3D Minigolf

Creating and playing complex maze-like minigolf courses

Maastricht, Netherlands, June 19th 2018

Department of Data Science and Knowledge Engineering, Maastricht University

Bachelor Year 1, Block 2.3, Group 12

Samuel Kopp, Hendrik Baacke, Siemen Geurts, Annelot Pruijn, Ruben Bijl

Abstract

The purpose of this paper is to analyse and evaluate different approaches to handle a complex maze-like course, in this report referred to as the "Mini golf-course". The focus lies on evaluating which algorithmic approach works best in the given circumstances and where the limits of capability are located for each AI. In addition to that, a small random error in the initial position of the ball gets introduced and experiments are conducted. These help to further investigate how and up to which point the different AIs are able to find solutions to circumvent this problem. Apart from the main focus, the report introduces the most interesting parts of the physics framework and the group's take on the game-visualisation techniques. This paper draws the conclusions made from a combination of online resources and the use of heuristics in form of a practical implementation of various types of bots. The bots performances are then being compared, so that applicable suggestions can be derived.

Table of Contents

Introduction

The assignment is to create a simulation program for a golf course. It has to capture all the excitement of a real life putting experience. The program should realistically simulate the physics of real-life putting, and allow players to play against each other or an artificial intelligence player, called a bot. The program should also allow users to create their own putting greens.

The terrain of the game is described as a function h, giving: z = h(x,y), where z is the height at every point x,y. The terrain also has bodies of water, which are found where the height z is negative. Next to this the terrain will have obstacles, which will make the game more difficult for the player and bot.
The ball has to make its way across the green from its starting point to the target, while avoiding any pools of water and obstacles. If the ball rolls into the water, the player incurs a penalty. The goal for the player and the bot is to guide the ball into the target with as few shots as possible.

Simulating three-dimensional environments and manipulating physical objects is a problem which appears in a range of areas from robotics to video games. In this report, the computer simulation of a ball moving on a surface with slopes, obstacles and physical constraints like gravity and surface friction is presented. This simulation is used to create a video game version of a mini-golf course.
This report will first give an overview about the theory used for the program: Physics Engine, Collision Detection, Terrain Generation and Visualisation Techniques. Then it will discuss the AI algorithm used in the program.

The research questions to be answered in this report are whether it is possible to extend the previously programmed Bots to handle maze-like courses and what the limits of each bot implementation are.
Moreover is it interesting to see how the different AI-algorithms handle a small error in the initial position of the ball. How does the varying of this noise affect the computational outcome and which strategies does each Bot follow? Is there a course design for which the derived strategy from the noise case is better compared to the conventional strategy?
The last research question is about which bot performs best in any given circumstances.
If there is not a clear winner of the competition, what are the strengths and weaknesses of each AI?

1. Theory

1.1 Physics Engine

In order to calculate the change of location of the ball in respect to acceleration and velocity a higher-order ordinary differential equation method was used. The Runge-Kutta 4th order method can precisely approximate the course of the ball and was thus implemented in the game. This method calculates 4 separate values, each using the step size and a derivative at a certain point. Once it has calculated these values, it uses them to approximate the next point by taking the previous point and adding a sum of these values.

$$k_{i,1} = h_i f(t_i, w_i)$$
$$k_{i,2} = h_i f(t_i + \tfrac{1}{2}h_i, w_i + \tfrac{1}{2}k_{i,1})$$
$$k_{i,3} = h_i f(t_i + \tfrac{1}{2}h_i, w_i + \tfrac{1}{2}k_{i,2})$$
$$k_{i,4} = h_i f(t_i + h_i, w_i + k_{i,3})$$

Formulae 1- 4: Evaluation of the derivative function multiplied by the stepsize at 4 different points

With:

t_i = *time at step i*
w_i = *function approximation at* t_i
h_i = *the timestep, i.e.* $t_{i+1} - t_i$
f = *the derivative function of the desired function*

First, it calculates the four k-values. These values are function evaluations at different points between the current time and the next time step. The h_i represents the the stepsize, i.e. the difference between the current time value and the next. In the case of this game, h_i is $\frac{1}{60}s$, as the ball's position needs to be updated 60 times per second. The w_i represents the approximated function value at the current timestep. This value is used to approximate the next function value, hence the initial input to this equation, ie. the start position of the ball, has to be given in order for the equation to be applied. Once the four k-values have been calculated, they can be used to calculate the function approximation at the next time step using the formula below.

$$w_{i+1} = w_i + \tfrac{1}{6}(k_{i,1} + 2k_{i,2} + 2k_{i,3} + k_{i,4})$$

Formula 5: Final equation for Runge-Kutta 4th order

However, in the case of minigolf, the method of acceleration is given, which is the second derivative of the location, hence the Runge-Kutta 4th order method needs to be implemented twice. In the first layer the velocity was calculated at a certain point, which can then be used again to calculate the location of the ball at a certain point.

Since the location of a ball exists of 2 coordinates, the change in x and y coordinates need to be calculated separately but simultaneously. Hence, our method consisted of two layers: one for finding the velocity and another for finding the location; and within each layer Runge Kutta was carried out twice, once for the x-coordinate and once for the y-coordinate.

In preliminary versions of the game, Euler's method was used for the physics engine. Euler's method is defined as:

$$w_{i+1} = w_i + h * f(t_i,\ w_i)$$

Formula 6: Euler's method

Although the implementation of this method is a lot easier and it runs quicker, the accuracy is sacrificed.

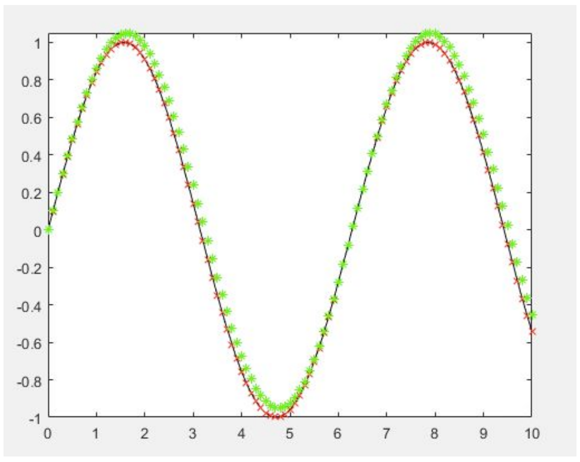


Fig. 1: RK4: Red, Euler: Green
Stepsize = $\frac{1}{10}$

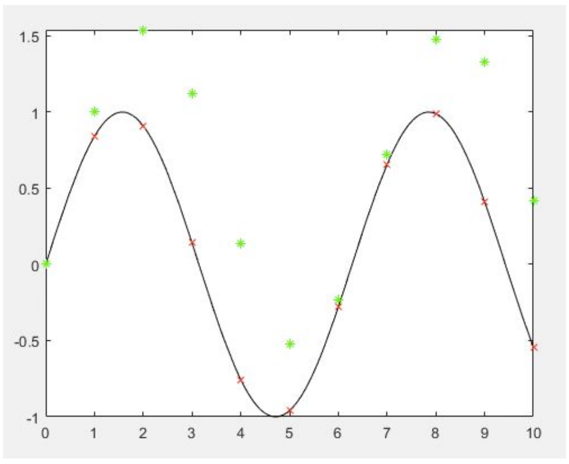


Fig. 2: RK4: Red, Euler: Green
Stepsize = 1

In the first graph, Euler's method and Runge-Kutta's fourth order method both approximate the graph quite well, although it is already visible that Runge-Kutta is more accurate. In the second graph, the step size is increased, reducing the accuracy of Euler's method while Runge-Kutta remains very accurate.++

1.2 Collision Detection

In order to perform collision when the ball hits an object, an advanced detection based on the principle of Ray Casting has been implemented. When the ball moves, the position of the ball serves as the point from which a vector, the "ray cast", is spanned into the direction of movement. At each time step, which is 1/60 of a second, a check whether the ball is inside an object is performed. When that is the case, the program detects which object got hit and the ball bounces off according to its initial vector and the object's surface. Hence the X value is replaced by the Y value and vice versa. In addition to check for the ball to be "in" an object, the program also determines whether the ball is located outside the predefined grid. This is done, because there is a possibility that the ball "clips through" an object if its velocity is high enough. Essentially, the ball moves further than the objects diameter in the 0.01667s timestep without checking. This is the reason why the program does not only check which object the ball has collided with, but also checks in a range around the object, whether there are other objects with which the ball might have collided with.

- **Balls position ray casting vector (movement of the ball is the ray cast**
- **At each time instance, there is a check whether the ball collided with an object**
- **If yes, we check which object and the ball acts accordingly**
- **For each object there is different behaviour**
- **-timestep is 1/60**
- **Also check whether the ball went outside the grid, because if a certain velocity is reached that behavior occurs**
- **In addition to check which object the ball has collided with, we also check in a range around the object whether there are other objects with which the ball might've collided with**
- **We do so, because there is a possibility that the ball "clips through" and object if its velocity is big enough.**

1.3 Terrain Generation

In order to create an environment for bots to be tested in, a course needs to be generated, either randomly or given a particular input. These inputs should not be too specific, however, as this could cause confusion. In order to minimise this, a minimum required set of data should be asked for. In this particular case, the height of the terrain. However, a too large set of inputs may cause a large amount of manual labor, which is undesirable. As such, a small amount of height value is required to generate a given terrain. However, to apply physics to this terrain and create a gradual slope between these points, more information is required. In order to gain this information, cubic splines were implemented in the form of bicubic interpolation (BI). Cubic splines are a way of generating a cubic polynomial between two points, given only the x- and y-position. BI is a very similar method for creating a gradual slope between a set of points, the difference being cubic splines work in $\mathbb{R}^2$ and BI in $\mathbb{R}^3$. The easiest way of working with BI is on a regular grid of unit squares (1x1). The idea is to create a formula with a series of coefficients for each square, similar to how cubic splines create a formula with a series of coefficient for each interval between points. We can set up this formula using the following logic:

$$p(x,y) = \sum_{i=0}^{3} \sum_{j=0}^{3} a_{ij} x^i y^j.$$

$$p_x(x,y) = \sum_{i=1}^{3} \sum_{j=0}^{3} a_{ij} i x^{i-1} y^j,$$

$$p_y(x,y) = \sum_{i=0}^{3} \sum_{j=1}^{3} a_{ij} x^i j y^{j-1},$$

$$p_{xy}(x,y) = \sum_{i=1}^{3} \sum_{j=1}^{3} a_{ij} i x^{i-1} j y^{j-1}.$$

Formula 7-10: Formulae for bicubic interpolation

where *p(x,y)* is the height value for a given (x, y)-position, $p_x(x,y)$ the derivative in the x-direction, $p_y(x,y)$ the derivative in the y-direction and $p_{xy}(x,y)$ the x-derivative in the y-direction. Because the user has input a set of height values, an SLE can be generated after using these values to find the derivatives, which will return the series of coefficients *a* mentioned in the formula. Using these, the intermediate values can be calculated using the above formulas.

1.4 Visualisation Techniques

In order to visualise the terrain mentioned beforehand, 3rd party software is used, namely JavaFX. From its wide toolkit, only several features are required for this particular project; Triangle Meshes, Boxes and FXML for user input. Visualising the terrain can be done in two ways. One makes use of the Triangle Mesh and creates a smooth and slick curved surface which is easy on the eyes, but difficult in terms of readability (height values are hard to distinguish). In order to counteract its weaknesses, a second form of visualisation is implemented using the Boxes. The Boxes have a large computational impact, but by coloring these boxes, an accurate height map can be created, which makes it easier to distinguish and display altitude levels.
A desirable outcome is to combine the two visualisation techniques to create a map where both these assets are optimised, ie. a smooth surface while also showing the height clearly to the user.

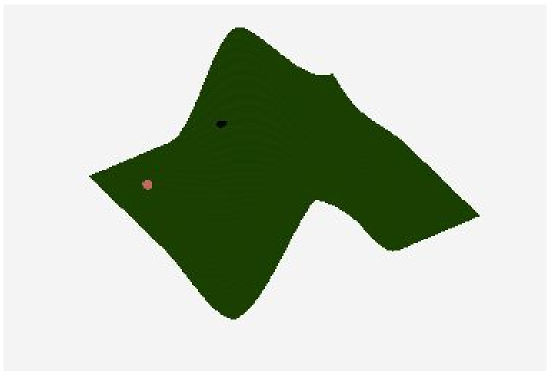


Fig. 3: JavaFX Triangle Mesh

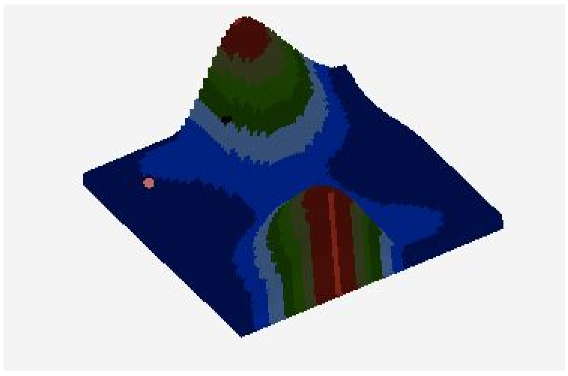


Fig. 4: Course Map using the Height map

2. AI Algorithms

Hitting a hole-in-one can be done by finding an initial velocity vector which will let the ball move into a target on the course before it stops. The amount of possible vectors for this approach can range from none to infinite. A maximum force of hitting the ball is introduced to make the game more realistic and more difficult.
The problem of hitting the ball into the target is separated into three stages. Initially the ball has to find its way into the target without any obstacles in his way. Then water, walls, mountains and trees are added to the course to create a maze-like environment. In the last stage a noise factor is added to the initial position of the ball and to its initial velocity in order to investigate how robust the algorithm is, when faced with small errors.
At all times it is a desirable characteristic to have the amount of function evaluations, and with that the needed computation time, as low as possible. This is also a characteristic which makes it possible to compare the efficiency of the Bots.
Various approaches to solve this problem are presented and compared.

--Add: keeping the amount of function evaluations low--

1. Hole-In-One without obstacles

This problem is a Continuous-Optimisation-Problem where f(x) is a function which returns the distance of the closest distance the ball is located to the target in the movement path before it stops. f(x): R2 {|x|< maxV} -> R with 2 dimensional vector x
-insert latex- -insert picture-

a) <u>Local search with geometric heuristic</u>

The first and the intuitive approach is to simulate a ball hit initially with the vector angle
that points straight to the hole. -insert picture-
Then the distance to the hole can be evaluated and the angle can be shifted
locally to reduce the distance until a hole-in-one was simulated.
The direction in which the angle has to be shifted, the magnitude of the shift and
the initial magnitude of the force of the vector have to be chosen.

To determine whether the angle of needs to be shifted to the left or to the right, the deviation to the start-hole line is computed. **-insert formula- -insert picture-**

The magnitude of the angle shift initially is set to a an initial high value and decreased every time the search bounces around the same value. -insert picture-
This way also solutions vectors with fractional numbers can be found.
The initial angle shift is empirically decided to be in between 5-10, which is
also often the expected search range for hole-in-ones with no obstacles.

In the case of the ball stopping before it reaches the hole and the
deviation of the start-hole line being zero, the magnitude
of the vector is increased by a constant factor of 1.2.

If the shift in angle parameter reaches the smallest possible value, the search is
trapped in a local minima with no solution. Because the angle shift is not small enough
to leave this minima anymore, the search is stopped and the parameters are reset.

To enable searches with extended the search space, the procedure is repeated
in different magnitudes of initial vector force, ranging from the maximum velocity force
to the lowest possible force that can move the ball.

b) Newton-Raphson Method

Another perspective on this problem is to see it as a root-finding problem,
where an input vector which results in a zero distance to the hole has to be found.
$f(x) = 0$;

In numerical analysis a method for finding successively better approximations to roots
is the root-finding algorithm "Newton-Raphson method" named after the two british
mathematicians.
The method starts with an initial guess which, here again, is the line spanned by the Start and
Hole Position, and then approximates the function by its tangent line. Then it uses the x-intercept
of this tangent line to get a better approximation to the function's root than the initial guess.
This method will be iterated until a solution is found.
-insert latex-

Because the derivative of the function is unknown, a finite difference approximation of the derivative is numerically computed. Too many function evaluations have to be avoided, as the cost of them is very high. Still a precise approximation accelerates the process of root finding, because the tangent line of the method will be more accurate at each step. -
$f'(x) = f(x + h) - f(x - h)\ 2h - h26\ f'''(\xi)$- latex-

A step size h of 0.0001 is used, as this results empirically in the fastest search.
The actual error is unknown,because there is no accurate information about the derivatives.
A larger step size results in a less accurate approximations
and a too small step size is not optimal, because small changes in values
can cause large changes in the derivative.

When the method results in an iteration with a velocity which magnitude is above
the maximum velocity the vector is scaled by a half.

This method is expected to converge, as the initial guess should be close enough to the root of the function. The function is expected to fail if the starting point
is stationary, since the tangent is parallel to the x-axis. In this case another starting point close to the original starting point is chosen.
The function is also expected to fail if the starting point enters a cycle.

c) Particle Swarm Optimization
reference http://www2.cs.uidaho.edu/~tsoule/cs472fall07/particleswarm.pdf
https://www.cs.tufts.edu/comp/150GA/homeworks/hw3/_reading6%201995%20particle%20swarming.pdf

As both previous algorithms are expected in some scenarios to converge locally
without finding a solution, a third algorithm is introduced.
Population based metaheuristics try to improve a candidate solution over the range of the search space. They can find solutions in a different part of the search space
as algorithm's which search locally from an initial guess.

Particle Swarm Optimization(PSO) is a population based stochastic
optimization algorithm developed by Dr. Eberhart and Dr. Kennedy in 1995,
inspired by social behavior of bird flocking or fish schooling.
It tries to iteratively improve a candidate solution with regard to a fitness value.
The fitness value in this case is the euclidean distance of a shot to the hole.

The algorithm has exploitative behaviour while still converging to a minima quickly
compared to other global search strategies(e.g genetic algorithm, simulated annealing).
This combination of exploitation and fast convergences is expected to improve
the hole-in-one search in difficult scenarios.

Particle Swarm starts with an initial population(a small population size is used here, as function evaluations are costly) with a random position in the search space. The positions represent a velocity vector for a shot. Iteratively the position of each particle is updated by a velocity vector(velocity in regards of movement in search space, not in sense of hitting the ball)
The velocity vector is updated by an inertia term, a cognitive and a social term. The inertia term represents a particle's current velocity. The cognitive term directs the particle towards its personal best position in all iterations and the social term moves the particle towards the position of the globally best particle.
-insert figure- -insert pseudocode-

For each term there is a parameter, which must be chosen to balance between exploration and exploitation to avoid premature convergence.
They are empirically chosen to be large for the social term in comparison to the cognitive and the inertia term(1;0.5;0.25).
A damping coefficient traditionally added to reduce the impact of the inertia term over time.
-insert latex-

2. Hole-In-One with obstacles, water and mazes

Adding obstacles and mazes to the course, makes the function more chaotic. On input vectors where the ball collides with an obstacle or with water the function is discontinuous(hitting the water maps the function to the initial position) and not differentiable.
Because in the implementation of the newton method the derivative is approximated, the method handles these situations as if a derivative would exist at this point.

The local search with geometric heuristic is not expected to work as the start-hole line is not a good initial guess anymore. The newton's method is expected to fail at some points as function can be not continuously differentiable around the neighborhood of the root. A particle swarm optimization may possibly find a correct solution by adjusting its parameters accordingly, but it will take a long time until it converges to the correct local minima, where a solution is located as the function output is highly chaotic for mazes.
To improve the heuristics of the search a pathfinding algorithm has to be integrated.

<u>--------------------------------</u>

<u>A * Pathfinding</u>

The course is divided into a grid on which objects such as trees and walls exist. This obstacle grid is used in the A* algorithm in order to find the shortest route through the maze. It creates nodes for every space in the grid and subsequently gives them specific values determined by which obstacle is found on that space.

The algorithm starts at the initial node. It then considers which nodes are possible as next nodes and adds these to a priority queue. When determining which neighbours to add to the priority queue, the algorithm takes two things into account. First, the ball can't move to a node which contains an object. Second, when two walls are bordering diagonally, the algorithm doesn't consider the neighbour in between these two walls, since there is no possible way for the ball to travel to that node (Fig 5). The diagonal neighbour is also not considered even if only one of these walls is present, since a perfect diagonal shot isn't possible (Fig 6). However, when two trees are bordering diagonally, the neighbour in between is considered since trees are round and thus there is a bit of space for the ball to pass through (Fig 7). If there is water in the centre of a cell, the cell is essentially treated by the algorithm as if there is a wall blocking the way. Dealing with water unfolds as a difficult problem to deal with. The outlines of a water pond may be irregular compared to static objects like walls and trees. To make the algorithm more sensitive in respect of the irregular shape is definitely an aspect to improve upon in the future.

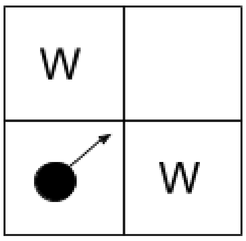


Fig 5. An illustration in which a neighbouring node is enclosed by two walls.

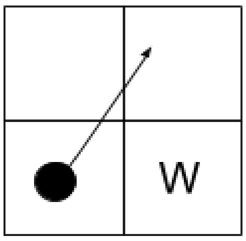


Fig 6. An illustration which showcases that a shot to the diagonal neighbour is not possible without traversing another node.

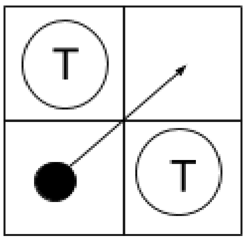


Fig 7. An illustration in which a passage to the neighbouring node is possible.

The priority of the nodes, when added to the priority queue, is determined by their f-value. Nodes with a smaller f-value are prioritized over nodes with a larger f-value. The f-value is given by the sum of two other values, the g-value and the h-value. The g-value represents the cost, or distance, of the path so far. Hence the initial node has a g-value of 0, the neighbouring nodes have a g-value of 1, and so forth. The h-value is determined by the remaining distance to the goal. This is calculated using the manhattan-distance, since this is double the minimal number of nodes that still need to be traversed in order to reach the goal. This gives more insight to how close the algorithm is to reaching the goal than the Euclidean distance would. The algorithm runs through a recursive loop for the first element of the priority queue until it reaches the goal node.

Add new distance to A*

Competition:

From this point on, two teams independently develop two algorithms that should result in optimal results in finding a hole-in-one for courses with obstacles, water and mazes. Afterwards the results for different courses are

The first team presents a combination of the A* pathfinding and the newton method, while the second team integrates the A* path to improve the heuristics of the Particle Swarm Optimization..

a) Pathfinding - Newton's (Guided Local Search)

This method first computes the shortest path from the start to the hole with the A* algorithm and then selects critical points neighboring several obstacles(corridors and edges) of this path. The newton's method is then used to iteratively optimize it's solution to find a vector that shoots a ball into a radius around the critical points of the path. The solution from one critical point is then used to initiate Newton's method for the next critical point. At last the vector that shoots the ball onto the radius of the last point n the path is used to initiate Newton's method to shoot into the hole. This can be seen as a "guided" local search, because the initial guess of the Newton's method is optimized to be closer to the root.
-insert pseudocode- -insert figure-

b)

3. Experiments and Analysis

In order to determine the performance of each algorithm in different environments a series of experiments is conducted. Indicators of a bots performance in all test cases are the path length, the computation time and the amount of computations needed to find an optimal path.

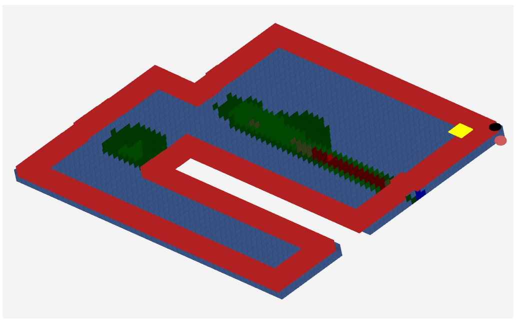


Fig. 7 :In the first experiment an A* implemented Swarm Bot competes against an A* implemented Geometric Bot. These two AIs have been established in phase two of the project. The A* algorithm is an extension for both bots. The bots are being run on a surface with few curves and edges. There are no obstacles, water or high mountains. The intention behind this procedure is to observe the average computational time of each bot on a simple course layout. Because there are no obstacles, water or mountains, the Swarm Bot is expected to perform better in this case. A straight path is more likely to lead to a quick solution.

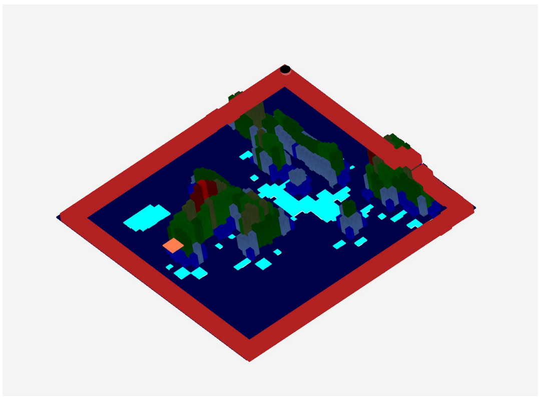


Fig. 8: The second experiment tests the performance of the two bot players in a course environment with no obstacles but vast amount of water and mountain ranges. The reason behind is to determine how the path choice of the AIs compares, when computing in an extreme terrain. The Swarm Bot is expected to perform better in this environment due to the fact that an abstract solution is more likely to be the most optimal and the Geometric Bot is not optimised for these types of solutions.

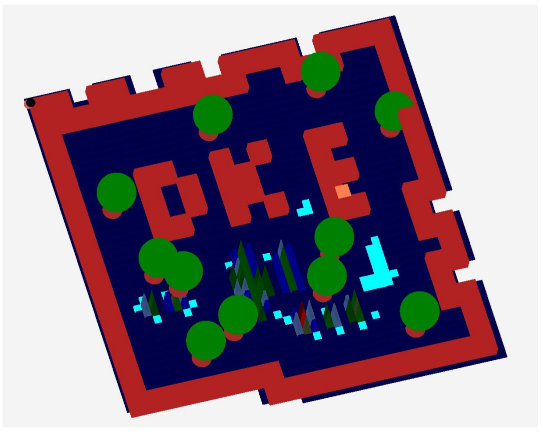


Fig. 9: The third experiment is set in a large area with obstacles, water and few mountain ranges. The intention is to retrieve data that indicates the needed amount of computations for each bot as well as the time needed to solve the course. The focus lies on the performance of the maze-solving algorithms A* and Dijkstra, but the experiment also poses a stress test for the Swarm Bot. It is expected that the comparison between the A* and the Dijkstra shows no statistically relevant outcome, whereas the Geometric Bot is expected to perform very poorly in this sort of course layout.

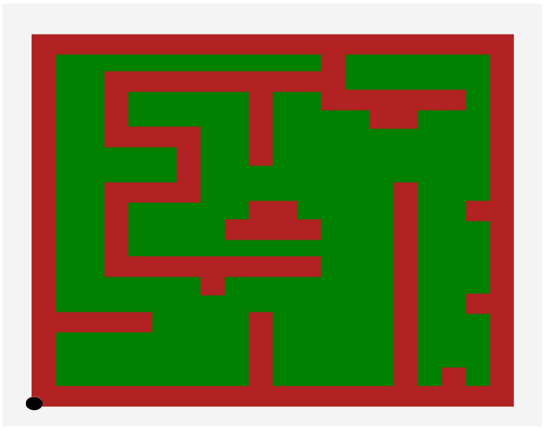
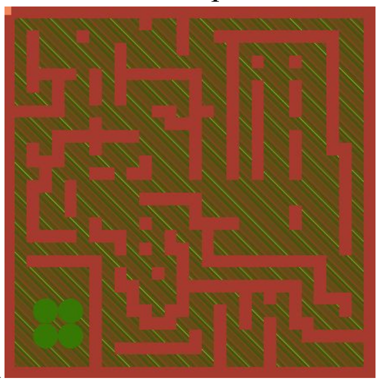


Fig. 10: A fourth experimental setup is focussed on analysing how the collision detection with ray casting performs. A bot has to operate on an environment which has an even surface and no water, but only obstacles. A* is expected to perform equally well. The difference between outcomes of the Swarm and Geometric Bot are at this point not predictable.

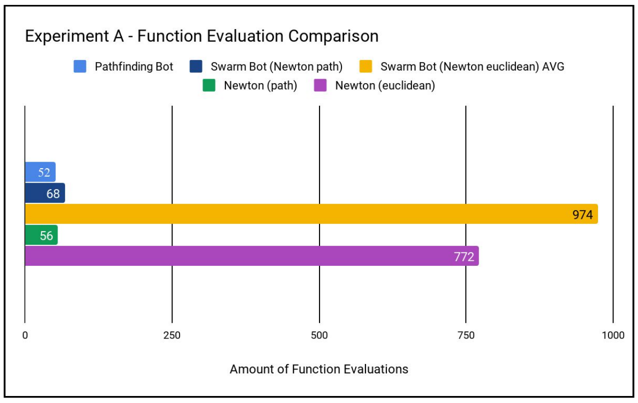
(An optional fifth experiment is to introduce a small error in the initial position of the ball. The group decides whether to do this experiment after the fourth optional task, the bot comparison/competition, has been implemented.)

Several different experiments were carried out in order to compare the bots and have them compete.

Experiment A: An initial experiment is introduced in which the bots aim to solve the maze given 

- **Experiment A:**

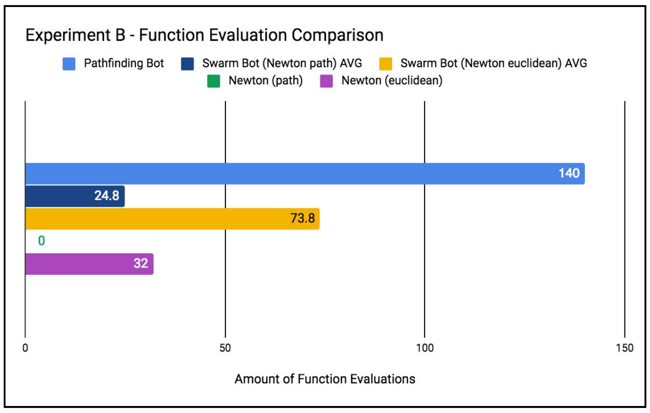
Least Evaluations needed: Pathfinding Bot
Most Evaluations needed: Swarm Bot (Newton Euclidian) (avg)
General Interpretation:



- **Experiment B with Hills and Water:**

Least Evaluations needed: Swarm Bot (Newton Path)
Most Evaluations needed: Pathfinding Bot

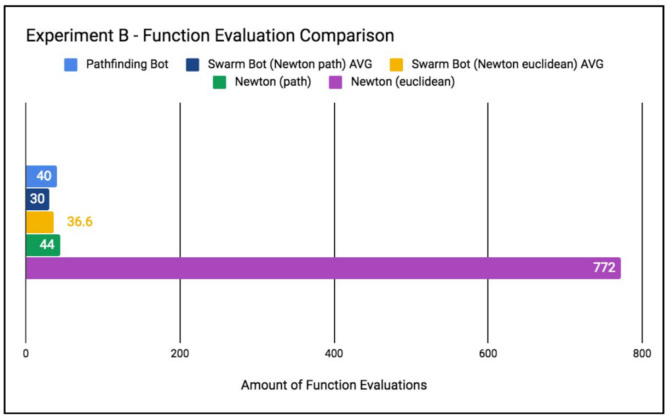
General Interpretation:
The Newton (Path) Bot is not able to find a solution due to the fact that the environment contains water, which makes the course non-differentiable for the bot.



- **Experiment B without Hills and Water:**

Least Evaluations needed: Swarm Bot (Newton Path)
Most Evaluations needed: Newton Bot (Euclidian)

General Interpretation:

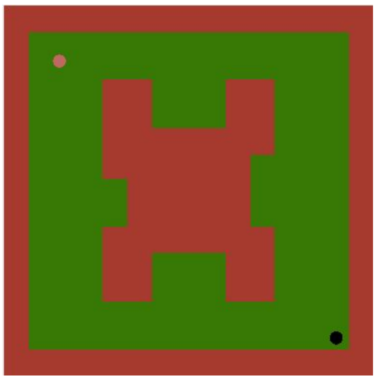


- **Experiment C With Hills and Water**

Least Evaluations needed: Swarm Bot (Newton Path)
Most Evaluations needed: Newton Bot (Euclidian)

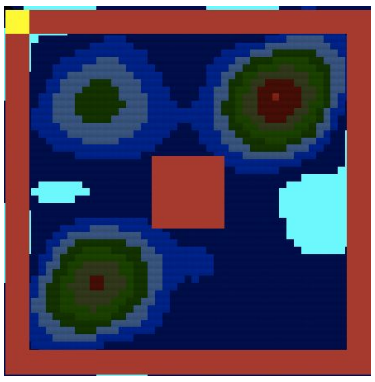
	Pathfinding Bot	Swarm Bot (Newton path)	Swarm Bot (Newton euclidean)	Newton (path)	Newton (euclidean)
A:Very hard	52	68	180 100 2152 668 1772 Av: 974,4 (high variance)	56	772
B with terrain	140	8 40	196 3	No solution (not	32

		28 24 24 Av: 24,8	2 96 72 Av: 73,8	differentiabl e because of water)	
B without terrain	40	10 40 28 32 40 Av: 30	8 48 7 8 112 Av: 36,6	44	772
C: Very easy maze (max vel 70)	72	14 5 No solution	7 5 8 180 68 Av: 53,6	No answer	12
C: Very easy maze with terrain (max vel 70)	No answer	No answer	40 36 28 32 164 Av: 60	No answer	60 (maybe no answer?)
D : extra maze	12	7 8 10 22 11 Av: 11,6	6 9 12 19 21 Av: 13,4	4	188
E: medium maze	112	28 No solution No solution No solution No solution	256 156 132 92 124 Av: 152	No solution	180



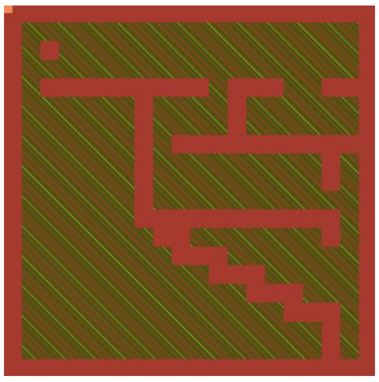
Very easy maze C

Without terrain and one big obstacle in the middle



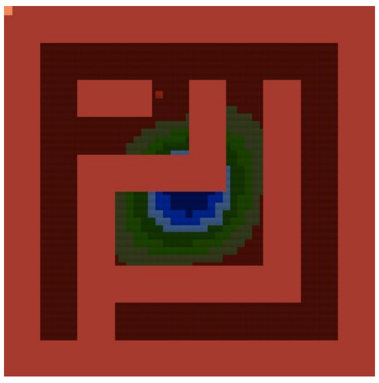
Very easy maze with terrain C

Only one obstacle in the middle, there are three hills and only two water pools.



Medium maze E

A lot of walls, no further obstacles, water or mountains. Bot needs to find a way in this real maze-like course.

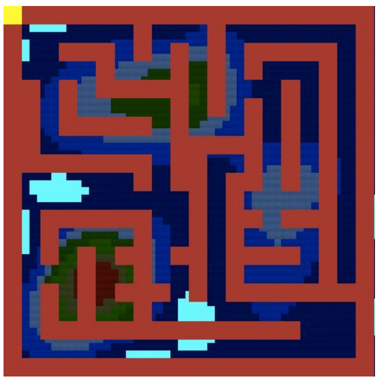


D

Small maze like course, with one big mountain in the middle. No further obstacles or water.

Very hard (medium??)

Very difficult maze with a lot of walls. Only obstacles are 4 trees in the left bottom, no further mountains or water.



Hard maze corridore (very hard??)

Very difficult maze with a lot of walls. Also a lot of water and mountains.

4. AI Comparison

5. Discussion

6. Conclusion

References

back, s. (2005, 3 17). *Very simple A* algorithm implementation*. From codeproject.com: https://www.codeproject.com/Articles/9880/Very-simple-A-algorithm-implementation

In R. Carlson, & F. Fritsch. (1989). An Algorithm for Monotone Piecewise Bicubic Interpolation.

Kaitila, C. (2013, 1 31). *A-STAR Pathfinding AI for HTML5 Canvas Games*. From buildnewgames.com: http://buildnewgames.com/astar/

J. Kennedy, R. Poli, & T. Blackwell (2007), *Particle swarm optimization.*

Mishra, A. (2016, 10 20). *How does a star algorithm work?*. From quora.com: https://www.quora.com/How-does-a-star-algorithm-work

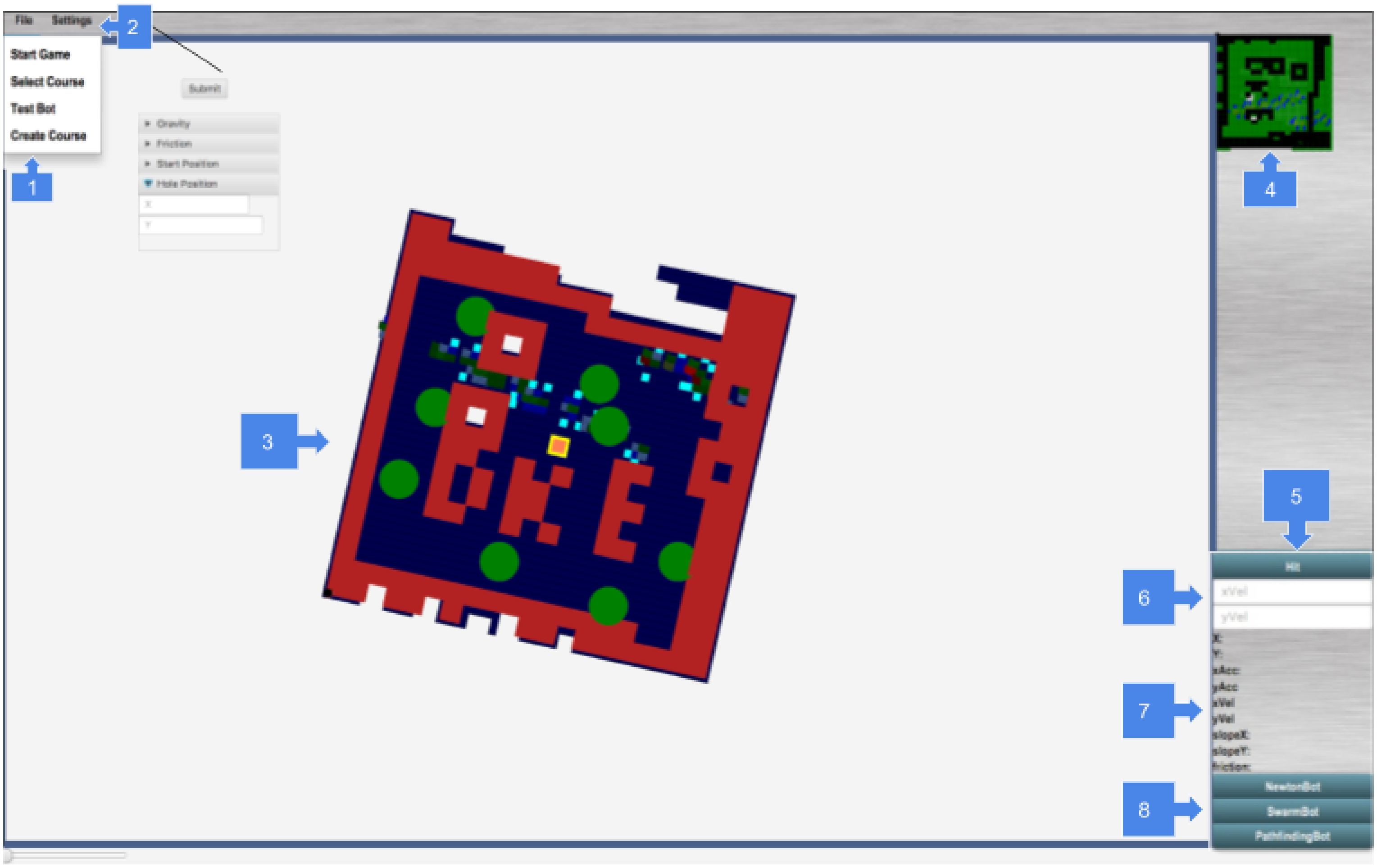
Patel, A. (1997). *Introduction to A**. From http://theory.stanford.edu: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

Reddy, H. (2013, 12 13). *PATH FINDING - Dijkstra's and A* Algorithm's*. From cs.indstate.edu: http://cs.indstate.edu/hgopireddy/algor.pdf

ROTH, S. D. (1980, 12 8). Ray Casting for Modeling Solids*. From sciencedirect.com:
Thaddeus Abiy, H. P. (sd). *A* Search* . From brilliant.org: https://brilliant.org/wiki/a-star-search/

Appendix

User Guide

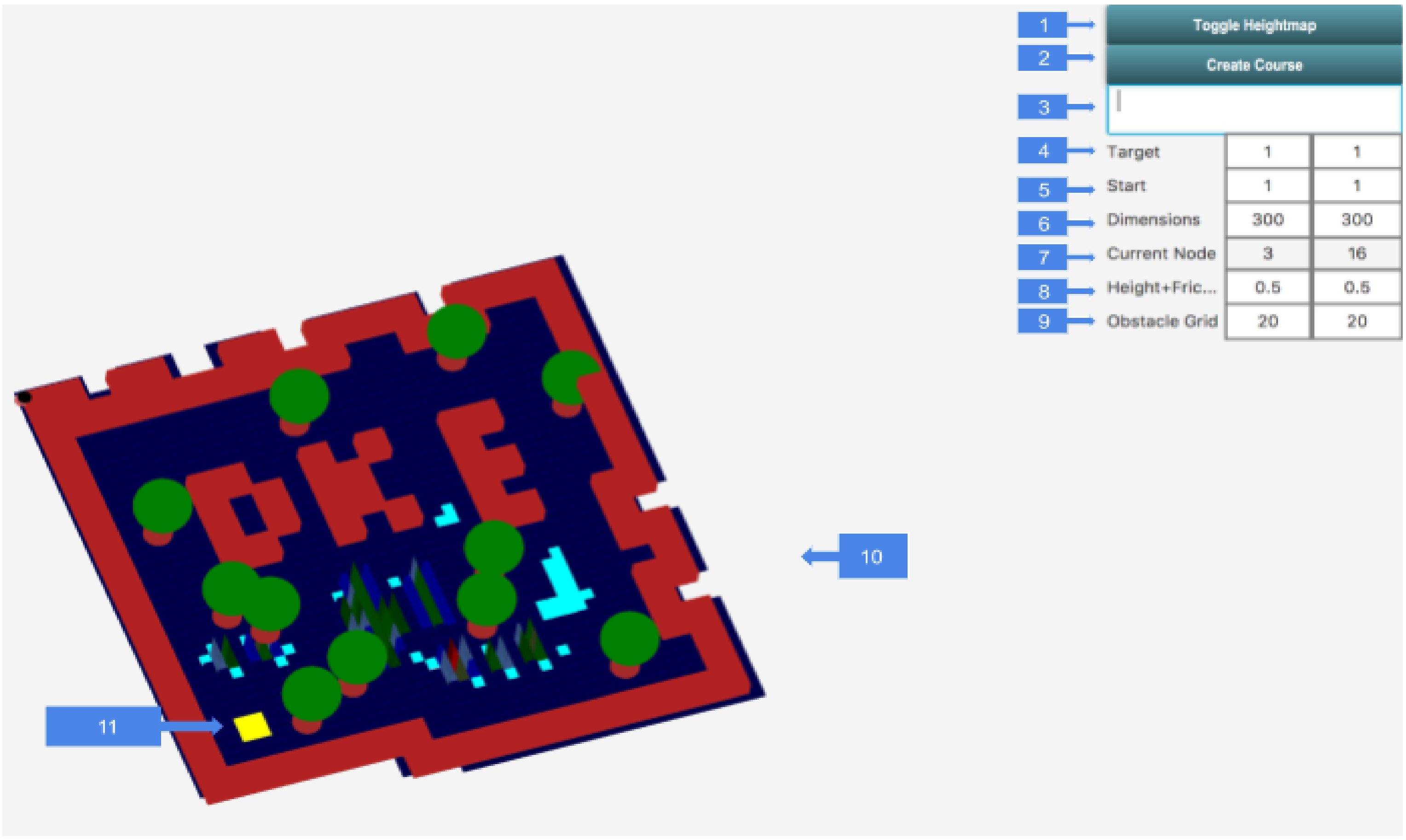


Key:

1	File Select Option: choose between Start Game, Select Course, Test Bot or Create Course
2	Settings Button: enables the user to change the physical constants as well as the Start and Goal Position
3	3D Course Visualisation, the user has the ability to move the camera Mouse Events: ● Hold Left Mouse Button and move: Rotate Course View ● Press Left and Right Mouse Button: move the Course ● Operate Mouse Wheel: zoom in or out

4	Displays the Minimap of the current course, shows the current position of the ball on the track
5	HIT Button: When pressed the Ball gets shot, based on the Parameters set
6	Interface for typing in Values for the X and Y Velocity
7	Shows the current Parameters that apply for the Ball for: ● X and Y Position ● X and Y Acceleration ● X and Y Velocity ● X and Y slope ● Friction Coefficient
8	Selection between different Bots: Newton Bot, Swarm Bot, Pathfinding Bot

Course Creator



Target	1	1
Start	1	1
Dimensions	300	300
Current Node	3	16
Height+Fric...	0.5	0.5
Obstacle Grid	20	20

Course Creator Key:

1	Switch between normal map and a height map, which displays terrain in a color based on the altitude
2	Save the named course to a file system
3	Give the course a unique name
4	Sets the Target Position with X and Y Coordinates
5	Sets the Start Position with X and Y Coordinates
6	Select the total area of the course
7	Gives Information about the Current Node that the User is applying change to
8	Select a Height of the terrain node(negative Height implies water), Select a Friction Coefficient for the node
9	Select how many objects are maximally allowed on the course
10	3D Visualisation of the Course, the user is currently designing; the user has the ability to move the camera Mouse Events: ● Hold Left Mouse Button and move: Rotate Course View ● Press Left and Right Mouse Button: move the Course ● Operate Mouse Wheel: zoom in or out
11	General Keyboard Events: W: Move tool up S: Move down A: Move left D: Move right SHIFT: change between the tools The User is able to select: <u>A</u>: A tool to place obstacles and trees (YELLOW) Keyboard Events: ● M: select Wall ● B: select Tree ● ENTER: confirm choice and place selected Object ● P: when located on the course feature: delete Object **<u>B:</u>** A tool to change the course (ORANGE) When this tool is selected, the user is able to change the Parameters from Key 4 - 9

	Keyboard Events: ● P: delete Terrain (makes it possible to create custom shaped courses)