatus getPowerStatus()
+ void setPowerStatus()
FreeCamera
+ void move()
camera
MainStage
- void startGame()
AdvancedStage
TerrainStage
- void preRenderFunction()
<<abstract class>>
InputScreen
TitleScreen screen
Stage stage
void init()
void parseInputs()
void setValues()
void keyInput()
TitleScreen
+ void create()
+ void show()
+ void render()
+ void hide()
+ void startGame()
+ void loadLevel()
RenderTile
+ boolean contains()
+ boolean containsWall()
+ boolean containsHole()
+ boolean containsBall()
+ void configure()
+ void updateInput()
+ void setColor()
+ void draw()
GameScreen
+ void create()
+ void show()
+ void render()
+ void hide()
- void controls()
+ boolean shoot()
+ int increaseHitCounter()
+ void createFlag()
screens


Derivation
+ float dx_dt
+ float dy_dt
+ float dvx_dt
+ float dvy_dt
+ Derivation getDerivation()
<<Interface>>
NumericalSolver
+ void solve()
+ float[] getPartialDerivatives()
Euler
RK2
RK4
StateVector
+ float x
+ float y
+ float vx
+ float vy
+ Vector2 prev
+ boolean stop
Physics
+ float magnitude()
+ float getAccelerationX()
+ float getAccelerationY()
math
Collision
+ boolean ballIsInTargetRadius()
+ boolean ballIsInWater()
+ Tree ballHitTree()
+ boolean ballIsInTreeRadius()
+ boolean ballIsInWall()
+ boolean isInSandpit()
+ void waterCollision()
+ void treeCollision()
+ void wallCollision()
Terrain
+ Spline spline
+ float getHeight()
+ Spline setSpline()
+ float[] getSlope()
+ float getKineticFriction()
+ float getStaticFriction()
+ void createRandomTree()
+ void createRandomSand()
Spline
+ float getHeight()
+ float getHeightFunction()
+ void createSpline()
- void assignDerivatives()
- float forwardDifference()
- float backwardDifference()
- float centerDifference()
- float[][] multiplyMatrix()
Game
+ float h
- NumericalSolver solver
+ void run()
+ void setNumericalSolver()
+ void endGame()
+ void runEngine()
Sandpit
+ Vector2 getPosition()
+ float getRadius()
KnotPoint
+ float x
+ float y
+ float z
+ float dX
+ float dY
+ float dXY
Quadrant
+ boolean contains()
+ float getHeight()
Node
+ Vector2 center
+ float size
+ boolean hasWallLeft()
+ boolean hasWallTop()
MazeGenerator
+ void generateMaze()
- void createMaze()
- boolean isValidIndex()
gamelogic


BotHelper
+ float calculateEucledianDistance()
+ int getFloodFillDistance()
+ void setFloodFillTable()
+ boolean checkIfBetter()
+ List<float[]> availableVelocities()
+ float[] randomVelocity()
FloodFill
+ int[][] runFloodFill()
+ void fillGraphTable()
+ int coordinateToIndex()
+ int getFloodFillFitness()
+ int getArrayValue()
+ boolean isInWall()
+ boolean isValidStep()
<<abstract class>>
AdvancedBot
List<Float> runBot()
<<abstract class>>
Agent
+ void calcFitness()
SA
+ List<Float> runBot()
+ Neighbour findInitialState()
- boolean viableVector()
- Neighbour getNeighbour()
- float getProbability()
+ double getTemperature()
Neighbour
+ Neighbour createClone()
RandomBot
+ void bestShot()
+ void randomise()
+ void getValidVelocity()
+ float[] play()
RuleBasedBot
+ void scaleC()
+ float[] play()
BRO
+ List<Float> runBot()
+ Soldier doLocalSearch()
+ void initializePopulation()
+ Soldier findNearestSoldier()
+ Soldier findBestSoldierInPop()
+ Soldier findWorstSoldierInPop()
+ float calcSDVelX()
+ float calcSDVelY()
Soldier
+ int damageCounter
+ Soldier createClone()
PSO
+ List<Float> runBot()
+ List<Particle> runThreads()
+ float[] updatedVelocity()
+ Particle doLocalSearch()
+ float[] getValidVelocity()
+ float[] intertia()
+ float[] personalInfluence()
+ float[] socialInfluence()
+ List<Particle> initializeParticles()
Particle
- Particle localBest
+ Particle createClone()
ParticleThread
+ void run()
+ void start()
+ Particle getParticle()
ai

References

[1] Edward L. Callegari Andrew John Ahluwalia Daljit S. Reiss. *Ordinary differential equations with applications*. Holt, Rinehart and Winston, New York; London, 1976.

[2] David M. Bywalec Bryan Bourg. Physics for game developers. 2013.

[3] Joanne Quinn Max Drummy. *Dive Into Deep Learning: Tools for Engagement*. Arnis Burvikovs, 2019.

[4] Ai - problem solving and search. http://www-g.eng.cam.ac.uk/mmg/teaching/artificialintelligence/nonflash/problemframenf.htm. (Accessed on 06/16/2022).

[5] Carlos A. Reyes-Sierra Margarita Coello Coello. Multi-objective particle swarm optimizers: A survey of the state-of-the-art. *IJCIR International Journal of Computational Intelligence Research*, 2(3), 2006.

[6] T. Rahkar Farshi. Battle royale optimization algorithm. *Neural Comput. Appl. Neural Computing and Applications*, 33(4):1139–1157, 2021.

[7] Rule 13 - putting greens. https://www.usga.org/content/usga/home-page/rules/rules-2019/rules-of-golf/rule-13.html. (Accessed on 06/16/2022).

[8] Otti D'Huys Christof Seiler Katharina Schneider Martijn Boussee. Project manual crazy putting. *Project Manuals*, 1(1):1–10, 2022.

[9] Paul W. Schmidt. Collision (physics). 2019.

[10] Euler H. Institutiones calculi integralis. *Opera Omnia, Vol. XI BG Teubneri Lipsiae et Berolini MCMXIII*, Volumen Primum (1768).

[11] J. Douglas Burden Richard L. Faires. *Numerical methods*. Brooks/Cole Pub. Co., Pacific Grove, CA, 1998.

[12] J. C. Butcher. On runge-kutta processes of high order. *Journal of the Australian Mathematical Society*, 4(2):179–194, 1964.

[13] Vijeyata Chauhan and Pankaj Srivastava. Computational techniques based on runge-kutta method of various order and type for solving differential equations. *International Journal of Mathematical, Engineering and Management Sciences*, 4:375–386, 02 2019.

[14] Mykel J. Wheeler Tim A. Kochenderfer. *Algorithms for optimization*. 2019.

[15] Xin-She Yang. Nature-inspired optimization algorithms. 2021.

[16] C. V. Wen X. H. Deutsch. Integrating large-scale soft data by simulated annealing and probability constraints. *Mathematical Geology*, 32(1), 2000.

[17] Riccardo Kennedy James Blackwell Tim Poli. Particle swarm optimization. *Swarm Intelligence*, 1(1):33–57, 2007.

[18] Shigang Zhou Fangfang Wang Fengjuan Wang. Effect of inertia weight w on pso-sa algorithm. *Int. J. Onl. Eng. International Journal of Online and Biomedical Engineering (iJOE)*, 9(S6):87, 2013.

[19] Andries P. Engelbrecht. Computational intelligence : an introduction. 2008.

[20] J. H. Ahlberg. Theory of splines and their applications. 2016.

[21] John Elkin Corey. *Surface interpolation by bicubic spline techniques*. Thesis, 1973.

[22] Paul Hyunjin Crawfis Roger Ohio State University Department of Computer Science Kim and Ohio State University Engineering. Intelligent maze generation. 2019.

[23] David Lee Theodore Sklar David Cohen. *Precalculus : a problems-oriented approach*. Brooks/Cole, Cengage, Belmont, MA, 2010.

[24] Mat Buckland. *Programming game AI by example*. Wordware, Plano, Tex, 2005.

[25] Donald W. Boyd. *Systems analysis and modeling : a macro-to-micro approach with multidisciplinary applications*. Academic Press, San Diego, Calif., 2001.

[26] Seyedali Mirjalili Hammoudi Abderazek, Ali Riza Yildiz. Comparison of recent optimization algorithms for design optimization of a cam-follower mechanism. 191, 2020.

[27] E.L. Ince. Ordinary differential equations. *Dover Publications, New York*, 1956.