Prototyping Walking Robots Using Genetic Algorithms: Design

Kevin Beck

Joseph Borsodi

Ryan Romanosky

Kristy Schworn

California University of Pennsylvania

Design Document

CSC 490: Software Engineering

6 December 2017

Instructor Comments and Evaluations

Table of Contents

Abstract

This paper explores the analysis of the specifications document and further details the design of the various components that comprise the project. As previously stated in both the requirements and specifications document, is the prototyping of walking robots using genetic algorithms. These algorithms provide unintuitive solutions to problems using iterative processes and rapid revision to develop the best solution. The analysis of the software outlined in the prior documents will provide a foundation on which this design document is created. This document specifically outlines the various components of the projects and how they interact within the project's execution. This document also explores the methods in which the software will be used and the flow of information between each portion of the developed software.

Keywords: Genetic Algorithm

Description of Document

Purpose and Use of this Document

The purpose of this design document is to give an accurate description of how the Prototyping Walking Robots with Genetic Algorithms project works. The design document builds upon the previous document—the Specification document—and will show the overall outline of the project. A more detailed, step-by-step overview to help the audience understand the methods and functions used in this project is, in addition, described. The diagrams and charts laid out in this document will allow for a more thorough comprehension of this project and its various classes. This document will be used during the implementation workflows of Prototyping Walking Robots with Genetic Algorithms.

Ties to Specification Document

This design document is an expansion and extension of the specification document. Topics and key points discussed in the specification document will make up the main points of the design document. New diagrams and charts will be created to augment the audience's understanding of these topics. As the design document can be thought of as an extension of the specification document, diagrams used in the specification document will be reworked to fit the constraints of the design document. Details involving the connections between each functional class and flow of information between them will also be discussed, and, if necessary, expanded upon.

Intended Audience

The intended audience for this document comprises software engineers and programmers. Specifically, those who wish to use our program to test and review our design and the feasibility of said design. As complex topics such as a class structure, system modules, and CASE diagrams will be touched upon and discussed, the intended audience will not include clients or users—with the exception that they might also be software engineers and programmers. As with the specification document, a background or at least basic experience in computer science will help the reader more fully comprehend the topics at hand.

Project Block Diagram

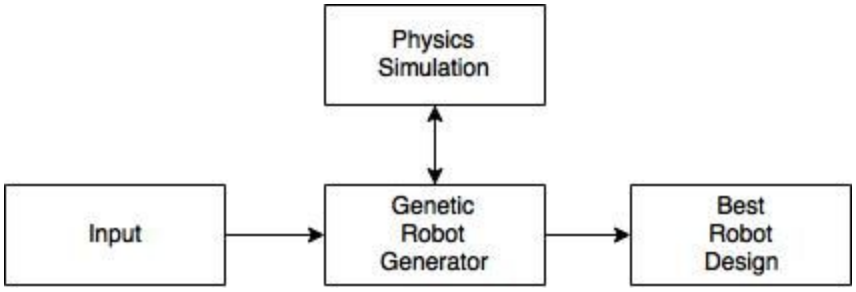

Figure 1. Block Diagram of System

System Overview

The software generated from this design document will use multiple software only subsystems to produce randomly generated robots that can traverse a certain terrain. This is a software only system that has no hardware association. The application would be installed at the client's location on their hardware which will allow them to run as many tests as they see fit. A user will be able to enter certain constraints through the use of a user friendly interface and then

begin to run simulations in a 3D engine. Based on the input from the user the 3D engine will generate populations of robots using genetic algorithms as well as a virtual terrain for them to traverse on. During testing a fitness score will be given to each bot. At the end of the simulation the final result, the robot with the highest fitness, will be returned to the user.

Design Analysis

Throughout the development of the requirements, specifications and design documents many different workflows and data flows were created and iterated upon. The basis of the project has maintained its focus on the original plan, the creation of software capable of testing and iterating upon robot designs to result in the best possible individual bot for a specified terrain. Over the course of the requirements phase various elements were refitted, redesigned, or entirely removed, however, through the specifications and design phases, the focus has remained entirely on the proper positioning of the classes and modules to effectively achieve the goal at hand. The following outlines some of the changes made through the design analysis and the differences of the current project from what was previously explained in the specifications document.

While our classes maintain their names and most of their prior functionality, the functions related to the genes of the robots have been assigned to the algorithm class rather than the robot class, leaving the robot class to have clearer focus on the actual construction and control of the actual robot entities. The algorithm, being responsible for the generation of the populations at large, has now absorbed the relatively cohesive task of generating the individual robots as well.

This enables better encapsulation of the data for the robot class, while keeping the goals of each class more distinct and clearly stated.

Another alteration comes from the combination of the settings data to their controller classes. This creates larger classes, but the functions of each individual class are much more focused on their domains. While this increased the complexity of the classes by a small margin, issues found during the design analysis. Often times, when a class had a separate settings class that stored the information, there was a lack of understanding of the function of the settings classes. By combining the Settings Data Class objects into the Controller objects, the design is much less obscure, and with thorough documentation will maintain its readability despite the increase in class responsibility.

The creation of the subfunctions in the Algorithms class which handle the selection, crossover, and mutation functions of the population manipulation helped tremendously with potential additions and versatility moving forward to implementation and beyond. The ability to isolate the crossover and selection functions specifically allow the application to double select or double crossover. These functions enable the design team the option of more nuanced manipulation of the population through different techniques.

The elimination of the various components and settings, and the revision of what could and could not be constrained by the user from the interface was also altered and changed to better reflect the goals of the project. There were also alterations to the various features available to the user for customization in the robot menu. This was the best optimize the Settings to Effect ratio that was beginning to increase. The user had many options to change, but those

changes were feared to be less than noticeable and therefore were tedium for which the user had to endure. By the elimination of these settings, the design team streamlined the process for the software's use.

Cohesion and Coupling

Throughout the development of the project, many iterations of various classes and structures were considered in the interest of cohesion and coupling. The goal being to minimize the coupling of the various components, while maximizing the cohesion of what those components were to accomplish. With this goal in mind, the project resulted in the classes detailed in the following pages of this document. This specific project struggled in the area of coupling, as many of the functioning parts of the project required information not contained within specific classes. Through multiple revisions, the design took advantage of the class structures in order to maximize the cohesion as much as possible, while conceding that certain portions of the algorithm class will delve into the robot's domain to some degree. Through each iteration there were benefits and costs associated with each structure of functionality and encapsulation of data, and the current revision is thought to be the most thorough and robust for the application goals.

The Terrain Class manages high cohesion and low coupling. The class' sole responsibility is the domain of the terrain within the program and simulation. The functions maintain their independence from other classes, and the data stored within the class are clear and apparent to the purpose of the Terrain Class. This class specifically manages a strong level of object oriented design.

The Robot Class also has low coupling and high cohesion. The purpose of the class is to manage the settings related to the robot class as well as create robots within the simulation. This class maintains control of the components of which one would expect. While the robot class is capable of developing a robot from a gene, it is not the class for which the genes are developed. This may lead to some confusion, but the best possible interaction was sought and the current design outlined in this document was found to be most intuitive of the explored designs. While the data and functions within the class are fitting and well encapsulated, not all classes are completely decoupled.

The Algorithm Class is the class that struggles with its cohesion and coupling. The function related to generating a gene for a robot seems as though it could be better placed within the robot class, but the algorithm class itself is best positioned for the further manipulation of the genetic code. For this reason, the ability to originally generate the sequence for which robots will be generated resides in the responsibility of the algorithm class. This allows the class to increase its cohesion slightly by being the class solely responsible for the generation and manipulation of the genetic information. However, this also increases its coupling, as it requires information from the robot class before it is able to create genes for use in the application.

The Simulation Class exists within this program solely as an aggregate of other functions of classes, and as an overview of the simulation testing process. The cohesion of the simulation class is clear, test and score the robot designs. The coupling of the class within the confines of the program is high. As a class it has a consistent identity, however the class requires the specific entities present in the project for functioning. Beyond the scope of the overlapping requirements,

the simulation class manages to maintain its identity as the controlling force behind the testing of the robots created by the robot class, on the terrain created by the terrain class. It lends itself well as a utility, managing various components of the project to achieve the goal of creating the best design.

Data Flow Diagram

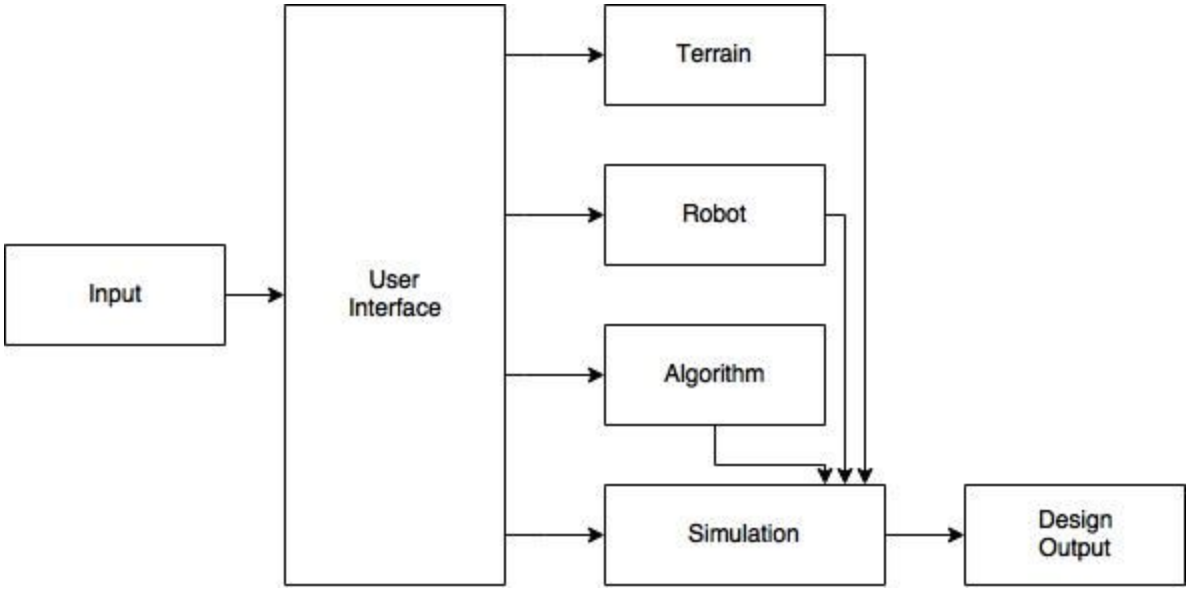

Figure 2. Enhanced View of Input (Data Flow)

Figure 2 shows an enhanced view of the input subsystem. As a result of this software relying heavily upon user input, a user interface will be created within the 3D engine to increase usability. All input from the user will come directly through the user interface enabling the ability to quickly get a simulation up and running in a short amount of time. Input will flow from the user interface directly into its intended controller class. The settings controller classes will be responsible for making sure that only valid data is being entered and will inform the user of any

invalid parameters. The simulation controller classes will interact directly with the settings controller classes by requesting the stored data for use within the 3D simulation. After all simulations are complete the final result will be passed along for output to the user.

Broad Description of Controller Classes

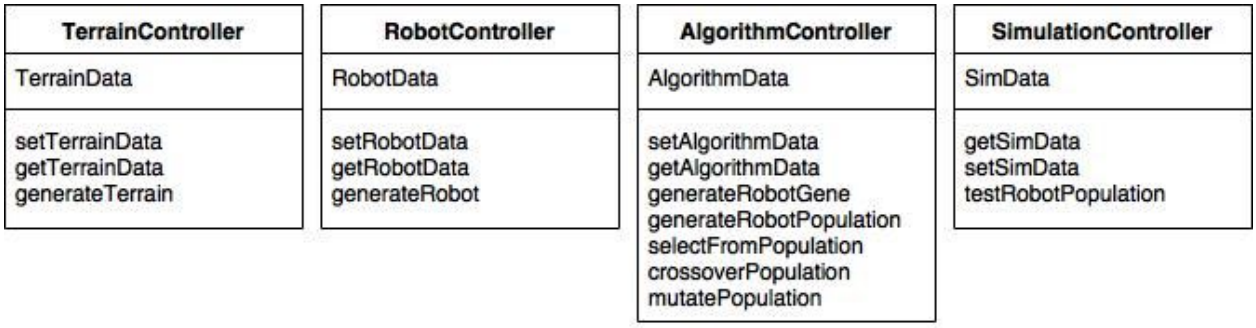

Figure 3. Controller Classes

Terrain Controller

The terrain controller consists of these data values: Structure type, Texture type, Texture Size, Texture Density, Obstacle Type, Obstacle Size, Obstacle Density, and Gravity. Along with each of these pieces of data is a setter function which sets the data to a specific value and a getter function that returns the value stored in the data types. Within the terrain controller class is the generator terrain function which is used to create the terrain on which the robots will be tested. The Generate Terrain function will be available publicly and accessed by the simulation class to initialize the testing phase.

The structure type will consist of an integer representing the physical layout of the terrain. These terrain types will consist of the following: flat plane, inclined plane, and staircase. Texture type will reflect the actual surfaces of the terrain. This value will change the level of

friction on the surfaces of the terrain. The texture size will change the size of each individual piece of texture. Texture density will determine how many pieces of terrain will appear. Obstacles will be movable objects on the terrain that will affect the movement of the robot. The obstacle type will include various shapes. Obstacle size will allow for the sizes of each obstacle to vary. The obstacle density will be able to determine the amount of obstacles in the robots way. Gravity consists of a force vector applied to the robots. The vectors magnitude may be altered to reflect a change in the strength of the force while the direction of the vector may be altered to reflect other external forces, such as wind.

Robot Controller

Each of the following factors in the robot controller data has both a minimum and maximum component. The components are: weight, length, segments, legs, and leg force. Weight is the total weight of the robot, with all its features. The length of the robot is the distance of the robot from the first segment to the end of the last segment. Segments are the robots body parts. The legs are the locomotion component attached to the robot's segments. The leg force data member determines the force applied at each joint in the associated leg.

The user will set a specific carrying weight and drag weight, if required. This may be applicable if the robot needs to carry equipment or other materials in the simulation. Similarly, the drag weight represents a weight the robot is to carry on a sledge or cart-type attachment. While these will not always be included in the simulation, the option to add different restrictions to the robot allows for more diverse applications.

The data members in the Robot Controller class will be accessed publicly by the getter functions while the variables will be set through the setter functions. Also included in the Robot Controller class is the generateRobot function. This function is passed a string which represents the genetic code for each individual robot. It parses the string into each component that is implemented into the construction of the robot. This function will be accessed by the simulation function to create the robots during the testing phase.

Algorithm Controller

The Algorithm Controller class will handle the functions related to the processing of the genetic component of the software. The data stored within the Algorithm class will be fitness type, selection type, crossover type, and mutation type. The data is accessible through getters and open to modification through setter functions. The fitness type will be the data used to determine the rankings of the individual robots. The types of fitness will be velocity and distance. This allows the user to select the factor they determine to be most relevant to their testing environment.

The selection type is the methodology used to select the members of a population whose genetic traits are passed along. The types of selections will be identified as gradient selection, where each member of a population has a weighted chance to be selected, and top half selection, where only the top half of the population passes on its traits.

The crossover type will be the method for which the genes are shared between members of a population. The types include paired, randomized, and weighted. Paired will take the genes from one robot, and divide half the genetic code from one and half of the genetic code from the

other, combining them to create a new robot gene. The remaining gene portions are combined to create another robot. Randomized crossover will take a feature of one robot, and randomly assign the feature to another robot. This will enable an individual to obtain features from multiple different robots within a single iteration of the simulation. Weighted crossover takes preference to the individuals who perform better and merges their genetic features into poorer performing robots. This uses an assumption that the genes of better bots are preferred in the population as a whole.

The mutation type will contain the information for how the individuals in the population will be randomly altered within each generation. The types of mutation will be as follows: randomized, favored, micro, macro. Randomized mutation means that each feature of a robot has an equal chance to be changed by a random amount relative to the value of that specific feature. The favored mutation type will favor a particular feature of the robot to be more susceptible to mutations. Micro mutations will favor smaller adjustments as opposed to the normal range of mutation. This will allow a slower movement in the robot designs over generations. Opposite from micro, macro mutations will enable the large scale mutations inside an individual. This will be favorable in the cases where local maxima are restricting the development of better performing robots.

The algorithm's functions utilize information contained in other classes to provide the simulation class with all relevant information for the processing and analysis of each robot's design on the specified terrain. The function generateRobotGene creates a genetic code based on the constraints of the robot class and the methodology specified in the algorithm class. The gene

represents a single entity of a population. The generateRobotPopulation returns a list of genes that represent an entire population. The first iteration of the generateRobotPopulation will return a randomized list of genes. The selectFromPopulation function will be used to select the individuals from the population that are used to generate the genetic composition of the next population. The crossoverPopulation function will be used to combine different robots' genetic code to create the next population set. The result of this function will be a complete generation of the population. Finally the mutatePopulation function is called which makes various changes to the genetic code unrelated to the prior generation. This mutation results in a complete and new generation.

Simulation Controller

The simulation class is the class that maintains and evaluates the simulation environment and returns the results of each robots design. This simulation class contains data pertaining to the view of the simulation. This data is altered through a set function and retrieved using a get function. Also the simulation class contains a function called testRobotPopulation. This function iterates through the population of the robots and evaluates each one individually. After calculating each designs' fitness, it returns the population with the corresponding score for each individual robot.

Detailed Class Descriptions

Terrain Class

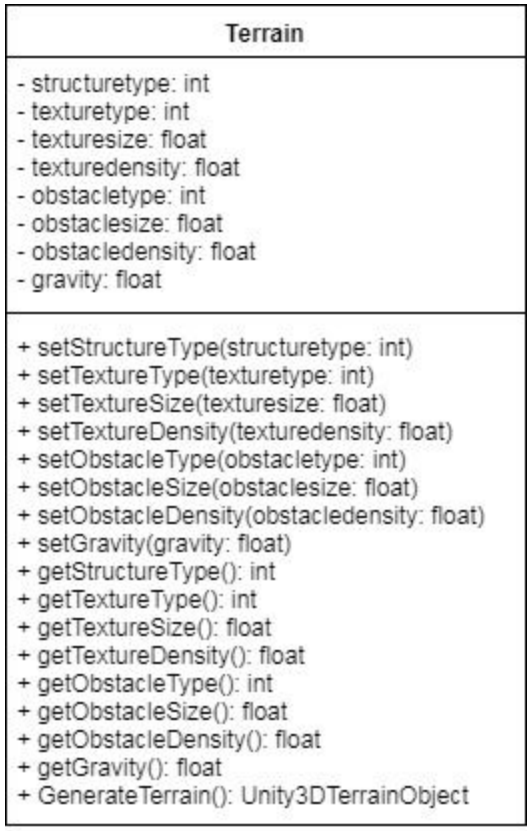

Figure 4. Terrain Class Diagram

Terrain Class Data Members

Structuretype: An integer value that represents the type and angle of the terrain used in the simulation. Flat plane, inclined plane, and staircase are all current values. More terrain types can easily be enumerated and added.

Texturetype: The texture type is an integer value that represents the physical surface of the terrain and the friction applied by each surface type. This will help delineate between smooth low-friction surfaces such as marble and rougher, higher friction surfaces such as sand.

Texturesize: A float value that represents the size of each piece of texture. This will allow for a sliding scale, moving from fine-grained textures such as siliceous sand to rough-grained textures such as gravel.

Texturedensity: A float value that represents the density of the terrain. This data member can allow for terrain such as sand peppered on top of another surface, to a full bed of sand.

Obstacletype: This data member represents the type of obstacle encountered by the robot. Similar to the terraintype member, each integer value represents a different type of obstacle. More types can be enumerated and included if and when necessary.

Obstaclesize: The obstaclesize data member is the obstacle analogue to texturesize. Like the texturesize data member, it is a float value that represents the size of the obstacles encountered by the robots. This can range from small obstacles such as stones to larger, boulder-sized obstacles.

Obstacledensity: The obstacledensity data member is a float value that represents the number of obstacles encountered by the robot as it makes its way forward. This can allow for the creation of obstacles only occasionally encountered to extreme obstacle densities such as those found on mountainsides.

Gravity: A float value in vector form representing a sum of the forces encountered by the robot. A vector was chosen to include other forces. This allows the inclusion of other forces such

as wind, if necessary. While only small changes of gravity are encountered by terrestrial elevation changes, the possibility of testing robots for extra-terrestrial terrains merits the inclusion of a changeable value for gravity.

Terrain Class Functions

The generateTerrainFunction will utilize the various data components within the Terrain class to instantiate a version of the terrain within the Unity3D engine. The function will first deploy the surface upon which the terrain is to be developed, this consists of either a flat/inclined plane or a staircase like pattern. After generating and instantiating this object within the physics simulation, the function will take the terrainTextureType and create a randomized pattern throughout the terrain surface. The texture objects added to the surface will use the specified texture size and the population loop for creating the texture will iterate for the specified texture density. After the generateTerrainFunction creates the texture, it will then use a similar process to create and instantiate the objects that are loose objects that may impede the locomotion of the robots. This process will similarly use the specified obstacle size and density factors to create the obstacles on all vertical surfaces of the terrain. Upon the completion of this process, the generate terrain function will update the physics simulation with the appropriate gravity as declared in the gravity data within the class. This will ensure the simulation is completed at the gravity specified by the user in the settings.

Robot Class

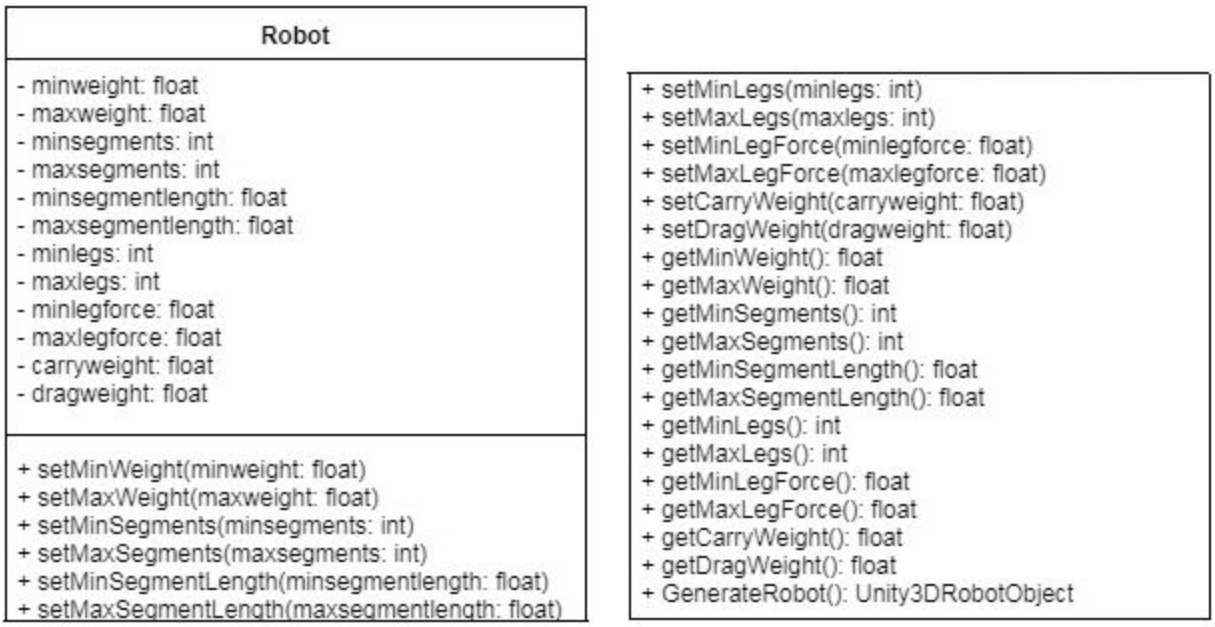

Figure 5. Robot Class Diagram

Robot Class Data Members

Minweight: A float value that represents the minimum allowed weight for the robot.

Maxweight: A float value that represents the maximum allowed weight for the robot.

Minsegments: An integer value that represents the minimum number of segments—or body parts—allowed for each robot.

Maxsegments: An integer value that represents the maximum number of segments allowed for each robot.

Minsegmentlength: This data member is a float value that represents the length of each segment of the robot.

Maxsegmentlength: This float value represents the maximum length allowed for each segment of the robot.

Minlegs: This robot class data member is an integer value that represents the minimum number of legs allowed for the robot.

Maxlegs: An integer value that represents the maximum allowed legs for the robot.

Minlegforce: This data member is a float value that represents the minimum force applied at each joint of the legs of the robot.

Maxlegforce: A float value that represents the maximum force applied at each joint of the legs of the robot.

Carryweight: A float value that represents an amount of weight that the robot will need to carry. This may not be necessary, but is useful to include in the case that a client wishes to test a caravan-type robot or, expanding upon the extra-terrestrial terrain scenario, scientific equipment.

Dragweight: A float value that is used to represent the weight that the robot would carry on an attachment such as a sledge or a cart. This may not be necessary but its inclusion increases the usefulness of the simulation for real-world applications.

Robot Class Functions

The generate robot function within the robot class allows for the construction and instantiation of a "physical" robot within the simulation environment. The process of creating the robot through this function is as follows. The function will parse the gene of the robot, first removing the number of body segments. A loop will begin, iterating through each segment of the body of the robot. The next data to be stripped from the gene, is the weight and size of each

body segment. Then the function iterates to the legs for each of the body segments, and then each legs' strength and size will be adjusted to reflect the information in the gene. Through this process, a final robot design will be instantiated into the testing area, and will automatically begin to attempt locomotion.

Algorithm Class

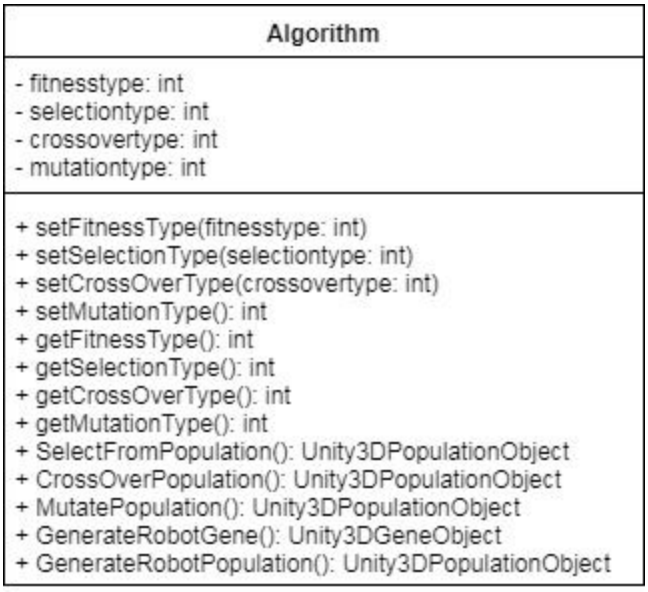

Figure 6. Algorithm Class Diagram

Algorithm Class Data Members

Fitnesstype: An integer value used to determine the rankings of the individual robots. The fitnesstype will either represent a fitness selection based on either velocity or distance traveled, depending on which type the client determines to be most valuable.

Selectiontype: The selectiontype is an integer value that holds enumerated values of which methodology the client needs. These methodologies are used to select which robots are able to pass on their genetic traits.

Crossovertype: An integer value representing the method that selects which genes are shared between members of a population. The types included at this point are paired, randomized, and weighted. More types can be included, if necessary.

Mutationtype: An integer value that represents the type of mutation chosen for the algorithm. Currently these types are randomized, favored, micro, and macro. More types are able to be added if the client deems it necessary.

Algorithm Class Functions

The GenerateRobotGene function will create a genetic code by which the robot class will create the robot within the simulation. The gene will be constructed using the constraints within the robot class, as well as the methodologies presented in the algorithm's structure. The function will first get all the min and max constraints from the robot class. Using those constraints, the class will create the Gene. First, the function will determine how many segments the body will require through randomized number generation. This value will be stored in a string. Appended to the string for each segment will be an associated weight and size. Then for each legged segment a value will be appended for the size and strength of legs on the specified segment. Finally the algorithm will attach any weights to the robot as specified from the user settings in the robot menu. This function will be used to create a gene at random for the first generation of

robots; following generations will be generated through the remaining functions of the robot class.

The Generate Robot Population will, upon first generation, create a list of robot genes to be tested as the first robot population. This will consist of a loop iterating over the size of the population to create a list of genes.

The SelectFromPopulation function will be passed a population object, sort the population by fitness and select the members whose genes will be passed forward. The result from this function is a smaller list of the subset of the population who will be represented in the next generation. The selection process leads to the crossover portion of the algorithm class as it is a pivotal portion of the genetic sequencing.

The CrossoverPopulation function takes various portions of the genes from the selected subpopulation and uses those to create a full list of genes that represent the next generation of robots to be tested. The crossover techniques used will be selected by the user. The types of crossover as previously specified are then used to return a list of a new population, none of whom have been tested in the simulation.

The final function in the Algorithm Class is the MutatePopulation function. This function will take a full population and slightly and randomly vary different portions of the population's features. The method by which this mutation occurs is specified by the user in the user interface.

Simulation Class

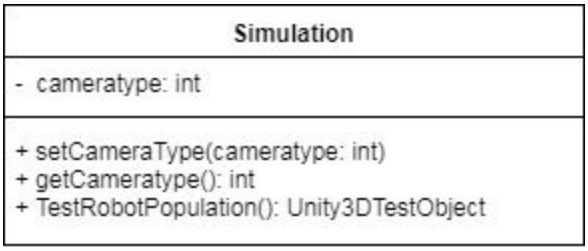

Figure 7. Simulation Class

Simulation Class Data Members

Cameratype: This integer data type reflects the current camera being used within the simulation to view the robots as they are being tested. The various angles through which the simulation may be viewed will be documented in an integer system.

Simulation Class Functions

The TestRobotPopulation Function is the culmination of many other functions all being utilized for the purposes of the judgment of each robot's design in the specified terrain. The first portion of the function will ask the terrain to create a terrain. Then the function will retrieve the population to be tested from the algorithm class. Each robot gene in the population will be generated by the robot class, then after the specified amount of time, the robot is assigned a fitness. This fitness score will be passed back to the algorithm alongside the population to the algorithm class for the creation of the next generation.

Portability

The inclusion of the Unity 3D engine has allowed our software to become extremely portable. Unity will be a very powerful tool used during the development of our software. As of

December 2017, Unity supports the exporting of programs and games to twenty-nine different platforms. These include not only the major platforms such as Windows, macOS, PlayStation 4, and the Xbox One, but smaller and lesser-used platforms like Amazon's FireOS, WebGL, Samsung's SmartTV operating system, and Apple's tvOS. Transferring work between a client's workgroup becomes a lot easier, as the need to worry about platform support is essentially non-existent. Unity includes the ability to export to Virtual Reality and Augmented Reality platforms like GearVR, Oculus Rift, and Apple and Google's ARKit and ARCore, respectively. Possible future work may include a client working on a simulation at a laboratory computer, transferring the simulation to their phone, and continuing said simulation on their commute home. Augmented and virtual reality can allow for a client to "watch" their robot from an up-close and personal viewpoint.

Reuse

Due to the use of Unity as our platform and the ability of a user or client to modify the constraints for each robot simulation, reusability of our program becomes one of its strongest features. Unity's rising popularity ensures that our core physics engine remains supported for the foreseeable future. Because the user interface runs atop Unity, the entirety of our program will also remain supported. Unity supports multiple languages in which we can write our classes and functions. These include C# (an always popular language), a JavaScript variant called UnityScript, and Boo, which is a variant of Python. We chose C# for a multitude of reasons. C# is by far the most popular language used by Unity developers. This means that the developer base for C# is much larger and, in turn, the finding support and solutions for any bugs found

during our development will be much greater. This also raises the possibility that other developers could create their own “plug-ins” for our program, to either add wanted features or modify existing ones. These plug-ins could then eventually be resorbed or co-opted back into the fold of our program. If a user or client needs to expand upon a constraint or modify the way by which certain elements interact with each other, they will have a greater chance of being able to do so if we use a language that has a wide user-base. This has another effect on reusability: finding and fixing faults suddenly becomes a lot easier. Working with a small development team, it can be difficult to find and fix faults, especially if the team is working on adding a new feature or improving an already existing one. Using C# as our language will allow other developers to create their own fixes or at the very least suggest possible fixes to our team. These fixes can then be tested and if found to be in good order, applied to our own code. This can save the development team a massive amount of time that would otherwise be spent finding and coming up with its own fixes. However, this is only part of the reusability of our program. Since the core of our program rests upon Unity and its physics engine, the scripting language used is able to be switched if in the future—however unlikely—C# suddenly declines. Indeed, there are even already existing programs that are able to convert most C# scripts into UnityScript. While these converters are never perfect, it would still greatly reduce the amount of time needed to switch languages, which in turn can help reduce development and maintenance costs.

Appendix I: Function Descriptions

Terrain Class Function Descriptions

setStructureType():

Input: integer Output: 1 or -1 (1 being good data, -1 being bad data)

Description: Sets the value of the structuretype variable in the terrain class

setTextureType():

Input: integer Output: 1 or -1 (1 being good data, -1 being bad data)

Description: Sets the value of the texturettype variable in the terrain class

setTextureSize():

Input: float Output: 1 or -1 (1 being good data, -1 being bad data)

Description: Sets the value of the texturesize variable in the terrain class

setTextureDensity():

Input: float Output: 1 or -1 (1 being good data, -1 being bad data)

Description: Sets the value of the texturedensity variable in the terrain class

setObstacleType():

Input: integer Output: 1 or -1 (1 being good data, -1 being bad data)

Description: Sets the value of the obstacletype variable in the terrain class

setObstacleSize():

Input: float Output: 1 or -1 (1 being good data, -1 being bad data)

Description: Sets the value of the obstaclesize variable in the terrain class

setObstacleDensity():

Input: float Output: 1 or -1 (1 being good data, -1 being bad data)

Description: Sets the value of the obstacledensity variable in the terrain class

setGravity():

Input: float Output: 1 or -1 (1 being good data, -1 being bad data)

Description: Sets the value of the gravity variable in the terrain class

getStructureType():

Input: void Output: integer

Description: Gets the value of the texturettype variable in the terrain class

getTextureType():

Input: void Output: integer

Description: Gets the value of the texturettype variable in the terrain class

getTextureSize():

Input: void Output: float

Description: Gets the value of the texturesize variable in the terrain class

getTextureDensity():

Input: void Output: float

Description: Gets the value of the texturedensity variable in the terrain class

getObstacleType():

Input: void Output: integer

Description: Gets the value of the obstacletype variable in the terrain class

getObstacleSize():

Input: void Output: float

Description: Gets the value of the obstaclesize variable in the terrain class

getObstacleDensity():

Input: void Output: float

Description: Gets the value of the obstacledensity variable in the terrain class

getGravity():

Input: void Output: float

Description: Gets the value of the gravity variable in the terrain class

generateTerrain()

Input: void Output: Unity3DTerrainObject

Description: Generates a Unity3D Terrain Object for use within Unity,

will be used as the terrain robots traverse

Robot Class Function Descriptions

setMinWeight():

Input: float Output: 1 or -1 (1 being good data, -1 being bad data)

Description: Sets the value of the minweight variable in the robot class

setMaxWeight():

Input: float Output: 1 or -1 (1 being good data, -1 being bad data)

Description: Sets the value of the maxweight variable in the robot class

setMinSegments():

Input: integer Output: 1 or -1 (1 being good data, -1 being bad data)

Description: Sets the value of the minsegments variable in the robot class

setMaxSegments():

Input: integerOutput: 1 or -1 (1 being good data, -1 being bad data)

Description: Sets the value of the maxweight variable in the robot class

setMinSegmentLength():

Input: float Output: 1 or -1 (1 being good data, -1 being bad data)

Description: Sets the value of the minsegmentlength variable in the robot class

setMaxSegmentLength():

Input: float Output: 1 or -1 (1 being good data, -1 being bad data)

Description: Sets the value of the maxsegmentlength variable in the robot class

setMinLegs():

Input: integer Output: 1 or -1 (1 being good data, -1 being bad data)

Description: Sets the value of the minlegs variable in the robot class

setMaxLegs():

Input: integer Output: 1 or -1 (1 being good data, -1 being bad data)

Description: Sets the value of the maxlegs variable in the robot class

setMinLegForce():

Input: float Output: 1 or -1 (1 being good data, -1 being bad data)

Description: Sets the value of the minlegforce variable in the robot class

setMaxLegForce():

Input: float Output: 1 or -1 (1 being good data, -1 being bad data)

Description: Sets the value of the maxlegforce variable in the robot class

setCarryWeight():

Input: float Output: 1 or -1 (1 being good data, -1 being bad data)

Description: Sets the value of the carryweigt variable in the robot class

setDragWeight():

Input: float Output: 1 or -1 (1 being good data, -1 being bad data)

Description: Sets the value of the dragweight variable in the robot class

getMinWeight():

Input: void Output: float

Description: Gets the value of the minweight variable in the robot class

getMaxWeight():

Input: void Output: float

Description: Gets the value of the maxweight variable in the robot class

getMinSegments():

Input: void Output: integer

Description: Gets the value of the minsegments variable in the robot class

getMaxSegments():

Input: void Output: integer

Description: Gets the value of the maxsegments variable in the robot class

getMinSegmentLength():

Input: void Output: float

Description: Gets the value of the minsegmentlength variable in the robot class

getMaxSegmentLength():

Input: void Output: float

Description: Gets the value of the maxsegmentlength variable in the robot class

getMinLegs():

Input: void Output: integer

Description: Gets the value of the minlegs variable in the robot class

getMaxLegs():

Input: void Output: integer

Description: Gets the value of the maxlegs variable in the robot class

getMinLegForce():

Input: void Output: float

Description: Gets the value of the minlegforce variable in the robot class

getMaxLegForce():

Input: void Output: float

Description: Gets the value of the maxlegforce variable in the robot class

getCarryWeight():

Input: void Output: float

Description: Gets the value of the carryweight variable in the robot class

getDragWeight():

Input: void Output: float

Description: Gets the value of the drag weight variable in the robot class

generateRobot():

Input: void Output: Unity3DRobotObject

Description: Generates a Unity3D Robot Object for use within Unity as a robot that will traverse the terrain

Algorithm Class Function Descriptions

setFitnessType():

Input: integer Output: 1 or -1 (1 being good data, -1 being bad data)

Description: Sets the value of the fitnesstype variable in the robot class

setSelectionType():

Input: integer Output: 1 or -1 (1 being good data, -1 being bad data)

Description: Sets the value of the selectiontype variable in the robot class

setCrossOverType():

Input: integer Output: 1 or -1 (1 being good data, -1 being bad data)

Description: Sets the value of the crossovertype variable in the robot class

setMutationType():

Input: integer Output: 1 or -1 (1 being good data, -1 being bad data)

Description: Sets the value of the mutationtype variable in the robot class

getFitnessType():

Input: void Output: integer

Description: Gets the value of the fitnesstype variable in the robot class

getSelectionType():

Input: void Output: integer

Description: Gets the value of the selectiontype variable in the robot class

getCrossOverType():

Input: void Output: integer

Description: Gets the value of the crossovertype variable in the robot class

getMutationType():

Input: void Output: integer

Description: Gets the value of the mutationvariable in the robot class

generateRobotGene():

Input: void Output: Unity3DGeneObject

Description: Generates a Unity3D Robot Object for use within Unity

getFitnessType():

Input: void Output: Unity3DPopulationObject

Description: Generates a Unity3D Population Object for use within Unity

Simulation Class Function Descriptions

setCameraType():

Input: integer Output: 1 or -1 (1 being good data, -1 being bad data)

Description: Sets the camera display location in the simulation class

getCameraType():

Input: void Output: integer

Description: Gets the camera display location in the simulation class.

TestRobotPopulation():

Input: void Output: Unity3DTestObject

Description: Generates a Unity3D Test Object for the robot population within Unity.

Appendix II: Team Details

Throughout the development of the design document, Kevin Beck explored many different topics related to the exact implementation of the various algorithmic applications as they related to the project's domain. Through this investigation, methods and descriptions of the specifics of genetic algorithms were shared and discussed at length during team meetings. As the writing of the design document progressed, Kevin developed a few specific areas. The brief descriptions of the controller classes were written and implemented. He wrote and revised the abstract, detailed function descriptions, design analysis, and cohesion and coupling. Kevin established dates and times for group meetings alongside Kristy. Kevin also contributed in the restructuring of the classes to accommodate the project's form as it moves towards implementation as it relates to Unity3D.

Joseph Borsodi was responsible for creating the project block diagram and explaining the overall system overview. He described how the system will be used once the client receives it as well as basic operation. Joe was also in charge of creating the data flow diagram and going into detail about how input will be acquired from the user. He described the basic portions of the user interface and how input will flow from the ui into the intended classes and from the classes onward. To help the team move into the implementation phase during the spring semester, Joe created the Function Description Appendix which includes a description of every function found in the main classes. The idea behind this appendix was to be a way to quickly identify every functions purpose along with its correct input and output. During the creation of this document Joe attended all team meetings organized by our workflow leader Kristy Schworn.

Ryan Romanosky was responsible for writing the description of document section, each class' data members description, as well as the portability and the reuse section. He described the purpose and use of the design document, its ties to the specification document, as well as the intended audience for the document. He also created the class data member section, giving an overview of each class' data members, which data type they hold, as well as how they affect the final simulation. He wrote how the project, thanks to its reliance on the Unity engine, could be exported to multiple platforms, including the possibility of future development for rising platforms such as augmented and virtual reality-based ones. He created the reusability section, describing how the use of C# as the scripting language and Unity as the core engine can allow for other developers to modify and expand upon our original project. He also dictated the notes for the team meetings, of which he attended all set by our Kristy Schworn.

Kristy was the workflow leader for the design document. Group meetings were coordinated by her, as well as outlining the document. She also created class diagrams which built off of the class overviews. Using the initial chart, she created a more detailed description of each class and all of the functions belonging to it. The Terrain, Robot, Algorithm, and Simulation classes hold every major function for the project. From these diagrams, a more detailed description can be given. Kristy contributed to the terrain controller description as well as the robot controller description. She also contributed to the simulation function description. Kristy revised the document, making corrections to any errors with spelling or grammar. The group came together to do a final revision before submission.

Appendix III: Workflow Authentication

By signing on the lines below it will be inferred that you endorse the Team Details found within Appendix II. You, as a member of the team, verify all information to be correct and all team contributions fairly recognized.

_____	_____	_____
Name	Signature	Date
_____	_____	_____
Name	Signature	Date
_____	_____	_____
Name	Signature	Date
_____	_____	_____
Name	Signature	Date