University of Nevada
Reno

Scripted Artificially Intelligent Basic Online Tactical Simulation

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
with a major in Computer Science

by

Jesse D. Phillips

Dr. Frederick C. Harris, Jr., Thesis Advisor

May, 2008

THE GRADUATE SCHOOL

We recommend that the thesis
prepared under our supervision by

JESSE D. PHILLIPS

entitled

Scripted Artificially Intelligent Basic Online Tactical Simulator

be accepted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

Frederick C. Harris Jr., Ph.D., Advisor

Sergiu Dascalu, Ph.D., Committee Member

Scott Bassett, Ph.D., Graduate School Representative

Marsha H. Read, Ph. D., Associate Dean, Graduate School

May, 2008

Acknowledgments

The work shown in this thesis has been sponsored by the Department of the Army, Army Research Office; the contents of the information does not reflect the position or policy of the federal government. This work is funded by the CAVE Project (ARO# N61339-04-C-0072) at the Desert Research Institute.

I would like to first thank my committee, in particular my advisor Dr. Frederick C. Harris Jr. Without his guidance and support I could not have finished this thesis. Also to the rest of my committee, Dr. Sergiu Dascalu and Dr. Scott Bassett for their patience and help in making this thesis possible.

Finally I would like to thank my friends and family. Billy Brandstetter, Mike Dye, and Jason Porterfield for the original work we did together back in undergrad and more recent work in graduate school. To Jeff Stuart and Joe Jaquish, thank you for your help over the first couple semesters of graduate school. Mike Penick thanks for some fun times while going to grad school and working til all hours of the morning on compilers. Roger Hoang for the great work you did on rendering some finite terrain and letting me port it. To my family, thank you for all your support during all of my school, especially these last three years.

Abstract

For many years introductory Computer Science courses have followed the same teaching paradigms. As these paradigms are run on simple console windows there is an area for an interactive way of seeing what code does. This thesis presents details of the idea, specification, design, and functionality of the Scripted Artificially Intelligent Basic Online Teaching Simulator, an interactive game that helps reinforce what is taught in class. Through reinforcement students script vehicles to fight each other in a three dimensional environment. In this environment users can play with other people and learning basic programming techniques along with Artificial Intelligence scripting. The users can navigate through the 3D world using a third-person camera or Blind Mode. This allows the user to use techniques learned in class and to observe what happens, resulting in immediate reinforcement of skills and concepts.

Contents

Acknowledgements **i**

Abstract **ii**

List of Figures **v**

List of Tables **vi**

1 Introduction **1**

2 Background **3**
- 2.1 Video Games . . . 3
  - 2.1.1 Engines . . . 3
  - 2.1.2 Games . . . 8
- 2.2 Artificial Intelligence . . . 10
  - 2.2.1 Scripting . . . 10
  - 2.2.2 Nueroevolution Engines . . . 10
  - 2.2.3 Inputs . . . 11
- 2.3 Terrain . . . 11
- 2.4 Previous Work . . . 12

3 Proposed Idea **13**
- 3.1 Teaching Tools . . . 13
- 3.2 Squad Control . . . 13
- 3.3 Neuroevolution . . . 14
- 3.4 Sensors . . . 15
- 3.5 Scripting . . . 15
- 3.6 Blind Mode . . . 16
- 3.7 Terrain and Virtual Reality . . . 17

4 Software Engineering **18**
- 4.1 Requirements . . . 18
- 4.2 Use Cases . . . 19
- 4.3 Modeling and Design . . . 21
  - 4.3.1 Class Hierarchy . . . 21

4.3.2 User Interface 26

5 User Manual **28**

5.1 System Activity 28

5.2 System Requirements 29

5.3 Starting 29

5.4 Single Player 32

5.4.1 Vehicle Control 33

5.4.2 Script Editing 36

5.4.3 Blind Mode 41

5.4.4 Mazes 44

5.4.5 Squad Control 47

5.5 Multi-Player 49

6 Conclusions and Future Work **52**

6.1 Summary and Conclusions 52

6.2 Future Work 53

Bibliography **57**

List of Figures

4.1 Use Case Diagram 20
4.2 Overall Heirarchy 21
4.3 Graphical User Interface Class Heirarchy 22
4.4 Base Class Heirarchy 23

5.1 System Activity 28
5.2 Windows Startup Screen 30
5.3 Linux Startup Screen 31
5.4 Single Player Start - Maze 32
5.5 Single Player Start - Combat 33
5.6 Command Menu 34
5.7 Help Menu 35
5.8 Tank Moving Forward 36
5.9 Tank Moving Forward Second 37
5.10 Tank Turning Left 37
5.11 Tank Turning Right 38
5.12 Tank Looking Down 38
5.13 Tank Looking Left 39
5.14 Tank Looking Right 39
5.15 Tank Looking Up 40
5.16 Tank Firing 40
5.17 Basic Script Editing 42
5.18 Advanced Script Editing 43
5.19 Starting Blind Mode 43
5.20 Allies Around 44
5.21 Enemies Around 45
5.22 Sensors 45
5.23 Maze GIF Image 46
5.24 Game Maze 46
5.25 Squad Script - Sending 48
5.26 Squad Script - Receiving 50

List of Tables

4.1 Functional Requirements 19
4.2 Non-Functional Requirements 19
4.3 User Interface Requirements 27

5.1 System Requirements 29
5.2 Python Requirements 29
5.3 User Interface - Buttons 34
5.4 Vehicle - Movement Controls 34
5.5 Editors 35
5.6 Python Commands - Beginner 41
5.7 Python Commands - Advanced 41
5.8 mazes.txt File Format 47
5.9 Python Commands - Squad Send 47
5.10 Python Commands - Squad Receive 49

Chapter 1

Introduction

For many years introductory Computer Science courses have followed the same teaching paradigms. These paradigms are run on simple console windows, so there is an interactive way of seeing what code does. We will show some previous work in teaching tools working inside of video games. Then we will go in to greater detail about the idea we came up with for an interactive teaching tool. In order to develop a project that is trying to do as much as we are proposing, we had to design everything before hand and we will be showing that.

This thesis presents some previous work in the video game industry, the open source game industry, and teaching tools with video games. We then go on to show what can be used to help teachers with reinforcing what they have covered in class. This thesis then presents details of the specification, design, and functionality of the Scripted Artificially Intelligent Basic Online Teaching Simulator (SAI-BOTS), an interactive game designed for reinforcing concepts and skills. In this environment users can interact with other people and learn basic programming techniques along with artificial intelligence scripting.

Higher level Computer Science students will be able to use SAI-BOTS for a new artificial intelligence technique called Neuroevolution. This gives SAI-BOTS a broader appeal compared to some of the programs covered in the background. Other new ideas proposed relate to sensor implementation, graphic rendering, and storage. The squad controls that are designed and implemented will show what can be done with squads rather than just having them bunched together.

The rest of this thesis is structured as follows: Chapter 2 covers some of the previous research in the field of video games and teaching tools with video games. Our proposed idea for where there are holes in the previous work is covered in Chapter 3. Our design of SAI-BOTS is shown in Chapter 4. Chapter 5 shows requirements and how to use SAI-BOTS. Finally, in Chapter 6 we give our results and conclusions and show some ideas for future work.

Chapter 2

Background

2.1 Video Games

The video game industry is comprised of two primary ideas, game engines and games. Section 2.1.1 covers some of the engines that are currently used, while Section 2.1.2 covers some of the actual games that are available. Since SAI-BOTS is supposed to be a interactive and enjoyable teaching tool, modeling SAI-BOTS after some games helps with some of the interactivity and enjoyment of games.

2.1.1 Engines

Game engines are the core backend of the video game industry. Without the engines, developers would have to write completely new code for input, graphics, and artificial intelligence. Most engines share common elements, some of which use different algorithms. The primary elements of a game engine are user input and interface, graphics rendering, model and image loading, audio, physics, artificial intelligence, and networking. Some other elements that developers have added to game engines are a scene graph, scriptable gameplay, and content creation tools.

There are many different game engines available in the video game industry. They are split into two different groups: open-source and commercial. In the following Sections we will be talking about some of the engines that influenced work on SAI-BOTS.

APOCALYX

APOCALYX [6] is a open-source 3D game engine. The primary design consists of a game engine that allows for a player to program his own team of bots and play them against other players bots online. APOCALYX created for bots to play against each other obtains input through a graphical user interface. This input primarily tests different values and manipulate the scene until the game runs how the user wants.

Graphics rendering is done through OpenGL [23]. Along with OpenGL APOCALYX uses the OpenGL Shading Language(GLSL) [24] to render objects faster and in a higher detail. The rendering engine has many added features making game creation easier. APOCALYX supports sky boxes, terrain creation and rendering, wave simulation, and multiple viewports. The rendering engine links with the model and texture loading available. The supported model types are 3DS [42], OBJ [45], MD2 [26], MD3 [25], Cal3D [46] and MS3D [12], with the last four supporting animation. The supported image types are JPG, PNG, and TGA.

OpenAL [43] is used for 3D Sound in the engine. MIDI, OggVorbis, and MP3's can be used to play music and other sounds in the world. The scripts users have the ability to check for sounds around them. This ability allows for the bots to try and determine where a sound came from and decide if it acts on that information. The physics that is built into APOCALYX includes a particle-based physics simulator, ODE physics simulator [40], and collision detection.

The artificial intelligence for bots in the world can be scripted through many different languages. Together with the scripting languages, APOCALYX also has a built in A* pathfinding algorithm [50] that the scripts can use to get bots around the environment. For multiplayer purposes APOCALYX uses WinSock2, LuaSocket or Etwork. These networking subsystems allow for users to play their scripted bots against other players.

The engine includes a couple extra features that are not necessary for game engines. The scene graph orders all objects in the world according to distance from the camera. Along with the scene graph, APOCALYX has scriptable gameplay through

Lua [11]. Even though the engine is written in C/C++, this scripting allows for a user to create a game without recompiling the engine everytime he/she changes a parameter for the world.

CryENGINE2

CryENGINE2 [13] is a commercial game engine developed by Crytek [14]. The primary design behind CryENGINE2 is to make the most realistic video game engine from the graphics to the physics. The artificial intelligence is scripted in CryENGINE2, but the user cannot change the scripts on the fly. The input system is the same as all other first-person shooter engines that take basic keyboard, mouse, and joystick input to control characters. Along with the input, the user interface is similar to most first person shooters showing a radar, ammunition remaining and available, health and a targeting reticle.

The graphics rendering system uses DirectX [34] 9 and 10, for use on Microsoft Windows XP and Vista, respectively. Also included in the rendering engine is a script-based shader system, level of detail terrain up to sixteen kilometers away, voxel objects for hard to create heightmap objects, and three different types of fog to allow for better immersion. CryENGINE2 uses a model format created by Crytek that has exporters from 3ds Max [5].

The audio subsystem allows for creation of more complex sounds resulting in more realistic sounds. It has an improved music playback system that can allow for film-like sound tracks. Hearing sounds seamlessly from everywhere around allows the user to be immersed in an environment that sounds realistic. There is an integrated physics system in CryENGINE2. This system supports vehicles, rigid bodies, liquid, rag doll, cloth, and soft body.

There are some interesting features of the artificial intelligence in CryENGINE2 that have not been completely utilized. Team-based artificial intelligence is a feature that not many game engines implement. Additionally, the ability to define behaviors with scripts exists. The network system takes care of managing all connections for

multiplayer modes. It is setup as a client-server architecture that allows for a low-latency system.

There is a second type of scripting system built into CryENGINE2 using Lua. This system allows for users to tweak game parameters, play sounds, and load different types of graphics. A graphical editor for designing game logic lets developers connect boxes and define properties rather than write code. CryENGINE2 supports multithreading, unlike most game engines. This support allows for the engine to determine how many processors there are and distribute computation and simulation across all processors.

Unreal Engine 3

Unreal Engine 3 [21] is a commercial cross-platform game engine developed by Epic Games [20]. The primary goal of Unreal Engine 3 was to make a complete development framework for a video game platforms. Epic Games created a complete scripting language for developers to use for writing artificial intelligence programs. User input and the graphical user interface come in the same form as most other first-person shooters. New interface variations may be created through a supplied user interface editor.

There are many different features that Unreal Engine 3 has, but due to the long development cycle many engines have implemented the same features. The primary development for the engine was done in DirectX 9 for Windows PC, but there is also an OpenGL rendering engine that allows for games using the engine to run in Linux as well. The rendering engine includes some of the primary features to make the game world look more detailed. Unreal Engine 3 has its own model format that gets preprocessed.

The audio subsystem allows developers to place sounds exactly where they want and set many different parameters for better control. Along with those parameters 3D sound positioning, spatialization, doppler shift, and multi-channel playback are available. Rather than writing their own physics system, Epic Games incorporated

Ageia PhysX [2] into their engine. This physics system supports rigid bodies, ragdolls, complex vehicles, and dismemberable objects. The physics subsystem talks with the audio subsystem to give some physics driven sounds.

The artificial intelligence in Unreal Engine 3 is the most advanced in current video game engines. There is the ability for pathfinding to take a non-player character through the world and for characters to know to press switches or hit buttons. The pathfinding has a more improved part as well that allows for non-player characters to go off their path in order to find cover or even complete a short-term goal. Epic Games also implemented team-based artificial intelligence so squads can react. This includes team coordination, objective management, and long-term goals. Due to the immense success of the multiplayer version for Epic Games' Unreal Tournament series, networking is a key part to the Unreal Engine. The client-server model that is employed supports up to 64 players on a dedicated server with 16 players on a client acting as the server at the same time. The only thing different between the multiplayer and singleplayer modes is that one plays with other users, but setting up games is exactly the same.

The biggest feature that Unreal Engine 3 has over other engines is the ease of use of the content creation tool. This tool allows developers to create new content for a game using the engine. Everything from the lighting to the scripting is in this one creation tool. Incorporated into this tool are the visual scripting and cinematic systems that let developers connect together the features of a game they want to be available.

Other Game Engines

There are many other game engines and not all of them could be covered. Some other commercial game engines are Doom 3 Engine, Offset Engine, Source Engine, and Torque Game Engine. All of these engines are widely used by the video game industry. Open-source game engines are not as widely used as the commercial engines, but there are still several to choose from. Delta3d, Ogre, and the Irrlicht Engine are

all cross-platform open-source game engines.

2.1.2 Games

The following sections cover some of the video games that use the engines mentioned in Section 2.1.1. The primary features of the games that we will discuss are the terrain used, artificial intelligence, and user interface.

Crysis

Crysis [17] is the game developed in unison with CryENGINE2 by Crytek. The basic terrain used in Crysis is a heightmap system that determines the elevation of the world at certain points. Along with the heightmap, there is a voxel system that allows for interior areas and awkward shapes to be created and placed in the world. This allows for some very high detailed worlds in which to do combat. The high detail nature of the world allows for the user to get immersed in the game, while also trying to perform simple manuevers like taking cover behind a building. This type of world can make it more difficult to design an artificial intelligence scheme to work well.

The artificial intelligence in Crysis was completely designed to be as realistic and believable as possible. By using some pathfinding algorithms to get around the world, the artificial intelligence also uses tactical manuevers to attack enemies. Working as squads to flank enemies as well as hiding in the world to ambush someone is a common element Crytek was seeking to acheive. When not in combat, the artificial intelligence soldiers exhibit scripted lifelike behavior like smoking, yawning, talking, and a couple other actions. The user interface that Crysis uses is the base version built into CryENGINE2.

GUN-TACTYX

GUN-TACTYX [7] is a game developed by the creator and main developer of the APOCALYX engine. There are many different terrains available to GUN-TACTYX through the APOCALYX engine. The primary terrain used is the heightfield terrain that has a simple level of detail algorithm, allowing players to play in a world mapped

around the real world. The secondary terrain is an infinite terrain algorithm that allows the players' bots to go anywhere and not worry about falling off the map. These terrain features give two different styles of play in which users can program their bots.

The main developer for APOCALYX and GUN-TACTYX also wrote JROBOTS [8], a java version of CROBOTS [39]. Due to the fact that CROBOTS, GUN-TACTYX and JROBOTS are all in the same vein of game types, this information will actually cover all three. The premise behind each game is for players to write scripts that tell how the bots in the world should react. These scripts have access to some very basic functions from the game that allow the bots to do some pathfinding along with taking in sensor data to perform certain actions in the world. The user interface shows all the remaining players' bots and the health of each of them, as well as allows the users to change some parameters on the fly but not changing the scripts on the fly.

Unreal Tournament 3

Unreal Tournament 3 [22] is one of the video games that Epic Games created while working on Unreal Engine 3. The terrain used is a very small area that deals mostly with running in and around buildings while fighting enemies. These maps make great areas for artificial intelligence pathfinding around the world, with a lot of nooks and crannies that the paths can go through.

Artificial intelligence can be changed through a graphical user interface where different blocks can be moved around and connected to each other. The blocks encompass what action should be run when the bot gets to that point. Since the scripts are edited through a graphical user interface, they are not editable on the fly. The user interface that Unreal Tournament 3 employs is the base version built into Unreal Engine 3.

2.2 Artificial Intelligence

2.2.1 Scripting

Most artificial intelligence that is currently scripted is done for simplicity. The benefits of using a scripting language are the ability to change how the artificial intelligence reacts and not needing to recompile the engine source code. Unfortunately most games implement a scripting system in such a way that scripts are not able to be updated while playing the game. Updating scripts is a beneficial feature, since scripts do not need to be compiled, this allows for users to see what new changes to the scripts do while playing the game.

2.2.2 Nueroevolution Engines

Neuroevolution is a growing research field within artificial intelligence. It uses genetic algorithms to evolve artificial neural networks rather than having someone play the game. Genetic algorithms use a fitness function to optimize the output of the neural networks based on what the user wants to accomplish. This idea allows for trainable agents to learn what they are supposed to do faster through offline simulations. The simulations save the highest rated neural networks so the user can test them out in the future.

There is a neuroevolution engine being developed at the University of Texas at Austin called Neuro Evolving Robotic Operatives(NERO) [49]. NERO allows for users to show robots how they should be acting in a world. The neuroevolution algorithm used is called real-time NeuroEvolution of Augmenting Topologies.

Artificial Neural Networks Evolve (ANNEvolve) [3] is an open-source neuroevolution engine. ANNEvolve has the ability to write code in both Python and C/C++ in order to evolve a neural network to do specific tasks. The primary use of ANNEvolve is to evolve a neural network to guide and control a sailboat. They have a complete mathematical model of sailboat physics and a whole population of Artificial Neural Networks to make the sailboat circumnavigate an island.

2.2.3 Inputs

The usual inputs used in most artificial intelligence engines are a mix between visual and auditory. Unfortunately most visual inputs mean that the computer controlled bot can see everything in the world and in essence seem too smart. If the artificial intelligence has too little data it ends up seeming dumb. The data that the artificial intelligence has access to needs to be minimized, while maximizing the appropriate actions it should take.

Scripting systems allow for developers to give access to specific features for the artificial intelligence to use. These features usually entail pathfinding, search for enemies, and attack enemy algorithms. With only a couple of abilities at the command of a script, users can figure out exactly what the artificial intelligence will do depending on what he/she has done.

In order to give the artificial intelligence the ability to see enough of the world to react and follow certain guidelines, there needs to be an artificial intelligence system that can be robust with very little input information. Due to the fact that a scripting system can be used together with neural networks, there needs to be a way for the inputs to relate to scripts and neural networks at the same time. This means that a user would have to be able to tell what certain inputs mean even though it could only be numbers that they are getting back.

2.3 Terrain

There are currently two different types of terrain in video games. Deformable terrain allows for a user to go around a small finite world and destroy the terrain. Level of detail terrain allows for a user to go around a large world, but it stays the same the whole time.

Fracture [33] is a game currently in development by LucasArts [32] that allows for the user to go around and manipulate the terrain as a weapon. As the game is still in development not much is known about how big the world is, but currently

from the videos it seems that the user can only go around a pre-defined area. This could benefit from having a larger world as there would be less load times as well as more area for users to deform the terrain.

All the terrains shown in the video game engines discussed have level of detail algorithms that make them run more efficiently. Unfortunately they do not cover as much area as an out of core terrain as discussed by Danovaro, et al. [15]. Getting out of core terrain to work with a game engine is complicated in that a vehicle can be in an area to be shown which is not loaded into memory. There is some recent work in the field of out of core highly deformable terrain discussed by Brandstetter [27] which may add insight into this dilemma.

2.4 Previous Work

We previously did work on this topic in Scripted Artificial Intelligent Basic On-Line Tank Simulator [9]. We developed a game engine with the Win32 API [35] that had the ability to load Lua [11] scripts on the fly. Their was also a blind mode implemented, but it gave the user the vision of the whole world, rather than a confined space around the vehicle. Multiple users were allowed to play against each other in a free-for-all match, where each user had control of the same number of tanks.

Chapter 3

Proposed Idea

3.1 Teaching Tools

While interactive and entertaining, most video games are not made for educational purposes. In most first-person shooters, like Unreal Tournament 3 and Crysis, the only goals entail using the biggest gun you have available to blow enemies up. Something that GUN-TACTYX does differently is that it allows players to write scripts to control the robots.

This feature can be used as an educational tool for artificial intelligence courses. Teachers can hold competitions in which students write all the scripts used and then the robots can battle. In some games it can still be interactive, but it is most definitely entertaining and educational. Students can take what they learned from class and work on implementing new tactics, as well as being able to watch all of the battles unfold and simultaneously working on some new ideas. We knew that in order to make SAI-BOTS a teaching tool, there needed to be a purpose behind the scripting.

3.2 Squad Control

An important aspect of multiplayer games is the ability to communicate with and coordinate attacks with teammates. Most games only allow the player to send simple commands and then allies can respond with a couple different actions. What we are proposing to do in this area is to allow users to send out commands to other units. These commands can be sent to a whole team, over specific team channels, so people

in the general vicinity can hear them and vocalized so people within one-hundred units can hear them. This will allow some units to act as receivers only and act depending on what intelligence is sent to them. Since the commands can be sent to anyone, as well as just a squad of units, different vehicles can be grouped together just by sending commands out.

Commanding a squad adds a higher level of difficulty to designing the system. This meant we had to figure out what information we wanted to give access to for sending commands to other units. Unlike most engines, we decided to allow the user to control any of their vehicles. Through squad commands the user should be able to send out directions if a vehicle is in trouble, an enemy is around, or even how to navigate through a maze. This functionality is key to the tactical side of SAI-BOTS and allows for multiple strategies to be used.

3.3 Neuroevolution

Neuroevolution can be used for creating many different types of advanced artificial intelligence. We discovered in previous engines that neuroevolution typically allowed only one fitness function to be used for evolution. While this makes complete sense, we felt that there should be an easier way to allow for one unit to be evolved as a commander and the others as operatives. We feel that simulation engines should allow for the greatest number of ideas to be developed. As we decided to use Python [19], we went along with the notion of developing the neuroevolution techniques to be built into the engine and allow for developers to write the fitness functions in Python.

The primary purpose of improving our previous work is the integration of neuroevolution. Due to the developing nature of neuroevolution there have been many different ways to implement neuroevolution but very few ways to start testing a new idea, whether it is developing a new algorithm or trying different fitness evaluations at the same time.

3.4 Sensors

Something that came up when designing SAI-BOTS was the idea of being able to use neural networks to control the vehicles. Since neural networks take sensor numbers as input, it made sense that the primary form of input is sensor data from around the vehicle. We knew that we wanted to have some type of sensors for detecting how far from walls, allies, and enemies a bot would be, but to make it simpler for neural networks we came up with the option of simple sensors that relate to radar as well. This will give the user more options to improve their scripts with using more inputs and making more advanced scripts. Along with the use of a graphical context, we decided that for the best possible use of a simulation engine is to design it such that it can be run on any number of machines without graphical output. This allows for users to run their neuroevolution algorithms at faster speeds without having to render graphics, and at a later time run a neural network script that was created with the simulation.

With proposing a different type of sensor, we are going to allow the user to base decisions off numbers, exactly how the computer sees the input. The user should have access to all the sensors input and can decide to what he/she wants the script to react. This expanded set of sensors allows for multiple inputs to be referenced together and create a general outline of what is around the vehicle. We based these sensors around what is currently being done in research areas of robotics.

3.5 Scripting

We took a different approach to scripting compared to most other engines. Rather than using a language that needs to be compiled to run, thus making it a faster approach to controlling entities, we decided that by using Python we should take advantage of being able to load scripts on the fly. The ability to load scripts on the fly allows users to write scripts while the game is running. This led to a design of employing a user interface system that is cross-platform and dictated the type of

editor to use. Something else most scriptable engines do not allow users to do is control an entity themself. Our program will allow for the user to understand how the entity is controlled and learn better ways to write scripts.

Since scripting is becoming a more common approach in engines, even allowing users to write complete games through scripts, it made complete sense to approach scripting this way. Most engines do not allow for the artificial intelligence to be changed on the fly, but we decided to take this approach for the simple fact that it will keep a form of immersion rather than the user having to exit the program and recompile a script after changing it. This also improves the ability of SAI-BOTS to be used as reinforcement teaching for a class since students can try new things and reload the script.

3.6 Blind Mode

We decided to have the ability for the user to play the game in a blind mode. Most important is the new idea of only allowing for a small radar and sensors to be shown along with what they are seeing. The user will then be able to write scripts more effectively based on the sensors. Knowing the distance from walls, a user should be able to write a simple script that can navigate through a maze without hitting the walls. Although this pathfinding may not follow any direct pathfinding algorithm, and therefore might go back over area it has already covered, scripts can be used initially for pathfinding with neuroevolution as a better option in the future.

The mode originally came up as a way for the user to write scripts and even control a vehicle, while being able to see exactly how the computer is seeing what the script will see. By working with scripts in SAI-BOTS robots could even be programmed more easily through using a blind mode like this and even using it as a base. Due to the complexity of adding more sensors to each vehicle we are also going to have to come up with a more efficient way of mapping the access to all the sensors.

3.7 Terrain and Virtual Reality

Most game engines use a very simple terrain algorithm to make the world look like either it has more detail, while being efficient, as well as looking like it goes on to infinity or it is a small area while being deformable. We decided that we wanted to support two different types of terrain. The first terrain is created from a simple black and white bitmap of a maze, while the second terrain is created from reading in a digital elevation model of real world data.

Both terrains use a memory efficient rendering method where the data is read into a vertex buffer object. Indices for the data to be rendered are then set to build a two-hundred and fifty-six by two-hundred and fifty-six grid, with one side having some extra points. In the rendering process the index array is loaded in and rotated so extra side points are created to allow for different levels of detail in the terrain to be stitched together. This places the terrain completely on the graphics card for faster rendering.

We decided that we really want to allow for a user to run either the server or a client in a CAVE [28]. A client would be much more complicated to implement in the CAVE as we are also trying to figure out a way to have a three dimensional user interface. This type of environment would allow for users to gain a higher level of immersion and view things more like a commander would in current video games.

Chapter 4

Software Engineering

Chapter 2 introduced how entry level Computer Science and multiple Artificial Intelligence courses are taught. Chapter 3 covered the idea behind SAI-BOTS and why it is being developed. Reinforcing teachings of entry level courses in an interactive environment is a worthwhile project. The framework behind the system should provide an interface that makes it easier for new programmer's and artificial intelligence programmer's to see what their code does. An understanding of the goals, requirements, and functionality of the system allows for the building of a complete framework and system for classes.

This chapter covers the requirements, use cases, class model, and user interface design. We are following the methodology outlined in [4, 44]. The requirements represent the basic features of an interactive teaching simulator and the technologies needed to support them. The use cases describe the basic functionality of the system. The class structure along with more detailed descriptions of user interface design are presented later in the chapter. The aim of this chapter is to provide an understanding of the framework and methodology behind SAI-BOTS.

4.1 Requirements

To know how the rest of the system works, a basic understanding of the functionality and features is covered in Table 4.1. This was the basic design of the whole system before it was started. These features were conceived through many meetings with

colleagues and professors.

The non-functional requirements, that are shown in Table 4.2, show the supporting technologies, the software features of the whole framework, the basics of the interface, and features future developers can follow.

F01	SAI-BOTS shall provide 3D graphics with the ability to look around.
F02	SAI-BOTS shall have a Graphical User Interface.
F03	SAI-BOTS shall provide a real-time scripting environment.
F04	SAI-BOTS shall provide free-for-all game mode.
F05	SAI-BOTS shall provide team battles game mode.
F06	SAI-BOTS shall provide maze pathfinding game mode.
F07	SAI-BOTS shall provide the ability for the user to play over a network.
F08	SAI-BOTS shall provide real-time terrain deformations.

Table 4.1: Functional Requirements

N01	SAI-BOTS shall run Cross-platform.
N02	SAI-BOTS shall use Open source packages.
N03	SAI-BOTS shall be written in Python.
N04	SAI-BOTS shall use the wxWindows User Interface.
N05	SAI-BOTS scripts shall be written in Python.
N06	SAI-BOTS scripting shall run through Vehicle API calls.
N07	SAI-BOTS shall load models for different vehicles.

Table 4.2: Non-Functional Requirements

4.2 Use Cases

The use case diagram in Figure 4.1 shows the functionality provided to the end users as defined by the functional requirements. Only the major features from the functional requirements are shown. There are two actors in the system that influence how SAI-BOTS is run. The player provides input and scripts, while the server administrator runs a server that will take the input and scripts from players and validate what should happen.

Figure 4.1: Use Case Diagram

4.3 Modeling and Design

4.3.1 Class Hierarchy

Figure 4.2 shows the class hierarchy of the entire system. The entire system consists of the graphical user interface and base systems.

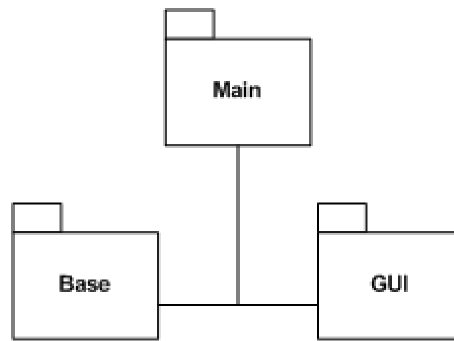

Figure 4.2: Overall Heirarchy

Graphical User Interface Classes

Figure 4.3 shows the class hierarchy of the graphical user interface structure. The graphical user interface is split up into four classes: main GL window, command window, help window, and start menu.

Main GL Window Class This class holds a copy of the game data that is being used and then renders it for the user to see. In order for the user to interact with the scene, this window also has keyboard and mouse functionality built in. The 3D Window talks with the command menu to get information about commander mode, blind mode, and which vehicle is currently being controlled.

Command Window Class The command menu class gives the user access to buttons for controlling features that would not work as keyboard or mouse controls. This allows the game data to be separated into multiple parts of the user interface. This class also gives the only access to quiting out of SAI-BOTS.

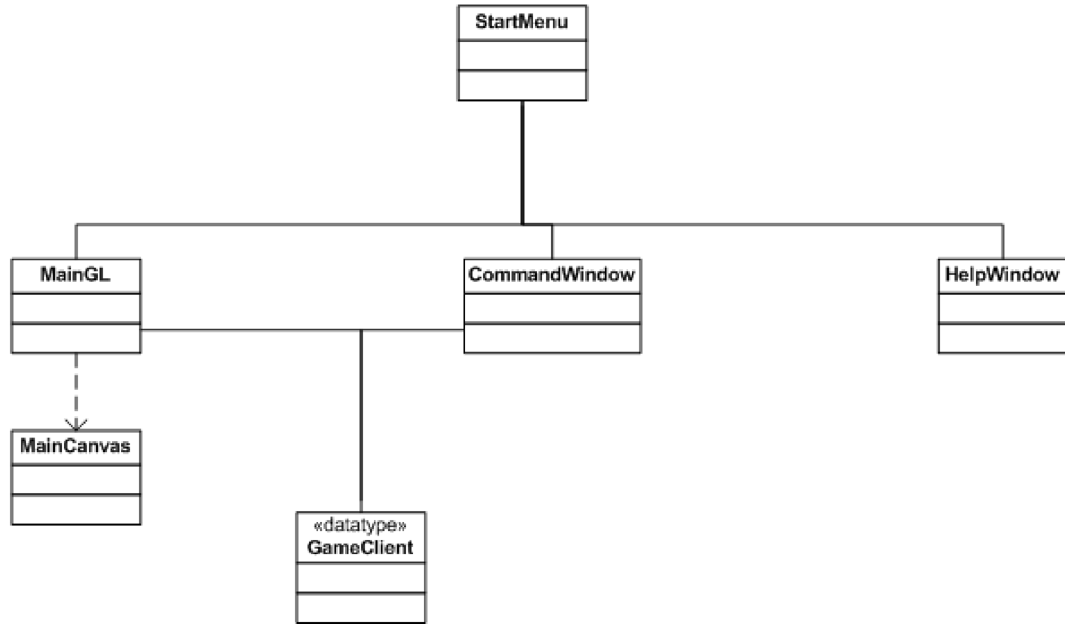

Figure 4.3: Graphical User Interface Class Heirarchy

Help Window Class The help menu is here for reference for the user to see what functions are available to them for use with the vehicles. Due to different levels of functions, this menu is split up into multiple different tabs.

Start Menu Class The start menu class creates the menu that comes up when the user starts SAI-BOTS. Within this menu the user is able to choose single or multi-player and then if he/she wants to go through a maze or to combat some vehicles controlled by the computer. This class then leads to the other two graphical user interface classes.

Base Classes

Figure 4.4 shows the class hierarchy of the base classes. Base classes are split up into multiple classes: cannon, camera, frame buffer, frustum, game client, image, lattice, model3d, quaternions, shader, tank, terrain, texture, triangle, vehicle, vertex buffer object, and weapon. This main system of classes is used to manage the simulation data. It is meant to be modular so that anything can be replaced.

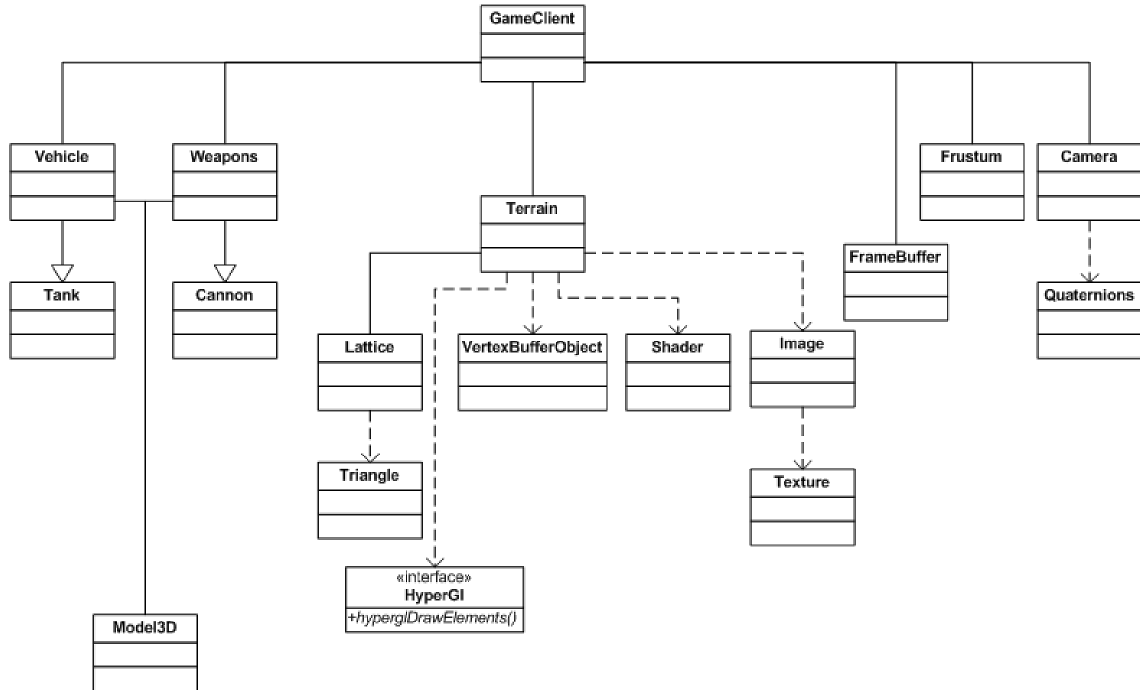

Figure 4.4: Base Class Heirarchy

Cannon Class Cannon specific data is stored in this class. The model to load and the speed of the weapon show dictate it's physical appears when rendered andt's movement speed.

Camera Class The camera class allows for us to use one camera for the entire game. If you are in commander mode you can fly around and get a birds-eye view of the world. If you are controlling a vehicle, the camera will stay behind the vehicle facing the way the vehicle is looking. In order to deal with gimbal lock in the camera, we use quaternions for all the rotations.

Frame Buffer Object Class This class builds a texture out of data that is passed to it. We use this to build a shader based heightmap that allows for the terrain to be created with less vertices stored.

Frustum Class The frustum class deals completely with culling out parts of the world that are currently not visible due to the current location, orientation of the camera, and obstruction of view. The terrain, vehicles, and weapons are dependent on this class.

Game Client Class This class holds the terrain data, all vehicles, the scripts for the vehicles, and how to render the world. The world is updated through this class, so if the user is in blind mode or has a script running a vehicle is provided with all of the relevent information for walls and vehicles around it. The game client is a very simple data manager, similar to a scenegraph.

HyperGL Interface This is a Python wrapper of the C++ glDrawElements for the terrain to use.

Image Class This is a class that reads in an image file and then creates a texture from that data. Uses Python Imaging Library [1] to read in the image.

Lattice Class This class reads in a dem [41] file and builds a triangle mesh of the terrain. The triangle mesh uses the triangle class and also deals with the intersection of the terrain.

Model3D Class The model3D class reads in an OBJ model file. Along with this file the class has two classes within it to store the materials and face groups of the model. This class uses the vertex buffer object class that we made to make the building and rendering of the model simpler.

Quaternions Class This is a simple quaternion class to help the camera not gimbal lock.

Shader Class This class takes in GLSL vertex and fragment shader files and holds the valid data for the shader to work properly. Multiple files can be added to a single

shader, which it then compiles and links into a shader program. After the program is built, we can then set specific variables that the shader is expecting and run the shader when needed.

Tank Class Tank specific information is in this class. It loads the tank specific model as well as how it should move and interact with the terrain. The weapons the tank has and its sensors are also stored.

Terrain Class The terrain class takes in a lattice data type and stores all the terrain data needed for the simulation. With the lattice data stored, the terrain then loads a shader and image to make the terrain not look flat. Even though the lattice data only stores the height value, the terrain class stores only the x and z values of the terrain. With the shader and using a frame buffer object the terrain uses a rendered version of the heights to determine how the terrain should look. It interacts with the vehicles and weapons.

Texture Class The texture class puts all the features of adding a texture and binding it in one place. This gets rid of a lot of duplicate code and makes the textures simpler to implement.

Triangle Class A very simple triangle class that holds the three vertices of the triangle. The other feature of this class is the intersection it has to determine the height at a specific triangle.

Vehicle Class The base information for all vehicles is stored here. Different vehicle types have their own individual data as well. It will load the particular model as well as set the way the vehicle will move. Other data held in this class are the weapons the vehicle has, its sensors, and if it gets destroyed what its explosion radius is.

Vertex Buffer Object Class Wraps the OpenGL vertex buffer object code into a single class. Initialization, building, and binding of the vertex buffer object all

happens here. This helps to make debugging vertex buffer objects easier and cleans up the code.

Weapon Class All available weapon options are stored in here. This data tells the speed of each weapon, how much damage each does, as well as how much the physics should tell the terrain to deform.

Future Systems

Future Systems will all be covered in Section 6.2.

4.3.2 User Interface

Designing a user interface has many facets. The first stage of designing a user interface is to follow the Functional Requirements that were presented in Table 4.1. Through the use of the wxWindows [51], prototyping the user interface was simple and allowed multiple variations to be tested.

These variations lead us to the User Interface Requirements presented in Table 4.3 We worked on getting a design that new students would be able to use when they are first shown SAI-BOTS. Since we knew that not all of the new students would have the same level of computer knowledge we had to decide how simple or complex to design the user interface. We came up with the minimal number of buttons needed for a user to be able to do the basic commands. There was also the idea of allowing the user to see what commands are available to be used.

This design ended up with three primary windows while the user is playing and a fourth window for when the game starts. The main GL window is where the actual rendering of the graphics happen. All of the buttons the user will be using are in the command window. The user will see all of the script commands in the help window. Finally, the start menu comes up when the user starts the game. Snapshots of the user interface developed are shown in the next chapter of this thesis.

SAI-BOTS shall have the ability to load a script.
SAI-BOTS shall have the ability to open the script.
SAI-BOTS shall have the ability to run the script.
SAI-BOTS shall have the ability to reload the script.
SAI-BOTS shall have the ability to choose an editor.
SAI-BOTS shall have the ability to choose a vehicle.
SAI-BOTS shall have the ability to become a commander.
SAI-BOTS shall have the ability to view only sensors.
SAI-BOTS shall have the ability to quit.
SAI-BOTS shall have the ability to show filename of the script.
SAI-BOTS shall have the ability to show the script commands.
SAI-BOTS shall have the ability to render the graphics.

Table 4.3: User Interface Requirements

Chapter 5

User Manual

5.1 System Activity

Figure 5.1 shows the different routes users can take running SAI-BOTS. The first route is to play a client game, while the second route starts a dedicated server.

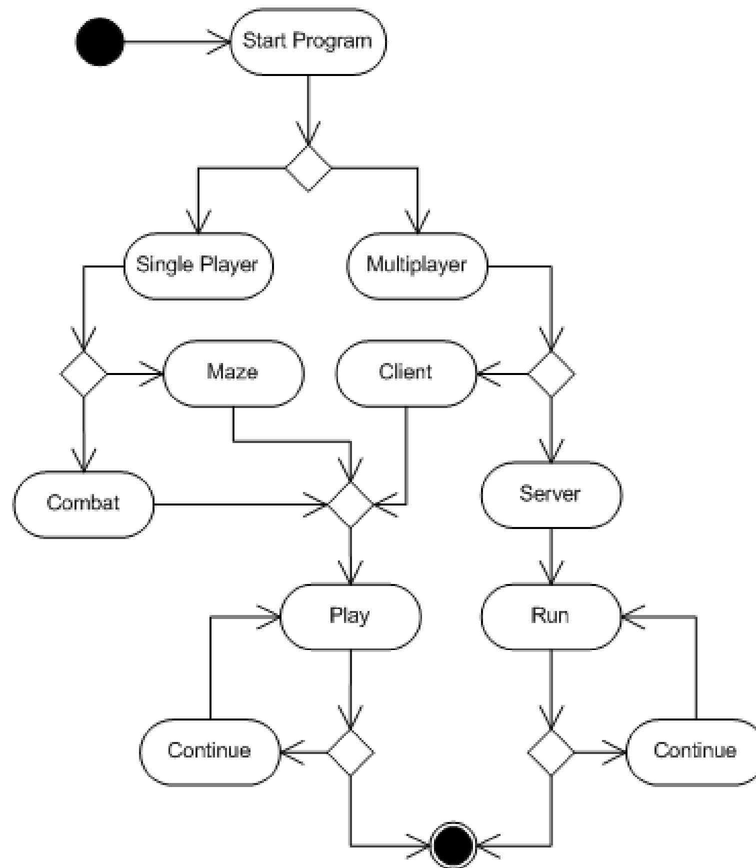

Figure 5.1: System Activity

5.2 System Requirements

The system requirements for SAI-BOTS are shown in Table 5.1. Since SAI-BOTS is using Python, there are software requirements for SAI-BOTS that are shown in Table 5.2.

Operating System	Windows XP and Linux
Processor	2.0 GHz or Greater
RAM	512MB
Video Card	GeForce 6600
OpenGl Version	1.5 or Greater
Hard Drive Space	500MB

Table 5.1: System Requirements

Python 2.5	[19]
Python Imaging Library 1.1.6	[1]
Python OpenGL 3.0	[38]
Python GLEW	[36]
Python wxWindows 2.8	[18]

Table 5.2: Python Requirements

5.3 Starting

After the user has made sure SAI-BOTS will run on his/her computer and downloaded it, he/she should double-click the SAI-BOTS installer. This whole process will take care of the required packages even if they are not installed on his/her system.

The user can either start SAI-BOTS by double-clicking the executable for SAI-BOTS or opening up a command prompt and starting it from there. Once SAI-BOTS is started, in Windows there should be a screen that looks like Figure 5.2 and Linux should look like Figure 5.3.

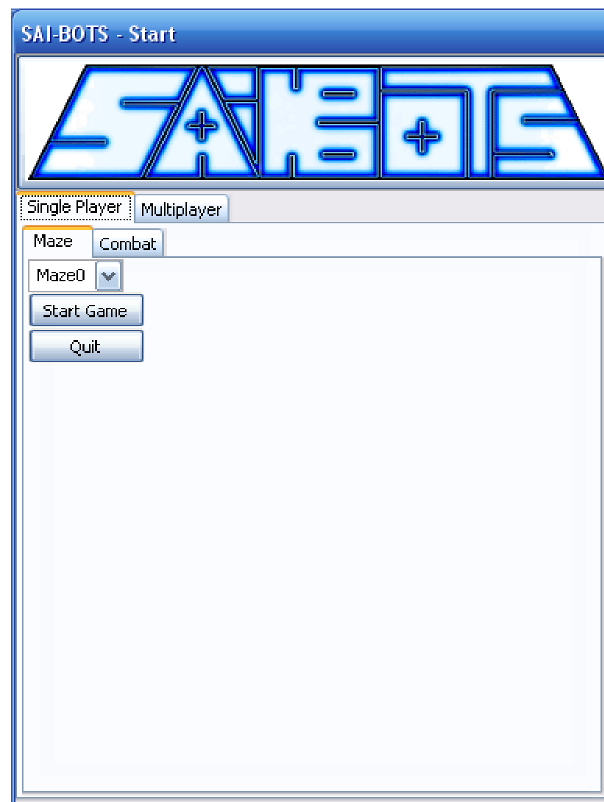

Figure 5.2: Windows Startup Screen

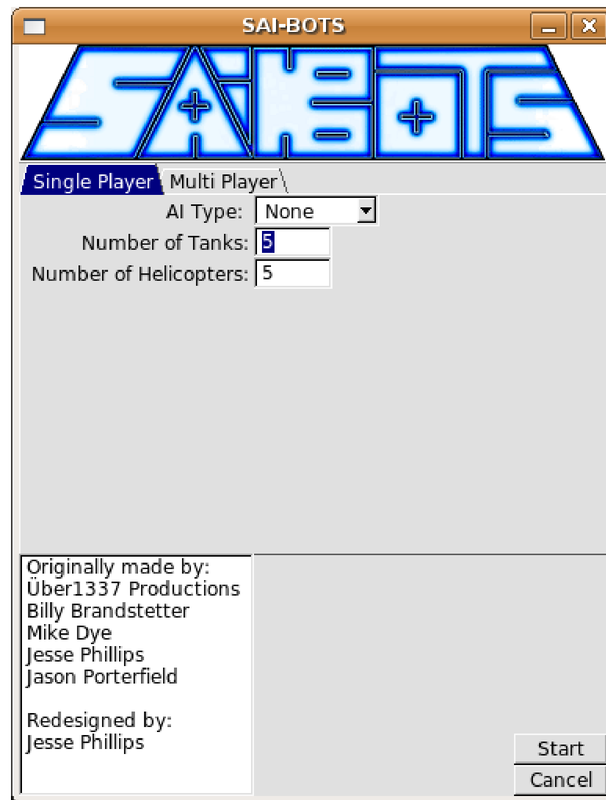

Figure 5.3: Linux Startup Screen

After SAI-BOTS has been started the user can choose whether to play a single player game or a multiplayer game.

5.4 Single Player

Starting a single player game has very few steps. There are two different modes for players to choose from. Figure 5.4 shows how to start a single player game to do pathfinding through a maze. The player can choose which maze to go through by selecting the maze from the drop down box surrounded by the red box. After selecting the maze, the player then clicks the Start Game button to be dropped into the world with the maze and a single tank.

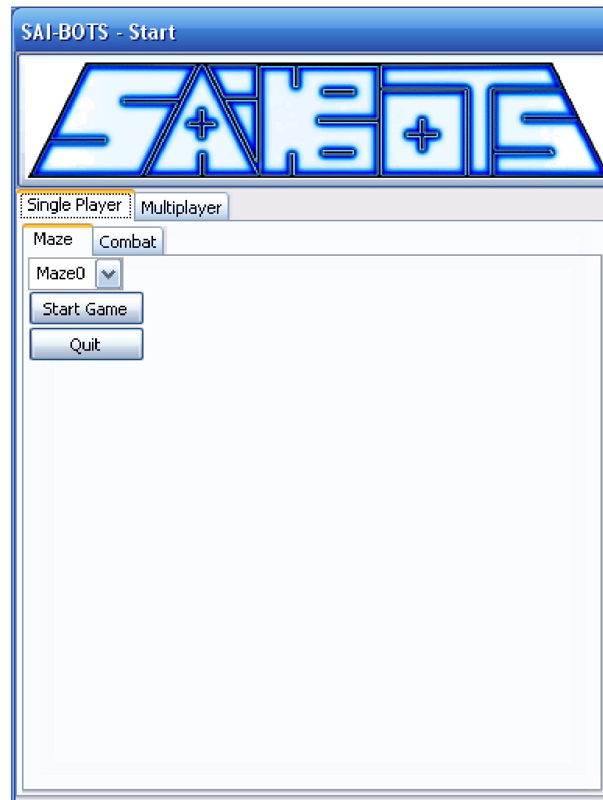

Figure 5.4: Single Player Start - Maze

The other option for players to choose is called combat and is shown in Figure 5.5. Some more advanced options can be chosen for single player combat, the first being if

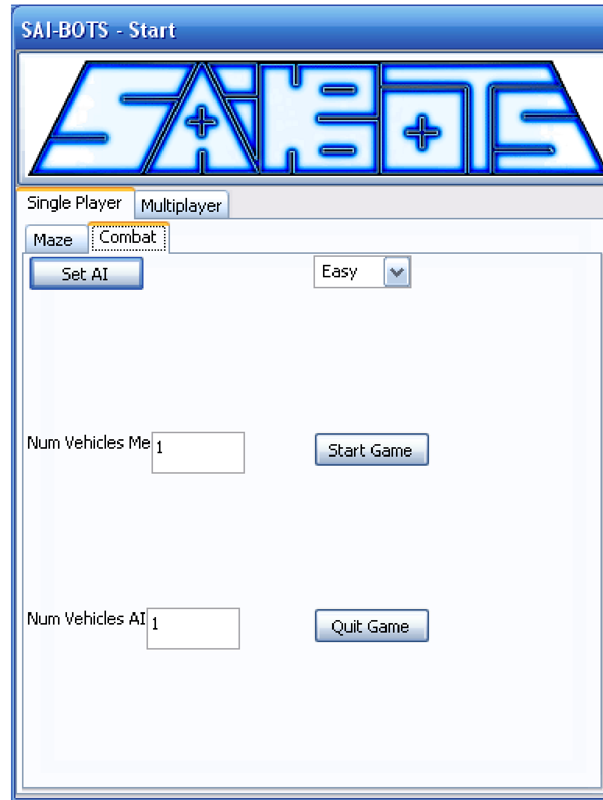

Figure 5.5: Single Player Start - Combat

one wants artificial intelligence opponents or not, designated by the area surrounded by a yellow box. If the Set AI toggle button is selected the game will also take into account the selection in the red box of what type of AI: easy, medium or hard. These options just have different scripts that the AI uses. The last two options for single player are how many vehicles to set for both the player and the AI, the default for each is set to one and is shown by the blue and green boxes, respectively.

5.4.1 Vehicle Control

At the beginning of the game the player starts off in control of the vehicle he/she is on. The command menu, shown in Figure 5.6, allows for the player to choose which vehicle to control.

Table 5.3 covers all the buttons that are shown in Figure 5.6 available to the players.

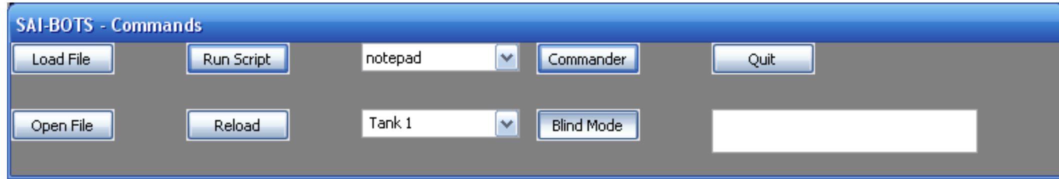

Figure 5.6: Command Menu

Load File	Load a Python file to control vehicle
Open File	Open Python file with selected editor
Run Script	Have Python file control vehicle
Reload Script	Reload the script file that has already been loaded
Blind Mode	Switch graphics to blind mode
Commander	Fly around the world as an "Eagle-Eye" camera
Editor	Which editor to open scripts in
Filename	Vehicles current file

Table 5.3: User Interface - Buttons

In the vehicle class all vehicles can be controlled through the same keys. The basic controls are show in Table 5.4.

Table 5.5 covers all of the editors that players can edit their scripts in. Of course if a player does not like one of these editors they can still open the script in their preferred editor and use that editor while the game is running.

The help window shown in Figure 5.7 has a set of tabs, one for each of the different scripting functions which are covered in Section 5.4.2 and Section 5.4.5.

W	Speed Up
S	Slow Down
A	Turn Left
D	Turn Right
Left Mouse Button	Fire Weapon
Right Mouse Button	Modifies what is done with the mouse movement
Move Mouse Up	Look Up
Move Mouse Down	Look Down
Move Mouse Right	Look Right
Move Mouse Left	Look Left

Table 5.4: Vehicle - Movement Controls

Editor	Operating System	Where to get
Notepad (Windows Default)	Windows	Comes installed
Notepad++	Windows	[16]
UltraEdit	Windows	[29]
gvim	Windows	[37]
gvim (Linux Default)	Linux	Package managers
xemacs	Linux	Package managers
gedit	Linux	Package managers
kedit	Linux	Package managers

Table 5.5: Editors

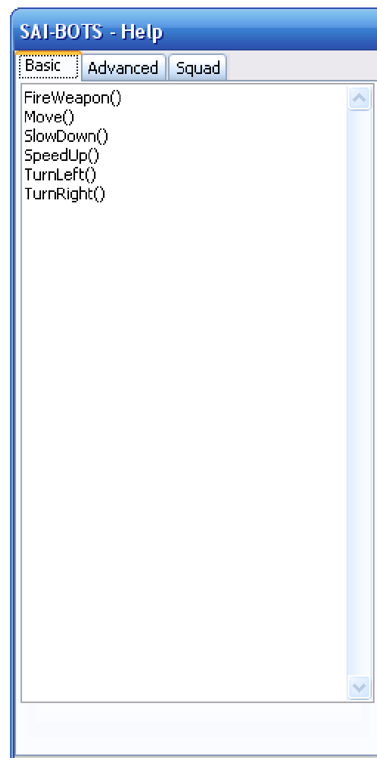

Figure 5.7: Help Menu

Figures 5.8 and 5.9 show a tank moving around the world getting closer to the enemy. Figure 5.10 shows what happens when the user or the script turns a tank to the left. Figure 5.11 shows what happens when the user or the script turns a tank to the right. Figure 5.12 shows what happens when the user or the script looks down, both the camera and the barrel of the tank move. Figure 5.13 shows what happens when the user or the script looks left, the camera, turret, and barrel all move. Figure 5.14 shows what happens when the user or the script looks right, the camera, turret, and barrel all move. Figure 5.15 shows what happens when the user or the script looks up, both the camera and the barrel of the tank move. Figure 5.16 shows what happens when the user or the script fires the weapon.

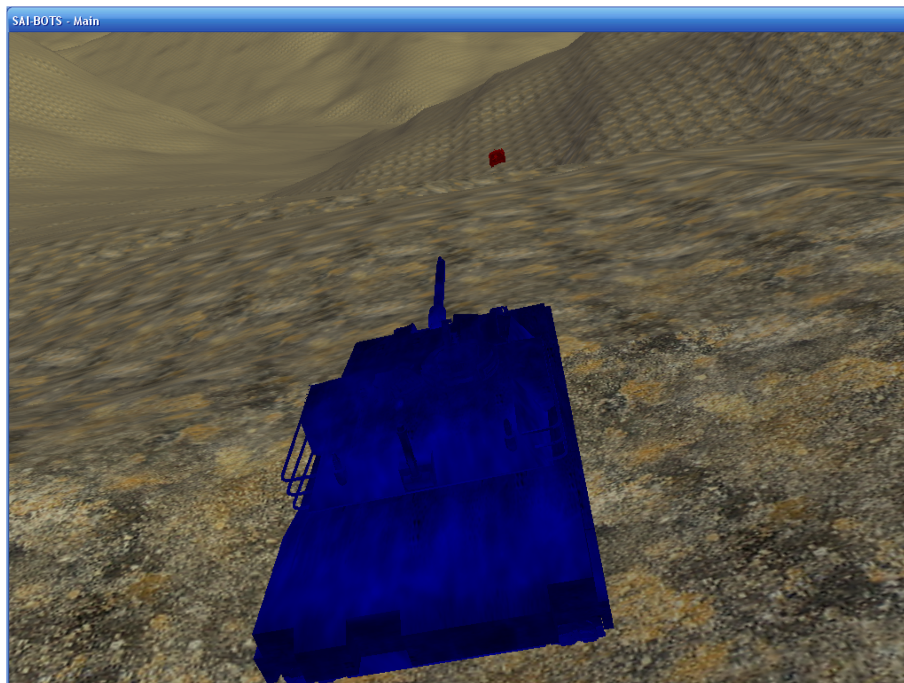

Figure 5.8: Tank Moving Forward

5.4.2 Script Editing

As the primary feature of SAI-BOTS, editing scripts while playing games is very important. Table 5.6 covers the beginner scripting commands limited to getting a vehicle to move around and to fire its weapon. Section 5.4.1 talked about the basics

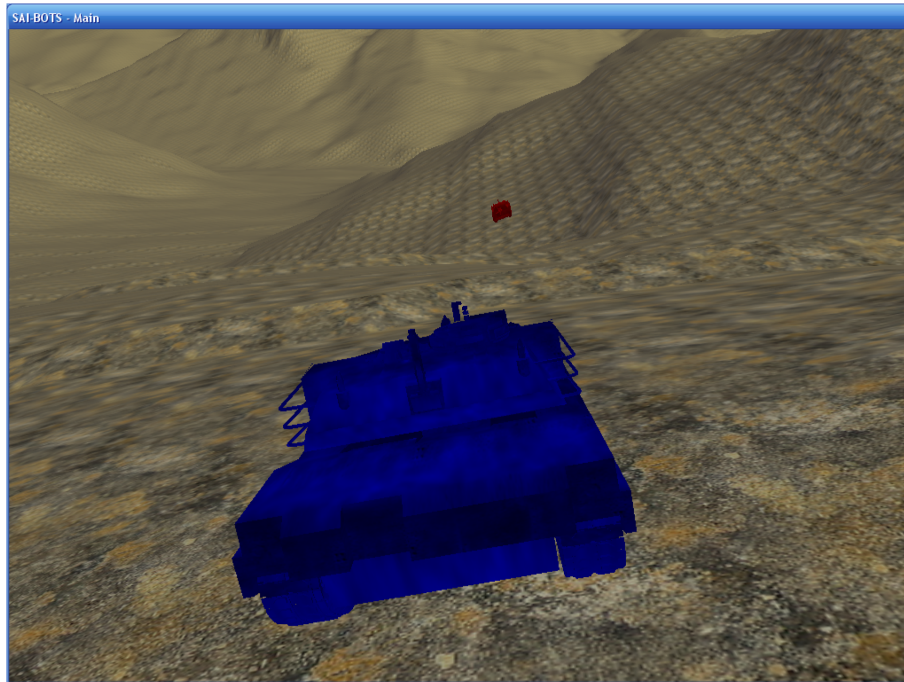

Figure 5.9: Tank Moving Forward Second

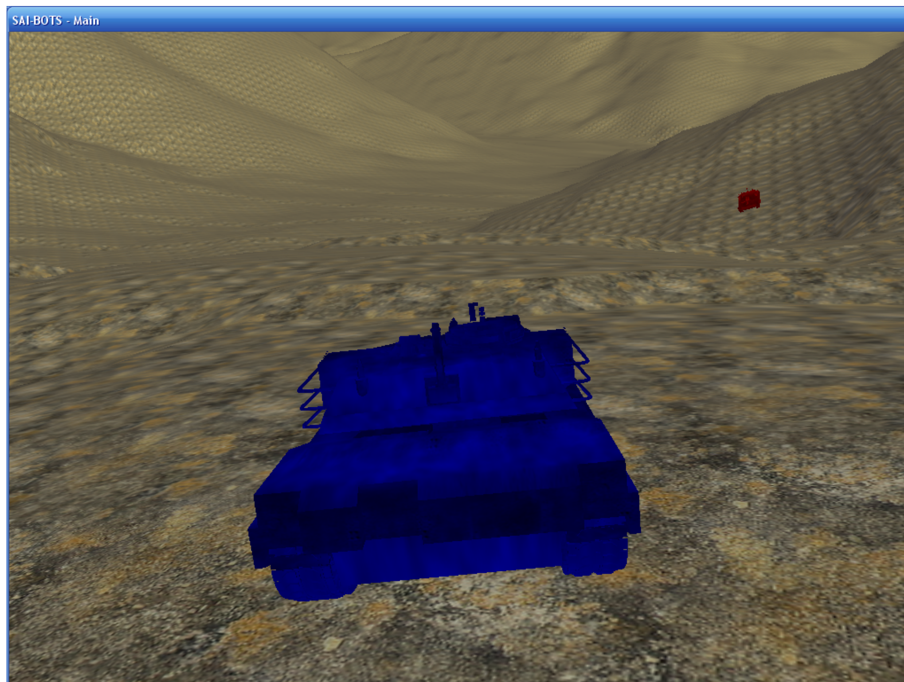

Figure 5.10: Tank Turning Left

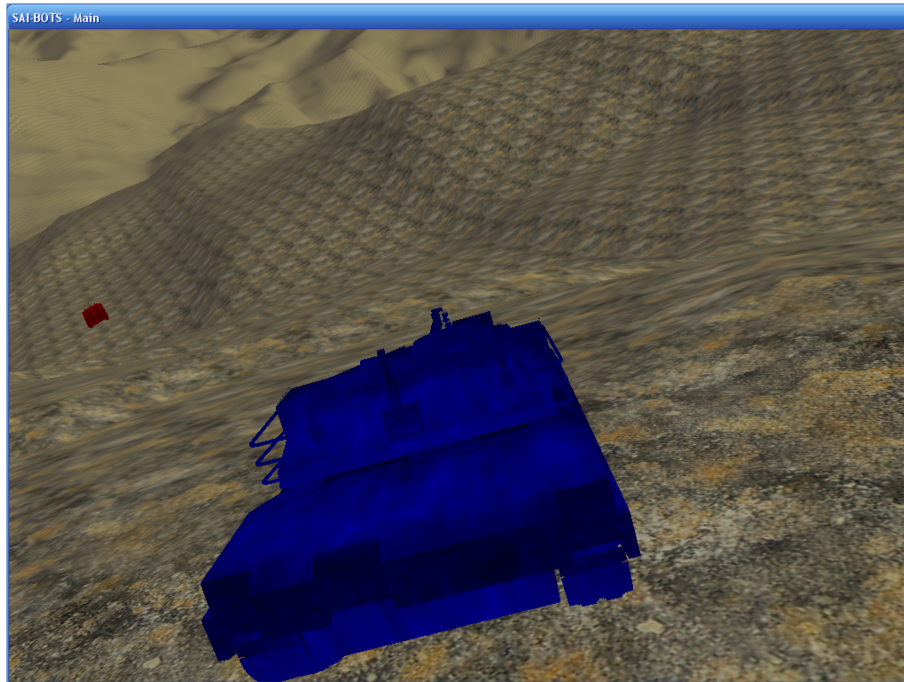

Figure 5.11: Tank Turning Right

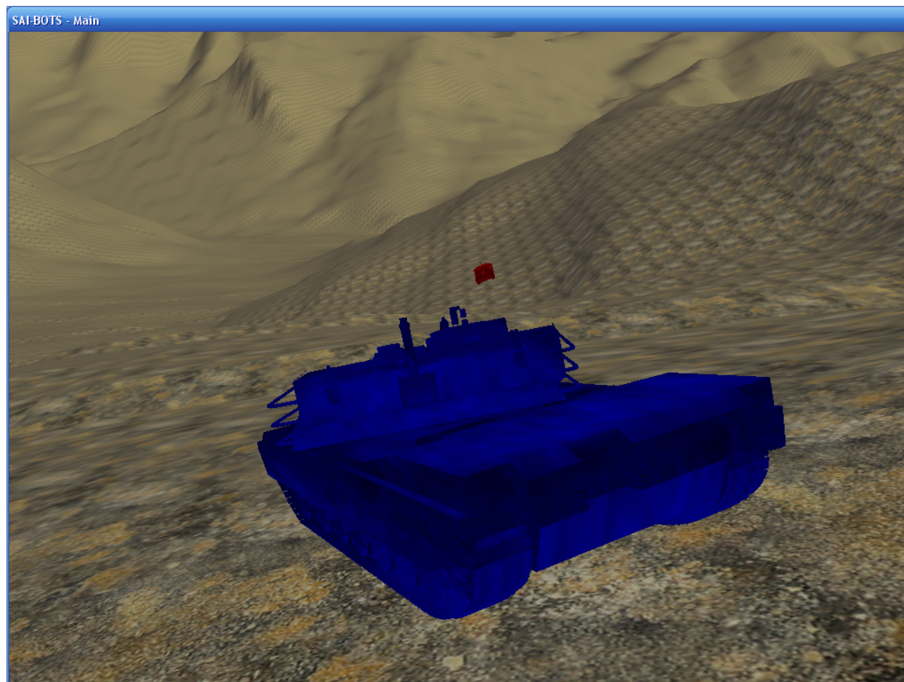

Figure 5.12: Tank Looking Down

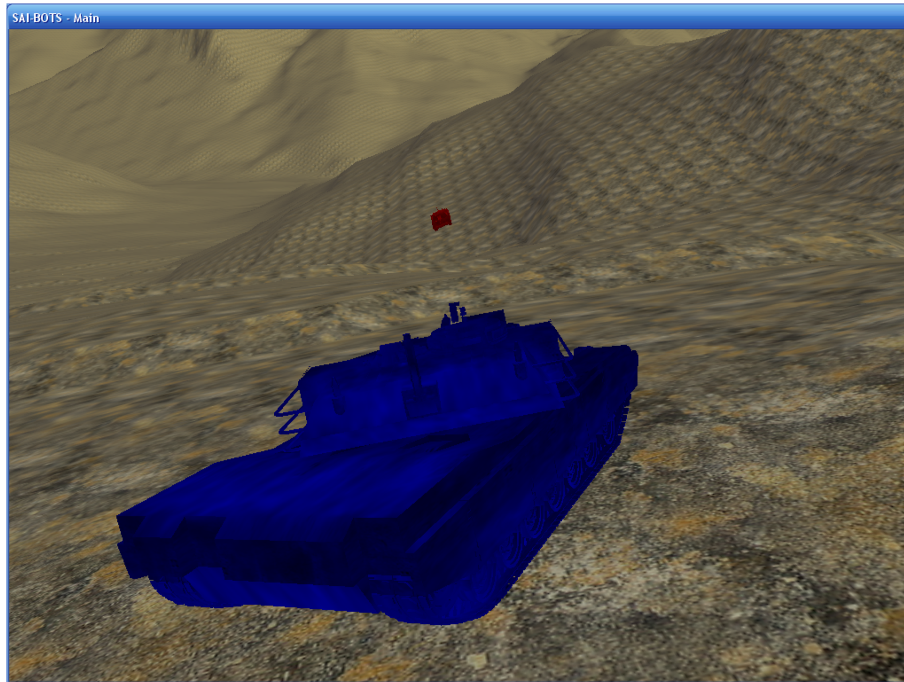

Figure 5.13: Tank Looking Left

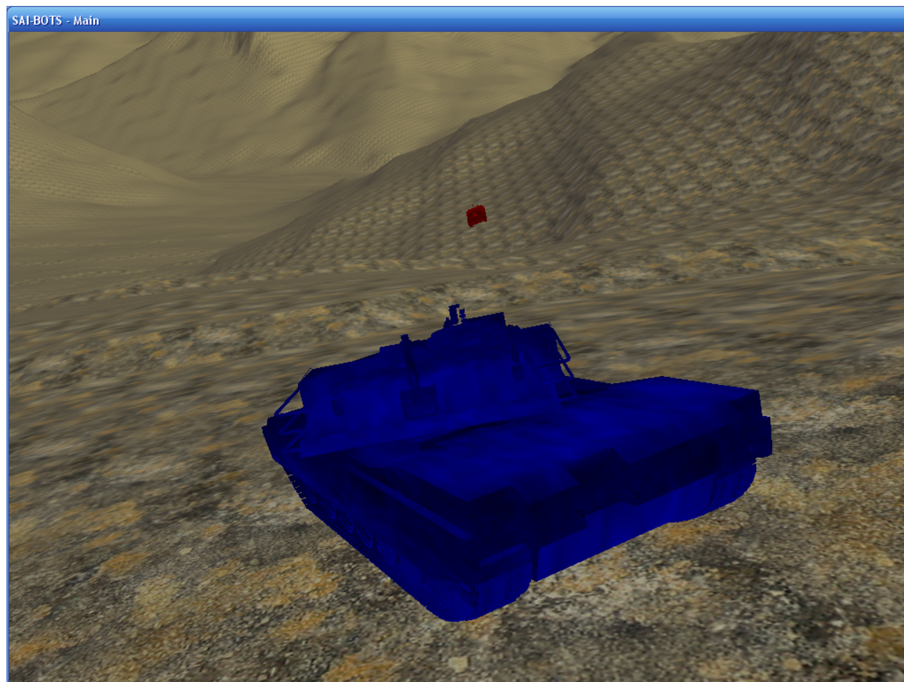

Figure 5.14: Tank Looking Right

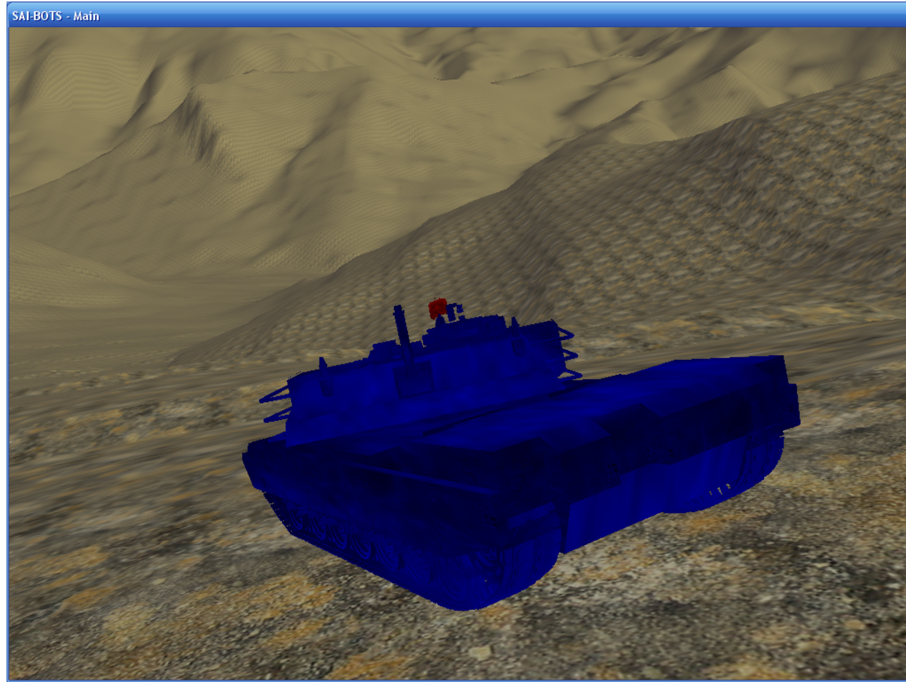

Figure 5.15: Tank Looking Up

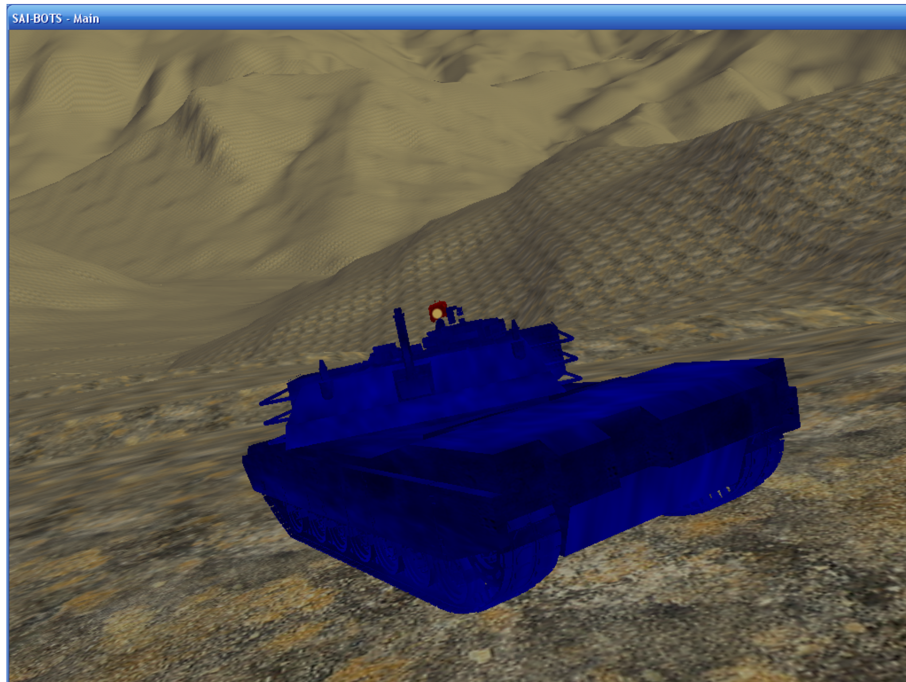

Figure 5.16: Tank Firing

Command	What it does
FireWeapon()	Fires the current weapon
Move()	Enacts movement commands stated
SlowDown()	Decrease the vehicles speed
SpeedUp()	Increases the vehicles speed
Stop()	Stops the vehicle
TurnLeft()	Turn the vehicle left
TurnRight()	Turn the vehicle right

Table 5.6: Python Commands - Beginner

of the user interface. After clicking the Open File button something similar to the following figures will show up,

Figure 5.17 shows a script that is only using the basic commands being edited.

Table 5.7 covers more advanced scripting commands, and ways to better control the vehicles. Figure 5.18 shows a script with more advanced commands being edited.

Command	What it does
GetPosition()	Get current position
LookDown()	Look down
LookLeft()	Look to the left
LookRight()	Look to the right
LookUp()	Look up
Sensors()	Get the active state of the sensors

Table 5.7: Python Commands - Advanced

5.4.3 Blind Mode

Figure 5.19 shows the button in the command menu that starts Blind Mode. Clicking this makes the 3D graphics window completely black and renders the radar graphics instead. This adds a complexity to writing scripts, but will also make it easier as this view is exactly how the computer sees the world.

Figure 5.20 shows what the radar looks like with allies around the current vehicle. Figure 5.21 shows what the radar looks like with allies around the current vehicle. Figure 5.22 shows what the radar looks like in the maze mode of single player with

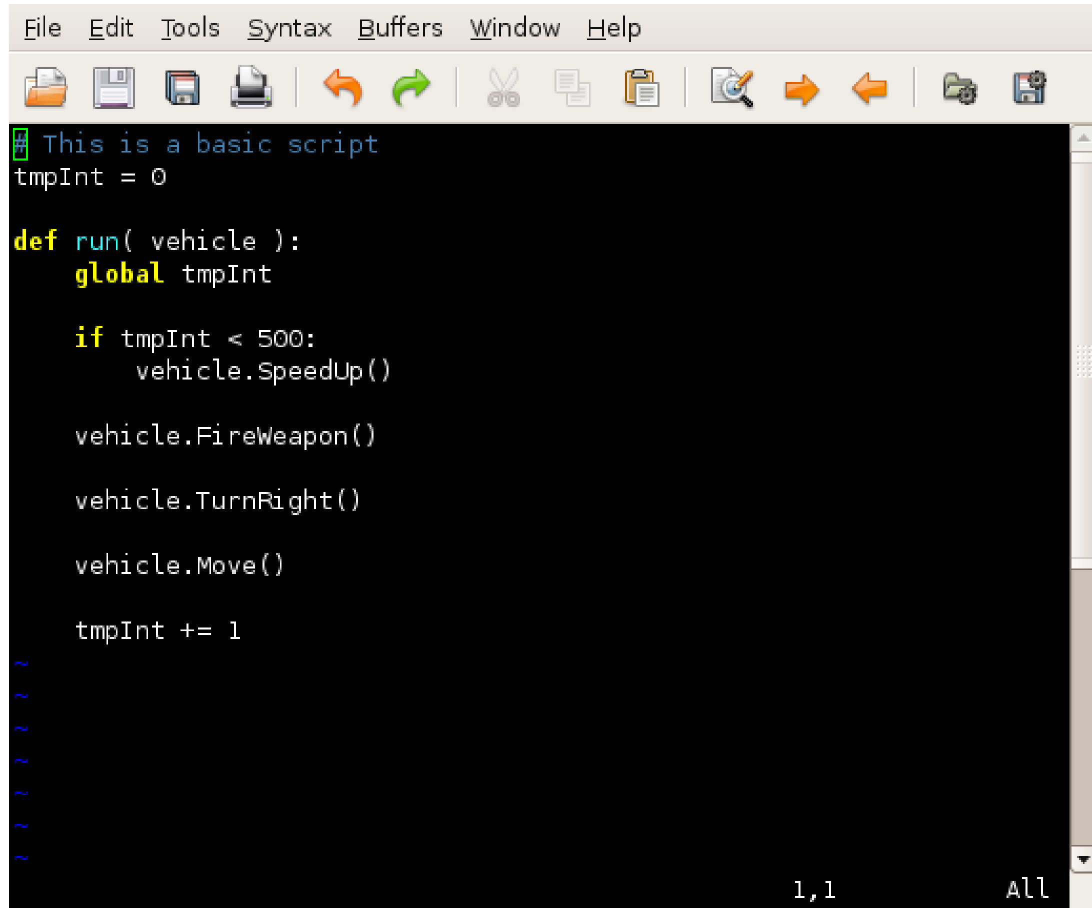

Figure 5.17: Basic Script Editing

File Edit Tools Syntax Buffers Window Help

```
# This is a Maze Pathfind script

def run( vehicle ):
    type, data = vehicle.Sensors()

    if type== "Walls":
        # Forward Sensor
        if data[0] > 40:
            vehicle.SpeedUp()
        elif data[0] < 10:
            vehicle.Stop()
            if data[2] > 10:
                vehicle.TurnRight()
            else:
                vehicle.TurnLeft()

            break
        else:
            vehicle.SlowDown()

        # Side Sensors
        if data[2] < 10 and data[6] > 10:
            vehicle.TurnLeft()
        elif data[6] < 10 and data[2] > 10:
            vehicle.TurnRight()

        vehicle.Move()
~
~
                                                    27,22          All
```

Figure 5.18: Advanced Script Editing

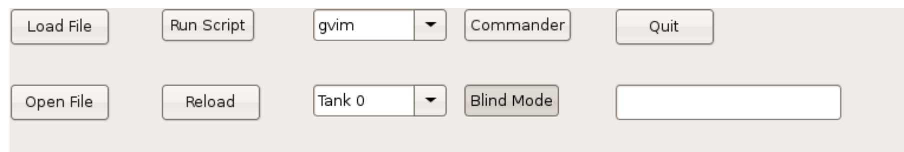

Figure 5.19: Starting Blind Mode

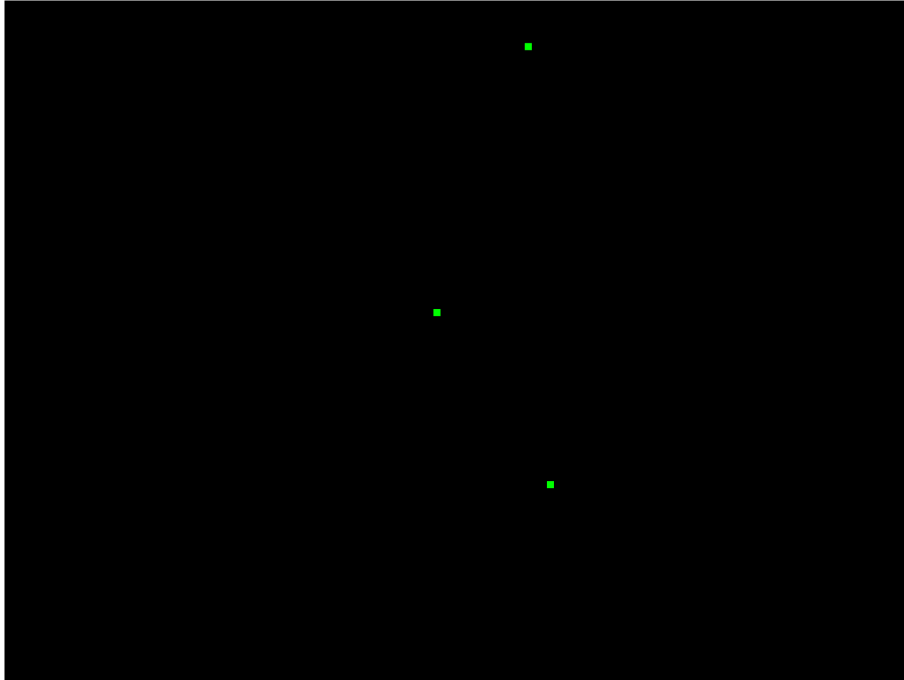

Figure 5.20: Allies Around

walls near the vehicle.

5.4.4 Mazes

The current set of mazes in SAI-BOTS were created with the Maze Maker [31] website. These mazes come out as GIF image files and look like Figure 5.23. In order to turn the image files into the DEM format that SAI-BOTS reads the user will have to use another package we made.

Figure 5.24 shows what the previous maze looks like inside of SAI-BOTS. The SAI-BOTS terrain converter is a simple wxPython application just like SAI-BOTS, but instead is just a user interface where the user can select files to convert from any file type to a DEM.

After the terrain is converted the user will have to place the new maze in to the data/dems directory of SAI-BOTS home. Then the user will also have to edit the mazes.txt file inside of the data directory following the guidelines shown in Table 5.8.

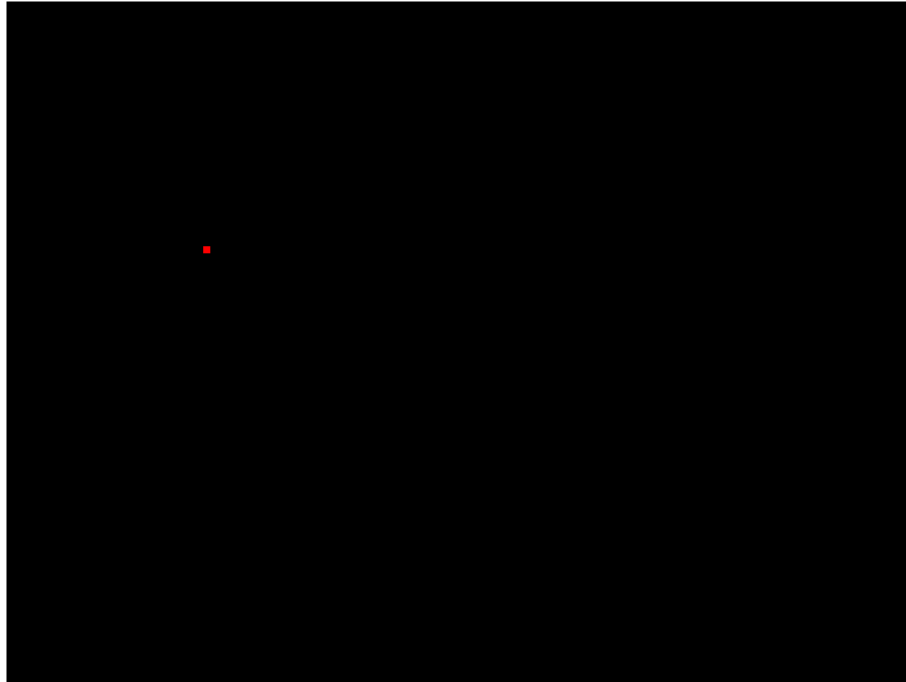

Figure 5.21: Enemies Around

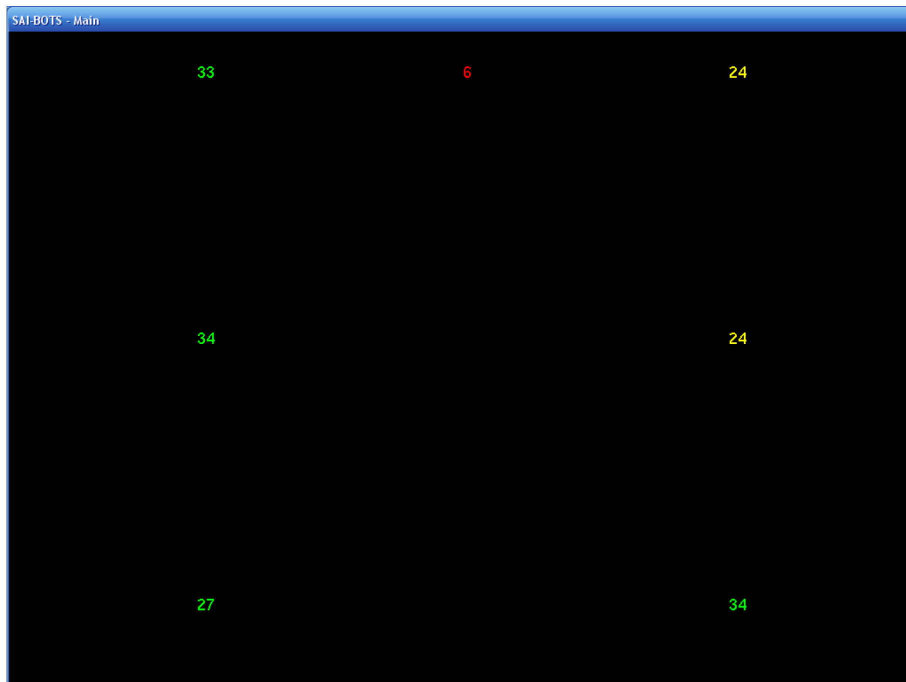

Figure 5.22: Sensors

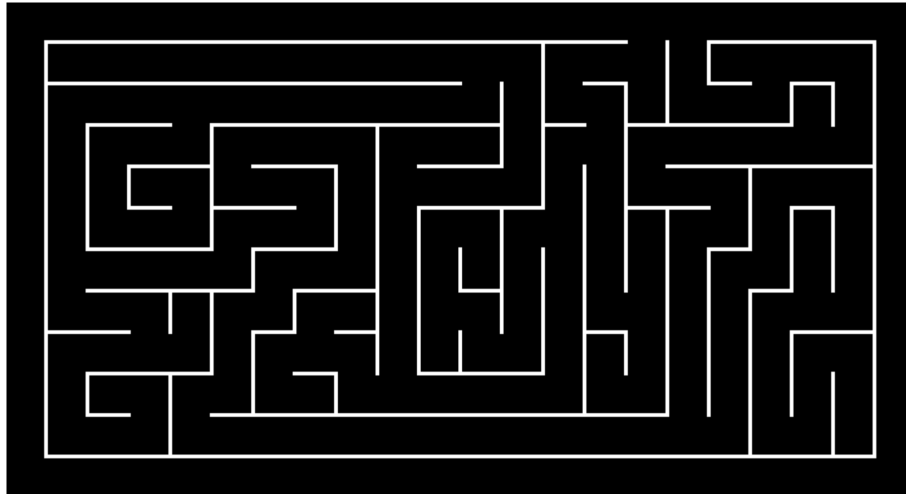

Figure 5.23: Maze GIF Image

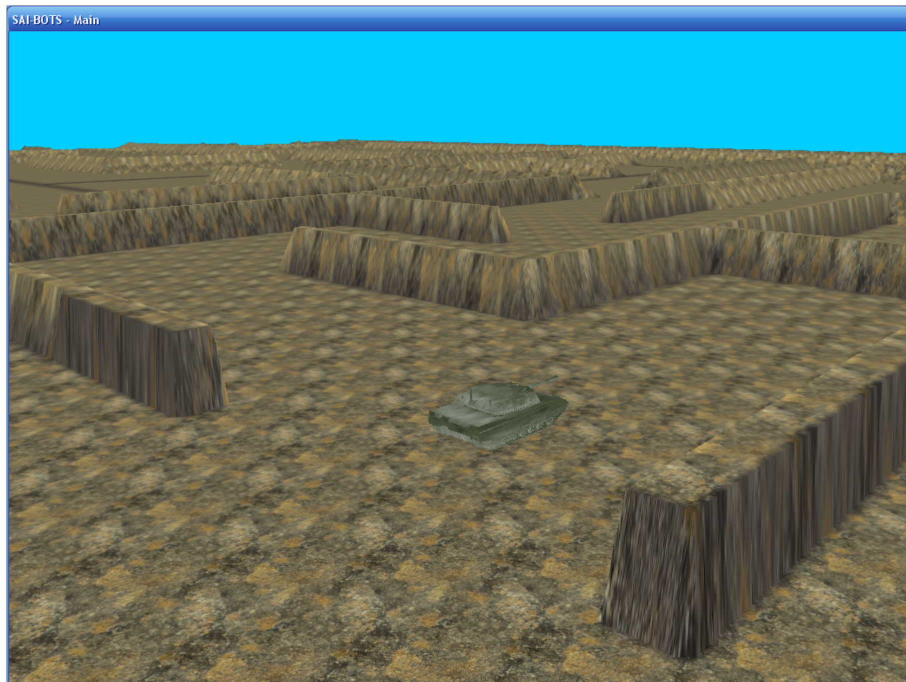

Figure 5.24: Game Maze

Title	What it does
Name	Name of the maze
filename	actual filename of the maze
(startX,startZ)	Start position for the vehicle
(endX,endZ)	End position for the vehicle
startFacing	Direction to start off facing

Table 5.8: mazes.txt File Format

5.4.5 Squad Control

The following Sections cover how the squad scripting commands work. First we will cover sending commands to other units, then we will cover receiving commands.

Sending Commands

There are many ways to communicate with the different units on the battlefield. The sending commands are shown in Table 5.9. In order to send a message that only allies can hear, players can choose whether to send the message to the entire team or just send it to a channel recieved only by other units listening to that channel. While you can send messages just to your team, you can also send messages to all units around you, including enemies.

Having the two different ways of sending commands to allies helps to make squad scripting simple, but also gives it the ability to be more complex than current scripting situations. Figure 5.25 shows what a script looks like with sending commands to other units.

Command	What it does
SendTeam(string)	Send <*string*> to entire team
SendChannel(string, int)	Send <*string*> to channel <*int*>
Say(string)	Say <*string*> for all players near by to hear
Yell(string)	Say <*string*> for all players within 100 units to hear

Table 5.9: Python Commands - Squad Send

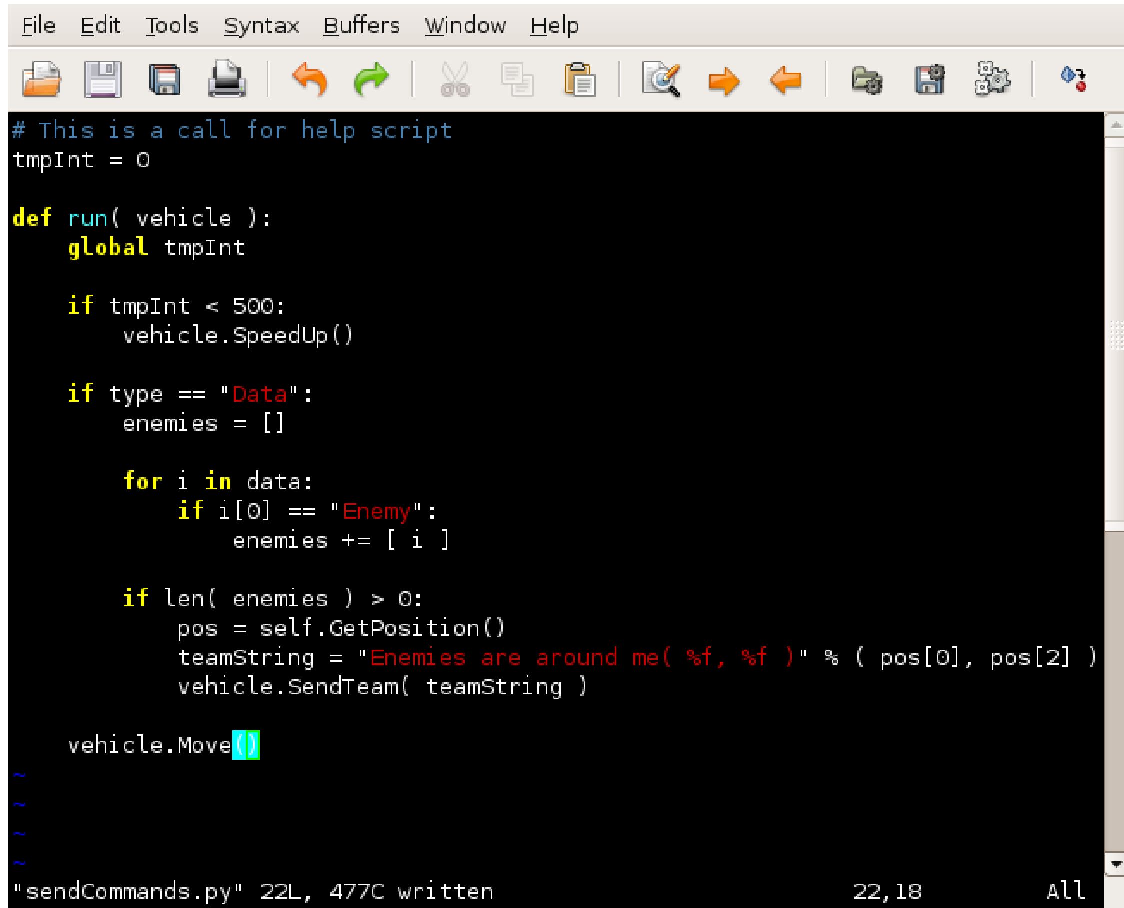

Figure 5.25: Squad Script - Sending

Receiving Commands

There are many different ways to get messages sent by different units. The receiving commands are shown in Table 5.10.

The first way gets all messages sent from your team, on all channels and even listening for messages around you. This way of receiving commands is the most complex as then the user has to parse through every single message. Then the player has two options to choose from for listening for messages from their teammates. The first option is to listen just for commands sent to the entire team, while the second option allows the player to listen on a specific channel for commands. The final way for players to receive commands is to listen for commands sent out from within a set vicinity. Figure 5.26 shows what a script looks like with receiving commands from other units.

Command	What it does
ReceiveAll()	Listen for all messages
ReceiveTeam()	Listen for messages sent to team
ReceiveChannel(int)	Listen for messages on channel $<int>$
Hear()	Listen for messages from Say or Yell

Table 5.10: Python Commands - Squad Receive

5.5 Multi-Player

After choosing to play a game in multi-player mode the user has the ability to choose whether he/she wants to start a dedicated server, start a server with a client, or just a client. The dedicated server allows for the user to have a seperate computer host a game while every other computer can just worry about rendering the world. A server with client starts a server and client both on the same computer, this has the possibility to slow down the game on that computer. Finally the client mode just starts up the graphical user interface for the user to interact with, but the simulation of the game is not run through the client.

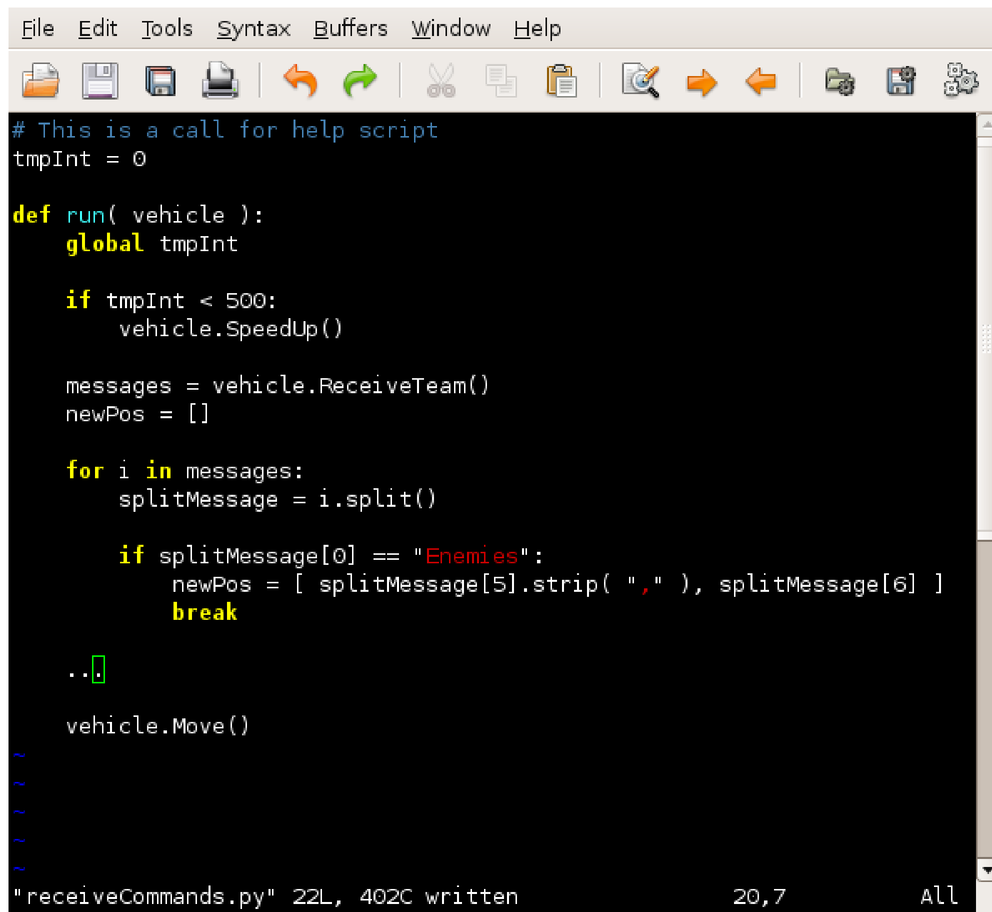

Figure 5.26: Squad Script - Receiving

Along with choosing client type the user then has many more options to choose from. For the dedicated server and server with client models the user sets how many players, how many vehicles they have, and what IP address the other clients should connect to. The client model only has two options to fill in, the color they want to use for their vehicles and what IP address to connect to.

Chapter 6

Conclusions and Future Work

6.1 Summary and Conclusions

Due to the success and entertainment value of video games in todays society, it makes sense to try and use a video game to help teach students concepts and skills in computer science. Once we decided what it was that we wanted to help teach, we started coming up with ideas for implementing. In this thesis we showed all the elements in current video game engines and video games and we also showed the hole in current video games that we wanted to fill.

Computer Science 105, at the University of Nevada, Reno, uses many tools to help reinforce what students learn in class. The course uses the MIPS Interpreter SPIM [30] to show how everything works in the background of the machine. In order to show students a graphical flowchart of how a program runs, the teacher uses RAPTOR [10]. Finally, to allow students to make some non-interactive videos through scripting with Alice [47].

SAI-BOTS was proposed as a reinforcement teaching tool for both beginning programming and artificial intelligence classes. We showed some new ways to think about and how to implement the idea of multiple entities working together as a squad. Neuroevolution was shown to be a way for users then to implement even more advanced artificial intelligence techniques. Due to how artificial intelligence currently views a game world, we showed how you can give the artificial intelligence the same type of data but not all of the data. On the fly scripting was a primary feature of

SAI-BOTS and we showed how it can work and the benefits of using it for helping to reinforce lessons. As shown in previous work, a blind mode allows for a user to view the world as the scripts view the world. Finally we showed the two different types of terrain that the user can choose from to write their scripts.

Abstraction of all parts from the rendering mode to vehicles and weapons results in modular code. This modularity was built in so that new ideas, like the ones shown in Section 6.2, can be added to SAI-BOTS easily. The core elements of SAI-BOTS as described in Chapter 4 are now in place. SAI-BOTS has successfully accomplished the intended functions as show in Table 4.1. All use cases listed in Section 4.2 are completely operational.

The functionality, defined in Chapter 5, of how to run SAI-BOTS is designed to be simple enough that as long as the user knows how to use a computer he/she can run the program. The current implementation is stable and ready for use in classrooms as well as being ready for all the future work covered in the next Section. The solid foundation built with SAI-BOTS should make it a great tool for teachers to use to help reinforce what they covered in class.

6.2 Future Work

For ideas of where to start working on improving SAI-BOTS, future developers are highly encouraged to review and consider the recommendations in the following Sections. With the following features added, SAI-BOTS could be a more extensive teaching tool further down some advanced branches of artificial intelligence.

C/C++ with Python

A key issue with the current version of SAI-BOTS is that running solely in Python makes the graphics run at a slower speed. A way to speed up the graphics is to make the game engine in C/C++ rather than Python. In order to allow for scripting of the vehicles as well as the world, Python could still be integrated to allow for simple changes that would not need to be compiled after the game engine is written. The

speed gained from using C/C++ for the game engine permit larger more complicated terrains as well as many more vehicles. Furthermore, increased complexity adds to the learning goal.

Buildings

Adding buildings to SAI-BOTS was an idea in the first design, unfortunately it was a little more complicated to implement than first thought. The ability to have buildings in the world can add a more complex way of controlling the vehicles as the sensors would have to see buildings as well. This could make for an interesting challenge for teachers to give their students, where it has nothing to do with firing weapons.

Buildings could be added to the XML configuration files as well. The main problem is dealing with placement of the buildings in the sense that there are many options for placing buildings.

Efficient and Deformable Terrain

With the terrain already in a vertex buffer object it makes sense to then make the terrain more efficient and deformable. The idea of deforming the vehicles and buildings, when they are added, leads to the idea of sensors on vehicles being destroyable. A simple level of detail algorithm could add a lot of efficiency to the terrain as well as vehicles and buildings. This is an advanced feature that was not really seen as something required originally, but is more of an aesthetic feature.

Artificial Intelligence

Neuroevolution was one of the other goals of this thesis, unfortunately only the sensors that neuroevolution can use were implemented so the final implementation needs to be added. Some of the basics behind what is needed for this is the ability to run without graphics, neural networks to be able to control the vehicles, and a more advanced system for running specific simulations.

Virtual Reality

The ability for players to engage in combat against each other allows them to try and create the best possible script they can. This also brings up the idea of allowing outside viewers to watch matches between players so that they can see how each individual player does. To bring this into a virtual reality environment provides a venue for outside users to use stereoscopic glasses to watch a game run allowing the three-dimensional world to pop out at the users.

Another potential feature is a three-dimensional user interface that would allow for the users to edit scripts on the fly inside of a virtual reality environment. The one option that we came up with is to make the scripting similar to the drag and drop boxes that Unreal Engine 3 has for script editing. The only difference between this and Unreal Engine 3 would be the ability to reload the scripts after they have been edited inside the virtual reality environment.

Assessment Tool

A teacher needs to be able to tell if what they are teaching the students is getting across and they are able to implement what is taught. For some assignments there are very few was to write code to do what is needed to be done, so the assessment tool will be able to tell the teacher how well the code is written and speed it runs at. The other assignments that have more complicated answers need to be able to be checked for copying. This would allow for a teacher to be able to tell if a student is asking for help and not going to the teacher or the teacher's assisstant. If a student is having trouble with a specific challenge,the student would even be able to see this themselves. The student would then be able to show the teacher's assisstant what they have and ask for help.

Usability Studies

In order to see the effectiveness of SAI-BOTS, a usability study would need to be run. Along with the assessment tool to make sure that students are writing better code

as well as not cheating, a usability study will show if current methods for teaching students are better. In order to run a usability study we would have to go through the Office of Human Research Protection [48] to make sure that our results are considered valid. This is a very simple step to be able to show how well SAI-BOTS helps to reinforce what students learn in class.

Bibliography

[1] Secret Labs AB. Python Imaging Library. http://www.pythonware.com/products/pil. Accessed March 10th, 2008.

[2] AGEIA. AGEIA. http://www.ageia.com/physx/. Accessed March 10th, 2008.

[3] ANNEvolve. ANNEvolve :: Evolution of Artificial Neural Networks. http://annevolve.sourceforge.net/. Accessed March 10th, 2008.

[4] Jim Arlow and Ila Neustadt. *UML and the Unified Process: Practical Object Oriented Analysis and Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[5] Inc Autodesk. Autodesk. http://usa.autodesk.com/. Accessed March 10th, 2008.

[6] Leonardo Boselli. APOCALYX. http://apocalyx.sourceforge.net/. Accessed March 10th, 2008.

[7] Leonardo Boselli. GUN-TACTYX. http://gameprog.it/hosted/guntactyx/. Accessed March 10th, 2008.

[8] Leonardo Boselli. JROBOTS - Main. http://jrobots.sourceforge.net/. Accessed March 10th, 2008.

[9] William E. Brandstetter, Michael P. Dye, Jesse D. Phillips, Jason C. Porterfield, Jr. Frederick C. Harris, and Brian T. Westphal. SAI-BOTS: Scripted Artificial Intelligent Basic On-Line Tank Simulator. In *In Proceedings of The 2005 International Conference on Software Engineering Research and Practice (SERP '05)*, volume II, pages 793–799, Las Vegas, NV, June 27-30 2005.

[10] Dr. Martin Carlisle. RAPTOR - Flowchart Iterpreter. http://www.usafa.af.mil/df/dfcs/bios/mcc_html/raptor.cfm. Accessed March 10th, 2008.

[11] W. Celes, L. Henrique de Figueiredo, and R. Ierusalimschy. The Programming Language Lua. http://www.lua.org/. Accessed March 10th, 2008.

[12] chUmbaLum sOft. chUmbaLum sOft. http://chumbalum.swissquake.ch/ms3d/index.html. Accessed March 10th, 2008.

[13] CRYTEK. CryEngine 2 and Sandbox 2 Tutorials. http://www.cryengine2.com/. Accessed March 10th, 2008.

[14] CRYTEK. Welcome to Crytek. http://www.crytek.com/. Accessed March 10th, 2008.

[15] E. Danovaro, L. De Floriani, E. Puppo, and H. Samet. Multi-resolution out-of-core modeling of terrain and geological data. In *In Proceedings of the 13th ACM International Symposium on Advances in Geographic Information Systems*, pages 200–209, Bremen, Germany, November 2005.

[16] Donho. NOTEPAD++. http://notepad-plus.sourceforge.net/uk/site.htm. Accessed March 10th, 2008.

[17] EA and CRYTEK. EA : Crysis. http://www.ea.com/crysis/. Accessed March 10th, 2008.

[18] Open Source Applications Foundation. wxPython. http://www.wxpython.org/. Accessed March 10th, 2008.

[19] Python Software Foundation. Python Programming Language. http://www.python.org/. Accessed March 10th, 2008.

[20] Epic Games. Epic Games. http://www.epicgames.com/. Accessed March 10th, 2008.

[21] Epic Games. Powered By Unreal Technology. http://www.unrealtechnology.com/html/technology/ue30.shtml. Accessed March 10th, 2008.

[22] Epic Games. Unreal tournament 3. http://www.unrealtournament3.com/us/index.html. Accessed March 10th, 2008.

[23] Gold Standard Group. OpenGL - The Industry Standard for High Performance Graphics. http://www.opengl.org/. Accessed March 10th, 2008.

[24] Gold Standard Group. OpenGL Shading Language. http://www.opengl.org/documentation/glsl/. Accessed March 10th, 2008.

[25] id Software. Description of MD3 Format (2006 Jun 29). http://icculus.org/homepages/phaethon/q3a/formats/md3format.html. Accessed March 10th, 2008.

[26] id Software. Quake2 Model File (md2) Format Specification. http://linux.ucla.edu/~phaethon/q3a/formats/md2-schoenblum.html. Accessed March 10th, 2008.

[27] William E. Brandstetter III. Multi-Resolution Deformation in Out-of-Core Terrain Rendering. Master's thesis, University of Nevada, Reno , December 2007.

[28] Fakespace Systems Inc. Fakespace Systems Inc. CAVE. https://www.fakespace.com/flexReflex.htm. Accessed March 10th, 2008.

[29] IDM Computer Solutions Inc. Ultra Edit. http://www.ultraedit.com/. Accessed March 10th, 2008.

[30] James Larus. SPIM MIPS Simulator. http://pages.cs.wisc.edu/~larus/spim.html. Accessed March 10th, 2008.

[31] John Lauro. Maze Maker. http://hereandabove.com/maze/mazeorig.form.html. Accessed March 10th, 2008.

[32] LucasArts. LucasArts.com. http://www.lucasarts.com/. Accessed March 10th, 2008.

[33] LucasArts. LucasArts.com — Fracture. http://www.lucasarts.com/games/fracture/. Accessed March 10th, 2008.

[34] Microsoft. DirectX® 10. http://www.gamesforwindows.com/en-US/AboutGFW/Pages/DirectX10.aspx. Accessed March 10th, 2008.

[35] Microsoft. Windows API Reference. http://msdn2.microsoft.com/en-us/library/aa383749(VS.85).aspx. Accessed March 10th, 2008.

[36] Charles Moad. Python GLEW. http://glewpy.sourceforge.net/. Accessed March 10th, 2008.

[37] Bram Moolenaar. vim Online. http://www.vim.org/. Accessed March 10th, 2008.

[38] Python OpenGL. Python OpenGL. http://pyopengl.sourceforge.net/. Accessed March 10th, 2008.

[39] Tom Poindexter. Tom Poindexter's CROBOTS Page. http://www.nyx.net/~tpoindex/crob.html. Accessed March 10th, 2008.

[40] Russell Smith. Open Dynamics Engine. http://www.ode.org/. Accessed March 10th, 2008.

[41] U.S. Geological Society. USGS Digital Elevation Models. http://rockyweb.cr.usgs.gov/elevation/dpi_dem.html. Accessed March 10th, 2008.

[42] Discreet Software. 3DS File Format. http://www.martinreddy.net/gfx/3d/3DS.spec. Accessed March 10th, 2008.

[43] Loki Software. OpenAL. http://www.openal.org/. Accessed March 10th, 2008.

[44] Ian Sommerville. *Software Engineering*. Addison-Wesley and Pearson Education, Boston, MA, USA, seventh edition, 2004.

[45] Wavefront Technologies. OBJ File Format. http://www.fileformat.info/format/wavefrontobj/. Accessed March 10th, 2008.

[46] the Gna! people. Cal3D - 3d character animation library. https://gna.org/projects/cal3d/. Accessed March 10th, 2008.

[47] Carnegie Mellon University. Alice.org. http://www.alice.org/. Accessed March 10th, 2008.

[48] Reno University of Nevada. UNR Office of Human Research Protection. http://www.unr.edu/ohrp/. Accessed March 10th, 2008.

[49] Austin University of Texas. nerogame.org. http://nerogame.org. Accessed March 10th, 2008.

[50] Patrick Henry Winston. *Artificial Intelligence*, page 94. Addison-Wesley, third edition, 1993.

[51] wxWidgets. wxWidgets. http://www.wxwidgets.org/. Accessed March 10th, 2008.