Maastricht University

Data science & knowledge Engineering

Project year 1 phase 3

Crazy Putting

Authors:
Andreas Zurhaar
Luc Sparidans
Amanda Kane
Arnaud Ruymaekers
Marco Rietjens
Julien Elia

Supervisors:
J. Paredis
P. Collins
A. Zarras

Abstract

The purpose of this report is to exhibit a 'Crazy Putting' experience involving real world physics and an intelligent bot to play the game. Simulating the physics is possible due to the help of solver methods, such as Euler and Runge-Kutta, to calculate the velocity and position of the ball. Splines are implemented to ensure that courses are not singular in shape, but provide a wide variety to the terrain. Bots are used to play courses using heuristics to make the optimum shots when hitting the ball.

Contents

Glossary

Chunks A chunk is a part of the world, which is represented by height-fields. 17

closedSet set that contains all the nodes the algorithm visited so far. 13

Differential equation A differential equation is a mathematical equation that relates some function with its derivatives. 9

HeightFields A height-field is a set of points containing the height at that point. 17

libGDX The graphics library used to make the visual representations of what is to be displayed in this project. 17

openSet set that contains the nodes that have been discovered but have not yet been visited. 13

Taylor series An infinite sum giving the value of a function f(z) in the neighbourhood of a point a in terms of the derivatives of the function evaluated at a. 8

1 Introduction

Putting is part of the golf's game. The aim is to hit the ball with a club called the "putter," and be able to put it in a hole situated on the grass. The courses can have multiple slopes and obstacles, including trees, water and pits. The fewer times the ball is hit, the higher the player's score.

A game company wants to create a putting simulation program, called "Crazy Putting". This game must provide it's players the excitement of a real-life putting game. The player must be able to play on a predefined terrain or create a new one, with the desired slopes imported through a mathematical function. The ball should move in a realistic manner. That means that it should follow real-life physics rules. Three different game-modes should be available, which are single player, multi-player, and a "bot mode", where the game is played by an artificial intelligence. The bot should have a good knowledge of the course, and be able to strike the ball in the hole while hitting it the fewest possible times.

All the different aspects of the program are presented, starting with the generation of a new terrain from a function. The physics of the game is also covered, and the functioning of the different solvers is explained. The different bots that can be used are then presented, and we will see in details how does each one work. The noise and the collisions of the game are then covered. We will then talk about the user interface and the graphics of the game, and we will address the way the software was designed. Finally, the different tests that have been made will be explained and commented.

2 Terrain generation

To represent a golf terrain we use either use a continuous function or a spline. Both allow the calculation of heights and slopes at different points on the (x, y) plane.

2.1 Function

To represent the terrain as a function, a multi-variable function of the form $z = f(x, y)$ is needed, with x and y representing the coordinates in the horizontal direction, and z representing the height of the terrain. This function is stored as a binary expression tree (BEV) for more effective storage and simple evaluation and derivation.

The problem with this approach is that a BEV stores functions using postfix notation, which is not a natural way to represent a function. In order to keep it user friendly, the user can enter a function using infix notation which is then converted to postfix notation, while also applying the order of operations. Not all mathematical operators are supported, but the most common ones are, i.e., $+$, $-$, $*$, $/$, (,), $\hat{}$, ln, sin and cos.

The Function is partially derived in both the x and y direction, using all general derivation rules. These derivative functions are then also stored in BEV's. The function and both partial derivative functions are stored separately per course. They are used by the graphics engine for displaying the height of the course and by the physics engine for calculating the acceleration due to gravity.

2.2 Bi-cubic spline interpolation

To approximate a surface from a set of points, a convenient way was to use bi-cubic spline interpolation. Bi-cubic is the extension of cubic spline in the space instead of the plane. A cubic spline is a set of cubic functions defined by: $a_0 + a_1x + a_2x^2 + a_3x^3$. The challenge of cubic spline interpolation is to find the coefficients of each cubic function using the given data.

The coefficients of the bi-cubic splines, where each spline is on the unit square, are given by the formula:

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -3 & 3 & -2 & -1 \\ 2 & -2 & 1 & 1 \end{bmatrix} \begin{bmatrix} f(0,0) & f(0,1) & f_y(0,0) & f_y(0,1) \\ f(1,0) & f(1,1) & f_y(1,0) & f_y(1,1) \\ f_x(0,0) & f_x(0,1) & f_{xy}(0,0) & f_{xy}(0,1) \\ f_x(1,0) & f_x(1,1) & f_{xy}(1,0) & f_{xy}(1,1) \end{bmatrix} \begin{bmatrix} 1 & 0 & -3 & 2 \\ 0 & 0 & 3 & -2 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & -1 & 1 \end{bmatrix}$$

Since the field is only defined by the given data, the partial derivatives (f_x,f_y) and the cross partial derivative (f_{xy}) need to be computed.

The partial derivatives with respect to x ($f_{,x}$) are approximated by using the 5 points difference formulas. The centred difference formula (1) is used as often as possible since it has the lowest error. Close to the edges of the field it lacks of data points. The forward difference formula (3) is used to compute $f_{,x}$ when $x = 0$ and the asymmetric difference formula (2) when $x = 1$. Similarly, when $x = width$ and when $x = width - 1$, the formulas (3) and (2) are used with $h = -1$.

The same principle is applied for the partial derivatives with respect to y ($f_{,y}$) but with the formulas (4),(5) and (6).

To compute the cross partial derivatives (f_{xy}), the $f_{,x}$ values computed earlier on the y axis are used. The formulas (7), (8) and (9) are used with the same principle as to computes $f_{,x}$ and $f_{,y}$.

$$f_{,x}(x,y) = \frac{f(x-2h,y) + 8f(x+h,y) - 8f(x-h,y) + f(x+2h,y)}{12h} \tag{1}$$

$$f_{,x}(x,y) = \frac{-3f(x-h,y) - 10f(x,y) + 18f(x+h,y) - 6f(x+2h,y) + f(x+3h,y)}{12h} \tag{2}$$

$$f_{,x}(x,y) = \frac{-25f(x,y) + 48f(x+h,y) - 36f(x+2h,y) + 16f(x+3h,y) - 3f(x+4h,y)}{12h} \tag{3}$$

$$f_{,y}(x,y) = \frac{f(x,y-2h) + 8f(x,y+h) - 8f(x,y-h) + f(x,y+2h)}{12h} \tag{4}$$

$$f_{,y}(x,y) = \frac{-3f(x,y-h) - 10f(x,y) + 18f(x,y+h) - 6f(x,y+2h) + f(x,y+3h)}{12h} \tag{5}$$

$$f_{,y}(x,y) = \frac{-25f(x,y) + 48f(x,y+h) - 36f(x,y+2h) + 16f(x,y+3h) - 3f(x,y+4h)}{12h} \tag{6}$$

$$f_{,y}(x,y) = \frac{f_{,x}(x,y-2h) + 8f_{,x}(x,y+h) - 8f_{,x}(x,y-h) + f_{,x}(x,y+2h)}{12h} \tag{7}$$

$$f_{,y}(x,y) = \frac{-3f_{,x}(x,y-h) - 10f_{,x}(x,y) + 18f_{,x}(x,y+h) - 6f_{,x}(x,y+2h) + f_{,x}(x,y+3h)}{12h} \tag{8}$$

$$f_{,y}(x,y) = \frac{-25f_{,x}(x,y) + 48f_{,x}(x,y+h) - 36f_{,x}(x,y+2h) + 16f_{,x}(x,y+3h) - 3f_{,x}(x,y+4h)}{12h} \tag{9}$$

(Faires & Burden, 2013)

To represent the spline, the coefficients matrices are stored in 2D array representing the position of their corresponding function. To evaluate the spline at a certain point (x, y), the right coefficients needs to be retrieved from the coefficient array. The x and y coordinates need to be normalized to the range $[0, 1]$ because the coefficients were computes on a unit square. The evaluation is therefore defined by:

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

The bi-cubic spline interpolation is used in the course editor to allow points on the map to be dragged up and down. Doing this changes the data points dynamically. It updates the required partial derivatives and interpolates the bi-cubic splines around the changed data points.

2.3 Bi-quintic spline interpolation

While bi-cubic spline allow smoothness along the curve, bi-quintic allows the smoothness of the first derivatives too. The coefficients of bi-quintic splines are computed with higher order partial derivatives. The second partial derivatives $f_{,x^2}$ and $f_{,y^2}$, the third partial derivatives $f_{,x^2y}$ and $f_{,xy^2}$ and the fourth partial derivative $f_{,x^2y^2}$ are all needed. The evaluation of the bi-quintic spline is defined by:

$$p(x,y) = \begin{bmatrix} 1 \\ x \\ x^2 \\ x^3 \\ x^4 \\ x^5 \end{bmatrix}^T \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{44} & a_{25} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{50} & a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix} \begin{bmatrix} 1 \\ y \\ y^2 \\ y^3 \\ y^4 \\ y^5 \end{bmatrix}$$

For this problem, the extra computation required for interpolation and the evaluation of the points on the terrain are not worth the barely notable visual change.

3 Engine

When the ball is hit it, is set into motion in a given direction and speed. In accordance with the user and system requirements in Appendix A.1 the motion of the ball is governed by two forces:

1. The force of gravity as determined by the sloping golf course;
2. The force of friction as determined by the terrain.

The role of the engine is to provide a simulation for this motion. The engine uses numerical methods for solving ordinary differential equations (ODEs) to approximate the change in velocity and position of the ball over time. This chapter describes the following chosen methods explored;

1. Euler Method- "a method that is easy to use but is not very precise unless the step size, the intervals for the projection of the solution, is very small." (Gerald, Wheatley,1915, p331)
2. Classical fourth order Runge-Kutta- "Presents methods that are based on more terms of a Taylor seriesTaylor series than the Euler methods and are thereby much more accurate. (Gerald,Wheatley,1915, p331)
3. Adams-Bashforth Multi-step Method - "Covers methods that are more efficient than the previous methods, which are called single step methods. They require a number of starting values in addition to the initial value. A Runge-Kutta method is frequently used to get these starting values." (Gerald,Wheatley,1915, p331)
4. Adams-Moulton Implicit Method - Implicit methods have higher accuracy than explicit methods of the same order.(Faires & Burden, 2013) A fourth order predictor-corrector method is implemented in the engine.

3.1 Physics of Putting

The position of the ball is described by its coordinates $p = (p_x, p_y)$ and velocity $v = (v_x, v_y)$. The equations of motion of the ball moving on a sloping course of height z = h(x,y) is then defined as follows:

Partial derivatives are required: $\frac{\partial_z}{\partial_x} = h_x(x, y)$; $\frac{\partial_z}{\partial_y} = h_y(x, y)$;

The gravitational force,G, due to the slope is given by:

$G = (-mgh_x(x,y), -mgh_y(x,y);$

where m is the mass of the ball, and $g = 9.81ms^{-2}$ the acceleration due to gravity.

The frictional force, H, is:

$H = -\mu mgv/||v||$ where $||v|| = \sqrt[2]{v_x^2 + v_y^2}$ The equations of motion that the engine solves is Newtons Law, a second order Differential equation:

$\frac{d^2p}{dt^2} = a = F/M$

Where the acceleration,a, is the rate of change of the velocity. The Velocity is also a derivative, the rate of change the position,p, of an object of mass, M, when it is acted on by force, F.(Gerald & Wheatley, 1989) $F = F(p, v) = G + H$.G is the gravitational force and H is the force due to friction of the moving ball.

3.2 Euler method

The engine updates the ball position and velocity after a single step size h using Euler method as follows:

$p(t + h) = p(t) + hv(t)$; $v(t + h) = v(t) + hf(p(t), v(t))/M$. Where the initial time, t and initial velocity are known. This equation is broken down into two components for the x and y coordinates:

$x(t + h) = x(t) + hv_x(t);(1)$

$y(t + h) = y(t) + hv_y(t);(2)$

$v_x(t + h) = v_x(t) + hF_x(x, y, v_x, v_y)/m;$ (3)

$v_y(t + h) = v_y(t) + hF_y(x, y, v_x, v_y)/m;$ (4)

In the first step, equations 3 and 4 use the initial velocity $v(t)$ of the ball inputted by the player or bot, the next iterations will then use the previous velocities for $v(t)$. $F(x, y, v_x, v_y)$ is a function for acceleration. Equations 1 and 2 will then use the new velocity calculated by equations 3 and 4 to update the balls position.

The global error for Euler method is $O(h)$. If h is small enough the error is smaller.

3.3 Classical fourth order Runge-Kutta method

The fourth order Runge-Kutta uses a weighted average of four estimates to obtain better accuracy. (Cheever & College, n.d.) Given the initial value problem of how to calculate the velocity and position with initial time, t, and initial velocity $w_iati = 0$ the following approximations are calculated:

$k_1 = hf(t_i, w_i);$
$k_2 = hf(t_i + \frac{1}{2}h, w_i + \frac{1}{2}k_1)$
$k_3 = hf(t_i + \frac{1}{2}h, w_i + \frac{1}{2}k_2);$
$k_4 = hf(t_i + h, w_i + k_3);$
$w_{i+1} = w_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$

The engine will first calculate the new velocity of the ball using Runge-Kutta, (RK Velocity). The function $f(t, w)$ will calculate the acceleration with t being the position $p = (p_x, p_y)$ and being velocity $v = (v_x, v_y)$ In k_1 the derivative is calculated at the initial point. k_2 and k_3 calculate the derivative at trial midpoints and k_4 at a trial endpoint. (Press, 2002). The engine will then use Runge-Kutta again to update the balls position, (RK Position). This time the derivative used in the function $f(t, w)$ is the velocity calculated by RK Velocity. The global error for Runge-Kutta is $O(h^4)$.

3.4 Adams-Bashforth method

The following four-stage Adams-Bashforth method is implemented to calculate the new velocity of the ball w_{i+1}:

$w_{i+1} = w_i + 1/24 * h(55f(t_i, w_i) - 59f(t_{i-1}, w_{i-1}) + 37f(t_{i-2}, w_{i-2}) - 9f(t_{i-3}, w_{i-3}))$

The first step of this multi-step method w_{i+1} requires velocities from the previous steps, w_i for $i < 0$. With w_0 being the initial velocity. The Adams-Bashford solver uses a Runge-Kutta method of the fourth order to obtain these values, this is called bootstrapping. The method chosen to bootstrap must be the same order as the multi-step method or higher. This motivated the choice to use Runge-Kutta 4. Therefore w_i, w_{i-1}, w_{i-2} are velocities found using bootstrapping. $f(t, w)$ is the function that will calculate the balls acceleration, with t being the previous ball positions and w the previous velocities. h is the step-size.

3.5 Adams-Moulton method

Adams-Moulton is a multi-step implicit method. The three-stage Adams-Moulton method is used by the engine using a predictor-corrector approach to calculate the velocity of the ball. The explicit Adams-Bashforth method is used as the predictor to estimate w_{i+1} and Adams-Moulton is the corrector. The formula used is as follows:

$w_{i+1} = w_i + \frac{1}{24}h(9f(t_{i+1}, w_{i+1}) + 19f(t_i, w_i) - 5f(t_{i-1}, w_{i-1}) + f(t_{i-2}, wi - 2))$

4 Bot

4.1 Simulating Bot

The Bots or AI's that are implemented are based on heuristics, while using simulations to predict the path the ball would take on the course. The simulations create a new physics engine object and a new ball to prevent the simulations from showing up in the game UI. For the normal courses, two bots were chosen for further testing. These bots use the results of a simulation of a shot to alter their hit's direction and intensity. These alterations are recursively applied until the bot manages to score a hole-in-one. The differences between the two bots are the heuristics they use and the changes made using those heuristics.

The first of which uses the position of ball after it has stopped as a heuristic. The ball can stop in one of the four Cartesian quadrants, centred on the hole's position and split along the x and y axis. Once the simulating bot takes a shot, the difference between the x and y coordinates between the ending position of the ball and the hole are calculated. After which, the sign that the resulting x and y values have are used to determine how the velocity for the next shot is scaled . If the x or y value is positive, the corresponding x or y component of the velocity is scaled by 0.95, and the components are scaled by 1.05 if the values are negative.

The second bot was made as an adaptation of the first bot. Since the first bot only has 4 possible steps to alter it's hit after each simulation it is somewhat slow and inaccurate. The alterations of the second bot aren't based on the position of where the ball has stopped, but instead uses the position on the ball's closest approach to the hole. This eliminates a problem the first bot was having on sloped courses. Since on a sloped course the ball will always end up at the bottom. The heuristic was changed to not use only 4 quadrants, but to use a variable input instead. The difference between the position of the ball's closest approach and the position of the hole is then taken as a vector, scaled by a certain scalar, and added back to the initial hit vector. This means if the ball passes the hole closely on the right, the hit vector is adjusted slightly to the left. If the ball doesn't reach the hole, the hit vector's length is increased.

4.1.1 Testing

Tests on the scalar value of the second bot were done on 5 different courses. It was tested for 7 different values, from 0.1 up to 0.5, 0.7 and 1.0. After those tests, an 8th value of 0.25 was tested as well. The results show the amount of simulations it took to hit a hole-in-one.

Testing results

Course	0.1	0.2	0.25	0.3	0.4	0.5	0.7	1.0
Plane	1	1	1	1	1	1	1	1
Slope	18	8	8	7	7	4	4	4
Cosine	5	3	3	3	6	7	7	7
Circular Cosine	4	3	2	2	2	2	2	2

Tests were done on both of the simulating bots to see how the amount of simulations and shots taken to reach the hole affected by different noise intensities. There are ten different levels intensities that start with one being the weakest, and five being the strongest. Both bots were tested on the same two maps while incrementally applying levels of noise starting from one and ending at five.

4.2 A*

The bot that was created to deal with complex or maze-like courses uses the A* algorithm. This A* algorithm is an improved variation on the Dijkstra algorithm for graph search and pathfinding. Finding the best path using this algorithm is done by an iterative approach of checking out all the nodes on a path. It begins at a start node and then works its way towards the goal node by visiting surrounding nodes.

A* is an search algorithm that requires domain knowledge and it can be classified as a best-first search algorithm. This means at each stage of the algorithms iteration it chooses which node to expand according to the accumulated cost so far and a heuristic. The algorithm minimizes the following equation: $f(n) = g(n) + h(n)$, where f is the cost to get to the goal, g is the cost to get form the start to the node that is currently checked and h is a heuristic.

For every terrain in our game the heuristic should be adjusted accordingly because it is problem specific, i.e. the heuristic for a plane would be the linear distance between two points. At each iteration the heuristic provides a value that describes the difference between the currently checked node and the goal node. To find the actual shortest path the chosen heuristic can never overestimate the actual cost of getting from the current node to the nearest node.

A priority queue is used to sort the nodes in the openSetopenSet, which contains the nodes that have been discovered but not yet have been visited, according to their respective $f(n)$ value. The node with the lowest $f(n)$ value will be the first element in the queue. To prevent the algorithm from visiting nodes more than once, it adds the visited nodes to a closedSetclosedSet Before adding a new node to the openset, the closedSet is checked whether or not it already contains that node.

At each step of the algorithm i removes the first element of the openSet while it still contains values. After this the $f(n)$ and the $g(n)$ values are updated and then the surrounding nodes are added to the priority queue if they are available, meaning that the location of the node is not obstructed by some obstacle. This process continues until the currently checked node the goal node. Once a goal node is found the algorithm traces its path back and creates a string containing the names of the nodes that lead to the goal in the path the algorithm found.

4.2.1 Motivations

The algorithm of choice for the complex bot was A* because by adjusting its heuristic function you can get different results. When your heuristic is costly it will find a route to the goal within a small amount of time. However, a more precise solution could be achieved by decreasing the cost of the heuristic function. To get a reasonably fast algorithm that also is precise, an equilibrium needs to be reached between these functions.

4.2.2 Testing

Simulations are used to acquire the velocity to hit a hole-in-one, so tests are continuously running.

4.3 Alternative approaches

Various approaches were explored, however, only few seemed to be effective or even possible. One of the examined approaches was a genetic algorithm, which after some research appeared to be impractical to implement due to the high complexity of the genetic algorithm. To be able to implement a genetic algorithm, there must be a way to do mutations and crossovers on certain populations. Since implementing a method to do mutations and crossover would require extra, and unnecessary, computations to convert the velocity to a population of strings that represent those vectors. Apart form this, the time it would require to train the genetic algorithm on each course would take a lot of time which is unreasonable for the user. Another option that was explored was rigid body dynamics. Rigid body dynamics is a common techniques in modern physics engines.No implementation of this technique was found that would take the function based courses into account.

4.3.1 Theories

The genetic algorithm starts to operate by taking variables of interest and converting those into a string and assigned a fitness value. Strings are then mutated with other strings, and based on how the fitness is calculated, the new mutated string is assigned a new fitness value. The mutation of strings will occur until the fitness value of a string matches the needs to complete the task at hand. (Capalbo, 2017)

4.3.2 Motivation

Motivations for the genetic algorithm in the context of the putting AI were too few to warrant implementation. However, there are incentives with using the algorithm. One example is when the ball is hit it is given a velocity vector and a direction. The velocity and direction could be mutated by a genetic algorithm until the mutated values would cause either a hole-in-one, or a well placed shot which will eventually lead the ball into the hole. The paramount problem with this approach is that mutations are random for the most part, making the odds that an appropriate mutation will be found in a timely matter very low.

5 Noise

Noise is implemented in the golf simulation; introducing a small random error in the initial position and velocity of the ball. This noise is implemented in the form of wind. The wind is a constant force added to the acceleration computation:

$F_D = \frac{1}{2}pv^2C_DA$ where p is the density of the wind $= 1.225kg/m^3$, (Batchelor, 2013)

v is the speed of the ball relative to the wind,

A is the cross sectional area of the ball,

C_D is the drag coefficient of the ball = 0.47

However the wind direction is rotated by a random normally distributed value,x, with mean 0 and standard deviation 1. A rotation matrix is used for this:

$$\begin{bmatrix} cos(x) & -sin(x) \\ sin(x) & cos(x) \end{bmatrix} \tag{10}$$

6 Collisions

6.1 Collision Detection

Collisions between the golf ball and obstacles need to be taken into account in order to get a realistic simulation. This is achieved by using the radius of the ball, as well as breaking down and representing the obstacle's four corners as vectors on the height-field. Edges are then created by taking the difference between vectors and labelled as AB. The next vector, labelled as AC, that is calculated is the difference between the centre of the ball and the nearest corner point of an obstacle. In order to get the point on AB closest to the ball the cross product is performed between the vectors AB and AC, which is then normalized. This newly calculated vector is labelled CP. Finally, a check is done to see if the radius of the ball is overlapping with the point CP. This is accomplished by taking the difference between the centre of the ball and the radius, along with with the centre of the ball and CP. If the difference between the centre of the ball and the point CP is greater then the radius of the ball then it is concluded that there is no collision.

6.2 Collision Reaction

Once the collision detection algorithm has found a collision between the ball and an obstacle, there must be a change in the balls direction and velocity to represent this. To show this, the normal vector is calculated where the ball hit the obstacle and added to the velocity of the ball. The normal vector is already calculated during the collision detection and was labelled as vector CP. After the CP vector is added to the balls velocity it will bounce off the obstacle.

7 User Interface and Graphics

7.1 User Interface (UI)

7.1.1 UI Course Editors

The User Interface (UI) has an option when at the beginning of the game named "Course Editors" that allows the user to create/edit courses and terrains. This option leads to four new options that allow the user to choose how to create or edit a course/terrain. The first option is labelled "Create terrain from function." Choosing this opens up a new window where a function can be placed in order to create a terrain that will mold to the shape of that function. Other edits that can be done include where the ball is located from the start, where the hole is located from the start, the start location of the hole, and finally the size of the ball and hole.

A second option is labelled "Edit a terrain created from function." While similar to the previously described option there is one difference. Now there are preloaded equations that a user can pick from to edit, instead of coming up with a function themselves.

The third option is labelled "Create terrain from splines." This option loads up a flat plane where you can choose where to create a hill or a sinkhole using splines to calculate what the equations will be. Once the user is done creating the course it is saved for later use.

Lastly there is an option labelled "Create a course." This option brings up a new window that request a user to select a terrain and assign that terrain a number. When the user plays the game all of the terrains will be played in the order they were assigned.

7.1.2 UI Start Option

When opening the game, the user has a "Start" option. This button opens a new menu, where the player can choose between one of the predefined terrains to play on. The game can also be played on a course, which is made of multiple terrains. This is done by using a toggle located on the right of the screen.

A click on the "Next" button will open a new window, where 3 game modes are offered; "Single Player", "Multi Player" and "Bot". This will allow the user to play alone, with a friend, or to let an AI play the game. A click on one of those 3 buttons will start the game.

The game can then be paused by pressing the "Escape" button. This will also bring up a new menu, with 5 different options, namely "Settings", which will be explained in the next section, "Main Menu", "Resume", "Give Up", and "Restart". These options respectively allow the user to go back to the main menu, resume the game, give up and restart the same game.

7.1.3 UI Settings Option

The "Settings" option can be accessed from the starting menu as well as from the "Pause" menu. As implied in its name, this button will open the game's settings, where some aspects of the game can be modified.

The first option is to adjust the game's volume using a slider. The other modifiable aspects of the game are the Solver, the input, the water and the wind. Five different solvers can be chosen to play the game; "Euler", "Runge-Kutta", "Adams-Bashforth", "Adams-Moulton" and "Bogacki–Shampine". The game input can be set to "arrow" or "manual", and water and wind can be turned on and off.

7.2 Game graphics

To represent what the program is doing, a 3D visualization was implemented. Visualization is achieved through the use of the libGDX libGDXlibrary. The world which includes the course or terrain is made up out of chunksChunks. These chunks are represented by the Height fieldsHeightFields, which evenly spaced grid of values, where each value defines the height on that position of the grid, so forming a 3D shape.(Unknown, 2015)

8 Software design & techniques

The game's systems are managed from a general class named "Statemanager." Statemanager keeps track of all the states and displays the one that is currently active. It keeps track of these states with a stack of states. If a new state is required due to user input, the state gets created and pushed onto the stack. There are a few kind of states that can be pushed to the Statemanager there is:

- State: General state.
- State3D: Graphical state.
- MenuState: Menu state.
- SubMenu: Sub-menu state.

The above mentioned classes make up the general framework of what states can be represented. Depending on what needs to be shown to the user, one of these states are used to create a new and more specific class. For example the game class in this project, which controls every operation when the game is currently running, extends the State3D class.

9 Results

Results for the bot simulation tests for (figure 2) show that on a flat plane, that there wasn't an increase of the amount of simulations in either AI until the noise intensity was raised to level five. However both bots were still able to make a whole in one for each intensity. The next test was done on a plane with a spline curve in-between the ball and the hole (figure 3). The data shows that the simulating bot is not able make it into the hole once wind noise is applied. AI is able to complete the courses until the noise reaches level five. There is also a noticeable increase in the average number of shots in proportion in the increase in noise level, which is to be expected.

10 Conclusion

In conclusion, the golf simulations terrain is formed with the use of splines. To calculate the updated position of the ball solvers like Euler and Runga-Katta were applied. The golf simulation can either be operated by the user or an AI. The simulating bot showed to work in pristine conditions, however it could not complete courses once noised was introduced. The AI is able complete courses on all levels of noise except for the highest level.

A Appendix

A.1 Requirement Specification

Here are the requirement specifications for this project. We have User Requirements, that are linked to System Requirements. These latter are then sorted in two categories, functional (F) and non-functional (NF) requirements. The first category is about services that the system should provide, what it should and should not do. The second category includes constraints on the service, that often apply to the system as a whole.

1. The motion of the ball should be governed by gravity and friction and the ball should not leave the surface.
    (a) The system should make use of a physics engine that continuously calculates the forces acting on the ball with the use of physics equations of gravity and friction and equations of motion. (F)
    (b) The height of the ball on the terrain will always be defined by the entered function of the surface. (F)
2. The user must be able to input a custom course (specify height function, friction coefficient, the start and the goals positions of the ball, the radius of the target and the maximum velocity).
    (a) The function should be parsed by a function parser. (F)
    (b) The system should be able to evaluate the function for different values of x and y. (F)
    (c) When the height of the function is below 0, this will represent water (F)
    (d) The system should allow the user to decide at the start of the game what the start and goal coordinates of the ball are. (F)
3. Add an artificial intelligence player capable of playing on the terrain
    (a) The AI will try to score a hole-in-one where possible.(F
    (b) The AI will be able to handle complex maze-like courses. (F)
    (c) The bot will have full knowledge of the geometry and slope of the terrain. (F)

   (d) A small random error in the initial position and velocity of the ball should be introduced to investigate how it affects the performance of the AI. (F)

4. Graphics - user can see the ball, the ball and the course interact

   (a) Graphics are in 3D.(F)
   (b) Use a graphic library such as LibGDX. (NF)
   (c) Use java 1.8 such that we can use the most recent version of LibGDX. (NF)

5. There will exist a mode where a player makes choice of initial velocity of the ball (speed and direction)

   (a) If the ball is not moving, the input window appears where he can input the two values mentioned above. (F)

6. Mode where initial velocity of the ball is read from a file

   (a) Implement a .txt file reader that reads the speed and angle. (F)

7. Physics engine and math solver must be original (i.e. don't use a pre-existing library)

   (a) The physics engine will make use of differential equations in order to find values needed to determine the movement of the ball. (F)

8. Collision detection when the ball hits water, sand etc.

   (a) The collision detection should detect when the ball hits an obstacle (like a tree) in its path and adjust the movement of the ball accordingly (for example place the ball at the coordinates in front of where it hit the obstacle). (F)
   (b) The collision detection should reduce the movement of the ball when the ball is on rougher terrain like sand (terrain with greater friction). (F)

9. If the ball ends up in the water, the user should either be able to replay the shot from the point around the water or from the position the shot that landed in the water was taken.

(a) The system should be able to detect when the ball hits the water through use of the height of the function (when the height is less than 0, the ball is in the water). (F)

(b) When the ball hits the water the user should be given the choice of replaying the shot from the previous position or from a position around the water. (F)

(c) When a choice is made by the user, the ball should be placed at the coordinates corresponding to the choice of the player. (F)

10. Allow for input and output of course functions to and from a file

    (a) The system must implement a file reader/writer (F).

    (b) The system should have the option of writing the function a course created by the user to a file. (F)

11. Scores of the players should be saved to a file

    (a) Save all the scores in a .txt file that can be read by the player by pressing a button on the user interface. (F)

    (b) The system should be the only agent allowed to modify the text file (for security). (NF)

12. Splines should be used to describe the height of the course

    (a) The user should be able to input points that will be used as the knots for the spline function. (F)

13. Add obstacles, such as sand and trees

    (a) Sand areas must have increased friction (F)

    (b) Trees will cause the ball to hit off it in the opposite direction (F)

14. The course is completed when the ball reaches the goal

    (a) The user should be given a visual representation of the target area of the ball (F)

    (b) The goal is defined by a coordinate and a radius that allow for the user to land the ball in an area (F)

   (c) Once the ball hits the goal coordinates, the user is notified with a message on the screen that the ball has reached the goal area, and allowing another shot is disabled (F)

15. Write documentation so that other programmers will be able to understand the code

    (a) Write javadoc documentation (NF)

16. The movement of the ball should be smooth

    (a) The timestep of calculations should be fast enough to allow smooth movement of the ball (NF)

17. The program should be simple enough for children to use

    (a) The graphical user interface will make use of sliders to pick the direction of the shot of the ball, and its velocity, and a button to shoot the ball in order to facilitate use for users of all ages (NF)

References

Batchelor, G. K. (2013). *An introduction to fluid dynamics.* Cambridge University Press.

Capalbo, M. (2017, Dec). *Lecture 8 development and evolution.* Maastricht University.

Cheever, E., & College, S. (n.d.). *Fourth order runge-kutta.* Retrieved from `http://lpsa.swarthmore.edu/NumInt/NumIntFourth.html`

Faires, J. D., & Burden, R. L. (2013). *Numerical methods* (4th ed.). Brooks/Cole, Cengage Learning.

Gerald, C. F., & Wheatley, P. O. (1989). *Applied numerical analysis. gerald* (7th ed.). Addison-Wesley.

Press, W. H. (2002). *Numerical recipes in c* (2nd ed.). Cambridge Univ. Press.

Unknown. (2015, Dec). *Libgdx heightfield.* Libgdx.