CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

TESTING THE EFFECTS OF MEMORY STRUCTURES AND RECALL ON NON PLAYABLE CHARACTERS IN UNITY3D

A graduate project submitted in partial fulfillment of the requirements

For the degree of Master of Science

in Computer Science

By

David Lu

August 2013
The graduate project of David Lu is approved:

_________________________________________  ___________
Professor Richard Covington, Ph.D.  Date

_________________________________________  ___________
Professor Gloria Melara, Ph.D.  Date

_________________________________________  ___________
Professor G. Michael Barnes, Ph.D., Chair  Date
ACKNOWLEDGEMENTS

I wanted to express my gratitude to professor Covington and Melara for providing their time and support. I wanted to give a special thanks to professor Barnes for inspiring me to create this project and experiment. With his knowledge and expertise I learned valuable skills that not only affect my academic life but my personal life as well. I would also like to thank California State University, Northridge for preparing me with the experience and tools necessary to achieve my goals. Lastly, without my family I would have not been able to achieve such success and I thank them for all their love and support.
# TABLE OF CONTENTS

- Signature Page i
- Acknowledgements ii
- List of Figures v
- List of Tables vi
- Abstract vii
- 1 Introduction 1
- 2 Unity3D 5
  - 2.1 Views 6
  - 2.2 Objects In Unity3D 8
  - 2.3 Elements of Unity3D 10
    - 2.3.1 Collider 11
    - 2.3.2 Rigidbody 11
    - 2.3.3 Camera and Terrain 12
    - 2.3.4 Triggers Handling in Unity3D 13
    - 2.3.5 Time in Unity3D 14
    - 2.3.6 Execution Order Functions in Unity3D 14
- 3 Experimental Design and Hypothesis 18
- 4 Methodology 24
- 5 An Experimental Run 27
  - 5.1 Initialization 27
  - 5.2 Running the scene 31
  - 5.3 Recording results 35
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4 Implementing with Unity3D</td>
<td>36</td>
</tr>
<tr>
<td>6 Results</td>
<td>38</td>
</tr>
<tr>
<td>7 Discussion</td>
<td>46</td>
</tr>
<tr>
<td>8 Conclusion</td>
<td>51</td>
</tr>
<tr>
<td>Bibliography</td>
<td>57</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 2.1-1: Views 6
Figure 2.3-1: Rotation Plane 10
Figure 3.0-1: Food Source Interaction 20
Figure 5.1-1: Texture Scene View 28
Figure 5.1-2: Wireframe Scene View 29
Figure 5.2-1: NPC State Diagram 32
Figure 5.2-2: Next Node Indicator 33
Figure 6.0-4: Average Total Distance for Memory Structures and Recall 43
Figure 6.0-5: Average Food Eaten for Memory Structures and Recall 44
Figure 6.0-6: Average of Food Eaten with Recall 45
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 4.0-1:</td>
<td>Experimental Conditions Table</td>
<td>23</td>
</tr>
<tr>
<td>Table 5.4-1:</td>
<td>Table of Classes</td>
<td>36</td>
</tr>
<tr>
<td>Table 6.0-1:</td>
<td>Table of Distance Travelled and Food Eaten Average</td>
<td>38</td>
</tr>
<tr>
<td>Table 6.0-2:</td>
<td>Univariate Analysis</td>
<td>40</td>
</tr>
<tr>
<td>Table 6.0-3:</td>
<td>Multiple Comparisons for Memory Structures</td>
<td>41</td>
</tr>
<tr>
<td>Table 6.0-4:</td>
<td>Multiple Comparisons for Percent Recall</td>
<td>42</td>
</tr>
</tbody>
</table>
ABSTRACT

TESTING THE EFFECTS OF MEMORY STRUCTURES AND RECALL ON NON PLAYABLE CHARACTERS IN UNITY3D

By

David Lu

Master of Science in Computer Science

In most games today there is some sort of artificial intelligence involved. Some of these artificial intelligent Non-Playable Characters (NPCs) [8] require memory structures to store locations of objects in their virtual environment. The type of memory structure used will depend on the game. Along with memory structures, recall was used to see the effects on the NPC's performance. An experiment was created to test the effects of memory structures and recall percentages on NPCs.

The focus on this experiment was to see how far an NPC travels and how much food it consumes while operating with a certain memory structure and recall percentage. The NPCs randomly travelled in a virtual environment searching for randomly placed food sources. When the NPC got hungry it used its memory structure, to search for a food source and used recall to see if it "remembered" where the food source was. Memory was structured on closest, random, FIFO, or LIFO. Percent recall was 100%, 66%, or 33%.

The distance travelled and food eaten were recorded. This experiment was implemented using Unity3D [10]. Both memory structure and recall significantly affected NPC lifetime. Closest, random, and FIFO were better than LIFO. 66% recall was the best. The experiment was designed with respect to AI for NPCs in games.
1 INTRODUCTION

Memory structures are used in a lot of programs and games. The type of memory structure used in a project will depend on the application/problem. Memory structures such as LIFO and FIFO are used in everyday life. FIFO for instance can be applied to a checkout line at a grocery store. The first person in line will be the first person to leave. If LIFO was applied to the same situation the first person in line will always be last. Since customers are always entering the grocery store that person may never get to leave the store. This is also transferrable to waiting in a queue to connect to a server, the server being the grocery store. In a game, using the wrong memory structure could be devastating. A game where an NPC [8] or Non Playable Character has to set priorities on what to do and where to go next will determine how successful the game is. If the NPC decides to attack before it finds a weapon due to having "to get a weapon" set last in its memory structure than it will not be successful in its attack. In this case, order is everything.

In addition to order of a memory structure adding recall to it can be an interesting feature to a game. It can control if an NPC "remembers" what their instruction was. Combine recall with a memory structure and it could affect how an NPC moves and acts and the overall quality of a game. Recall will control if the NPC "forgets" or "remembers" objects in the memory structure. Objects that could be directing the NPC on where to go next. If the NPC forgets, it will in a sense go through a list of objects that it can remember. This provides a little randomness to a game making it less predictable and more enjoyable.

I feel this topic is important because it helps better understand memory structures and see the impact they may have on a game when using recall. This can create a more
effective decision making NPC with the right match of memory structure and recall. With the wrong memory structure and recall a game can be unplayable. The NPC may not remember anything and mindlessly wanders around. It may also be too predictable and provide the gamer with no challenge making the game too simplistic. The question this report will discuss is how will both memory structure and recall affect an NPC in an experimental environment?

To answer this question I want to create a testing environment for an experiment to see the difference among the more common memory structures and some custom ones with the addition of recall. Since all games aren't created equal I wanted to create an experiment with a specific task in a specific environment. I want to keep this experiment as simplistic as possible and focused on testing memory structures and recall. Some games have a task for a player to perform such as navigating through a maze or solving a puzzle. This experiment will have its own objectives and will not involve any player interaction, instead it will use an NPC that will operate a basic form of artificial intelligence. This NPC will have to make choices based off its memory structure and if it can recall the objects in it or not.

The experiment will use the more common memory structures such as LIFO and FIFO and two custom ones: closest and random. It will also apply a recall chance of 33%, 66%, and 100% to the memory structures. An NPC will be using these memory structures and recall percentages to achieve its goal. The task or goal for the NPC is fairly simple, survive. Survival will be measured by the total distance travelled and the number of food sources eaten before "death". The environment will consist of a terrain that contains an NPC and food sources. The NPC will randomly explore the terrain for food sources.
When it discovers food sources it will "learn" by placing knowledge about the food source ("object") in its memory list. The memory list will follow a memory structure. For example, if the memory structure is closest the closest discovered food source near the NPC will be first in its memory structure followed by the second closest etc. In conjunction with memory structures I added recall into the experiment. The recall will be the chance the NPC will remember where the food source is. If the NPC fails to recall the first food source in its memory list it will move on to the next one until it remembers one. The NPC will start at one hundred food units in its food meter and then it decrements its food meter over time. Once the food meter decrements to 50% the NPC will begin foraging for food (its hungry). This is where the NPC uses its memory list, recall, and memory structure to locate food sources and try to live as long as possible. When the NPC's food meter reaches zero the NPC is dead the experimental "run" ends. When the run ends the results (memory structure, recall percentage, total distance travelled, number of food eaten) will be recorded to a text file for later analysis.

The experiment will consist of running all the conditions of the memory structures and recall. For example running LIFO on 33%, 66%, and 100% recall. To show that the results weren't done by chance each conditions will be run many times (15+) per condition. Any erroneous runs will be discarded and run again. Once all runs are done for all conditions the results are recorded and will be imported to SPSS [4] by IBM. SPSS will be used to calculate data such as the averages of total distance travelled, food eaten for all conditions, and test specific hypothesis.

Expanding on the question stated earlier "how will both memory structure and recall affect an NPC in an experimental environment" more in depth questions and
statements were created for this experiment. These will be my hypotheses for this experiment: Closest on 100% recall will have the highest distance travelled. Closest on 100% recall will have the highest amount food eaten. FIFO on 33% recall will have the lowest distance travelled. FIFO on 33% recall will have the lowest amount of food eaten. Closest should provide the highest amount of distance travelled followed by LIFO, random, and FIFO. Recall percentage 100% will have the highest distance travelled followed by 66%, and 33%. Memory structure and recall will be dependent on each other. Total distance travelled will correlate to food eaten.

To support my hypothesis I will be using Unity3D [10] to run the experiment. Unity3D is a gaming engine created by Unity Technologies [10]. I will be designing a project that will better help visualize the experiment in real time. I decided to use Unity3D because I have an interest in game development and thought it might be useful to visually "see" the experiment rather than relying on just text output. I will be able to see the NPC move around in the terrain as well as where the food sources are placed. Unity3D will also help in keeping track of the food sources in the NPC's memory list and how much food is in the NPC's food meter in real time. It also will help debug errors and make sure the NPC goes to the right location. I also wanted to learn and gain experience developing with Unity3D. In this report I will go over: Unity3D, the experimental methodology, the results of this experiment, and a discussion and analysis of the results.
Unity3D is a development kit to create PC and video games. It was developed by Unity Technologies and was designed to help studios as well as independent game developers develop fully functional games. There are two versions of Unity3D, the free version and the Pro version. The free version allows for development for PC games and web games while the Pro version can be used for Android [1], iOS [2], and consoles such as the Playstation3 [9]. Since Unity3D was developed to be multiplatform it can be scripted using C# [7], Javascript [5], and Boo [3]. MonoDevelop [6] is the scripting interface that comes with Unity3D and contains components such as a debugger and all necessary libraries associated with each framework and scripting language. Unity3D features a variety of tools at the developer's disposal. Such tools like the scene builder allows the user to drag and drop meshes, prefabs, and game objects into the game effortlessly. One tool allows you to play and pause the game at any time and change any variables such as health during run time to see how they affect the game. Another tool allows the user to create terrains, levels, and apply physics to game objects. All of these tools are integrated into their editor and can be used via the "Interface". The Interface allows you to interact with all the game objects, the scene, and the project files. It contains several views such as the console, the scene, the project files, the hierarchy, the inspector, and the game. Each view serves a specific purpose such as examining a game objects properties or how the in game view will look like. All of these elements are used to create scenes, game objects, components, assets, and prefabs.
2.1 Views

The views are the windows and interface to Unity3D’s editor. Each one serves a different purpose and allows the developer quick and simple access to objects, scenes, and their code. Refer to Figure 2.1-1. There are six views and the traditional top menu in the Unity3D editor. The six views are: console, scene, project, hierarchy, inspector, and game. The console is to view output from the code. It operates like any other console on any IDE and is useful to help debug your program and output data or view any errors that may occur. In my project I used it to monitor the amount of food in my scene.

![Figure 2.1-1: Views.](image)

The scene view is where all the interactions between the other views occur. This view is where the developer can manipulate the game by dragging and dropping prefabs and other game related objects. The window can also be customized to show different
views such as a wireframe view which creates a wire grid of the terrain and all objects to help visually see what is happening. The game can also be run in this screen and is fully controllable allowing the user to pause and stop the game when needed. An example of the flexibility of the scene view allows the developer to adjust the angle of the camera to get better views, you can drag in extra objects as the game is running, and you can move as well as rotate objects around in the scene during run time.

The project view contains all the files associated with the project. Files such as your scripts, scenes, textures, and prefabs are located there. These are the objects the user will drag onto the scene or onto another object. Unity3D advises import or export of scripts and files using the project view rather than creating the files directly on the file system to avoid compatibility problems. The scripts in the project view can be opened up with Monodevelop. It is an IDE that can be programmed with C# and Javascript. It's integrated with Unity3D so it automatically builds and reports errors in the console view of the interface.

The hierarchy view displays all the objects that are used in the scene. This is used to keep track of all the current objects in the scene. It allows the developer to select the object in the hierarchy view which will also select the object in the scene view which allows simpler access to the object. Not only does the object get selected in the scene it also helps display all the components and scripts attached to the object in the inspector view.

The inspector view shows the components that belong to the object. Every object in Unity3D can be associated with a name and/or a tag. The name is the unique id of the object and can be changed in the inspector view. The name is usually used to uniquely
reference the object. The tag is more like a category or group for the object. Multiple objects can have the same tag and when the developer needs to find all objects of group and edit a property, it can be done simply by using the tag. The inspector view also displays components such as scripts (that control events and movement), meshes and texture that give the object shape, and a transform. The inspector displays all the properties of anything selected in the hierarchy view and the project view. The inspector view allows the user to view the object or code without actually having to open it up in an editor or create any extraneous code to view properties or locations of an object.

The game view is what the player will see when they run the game. In this view nothing can be directly modified. The game view can be run directly in the editor when you click the play button. If the game requires player interaction, the game view will handle all key strokes and clicks when the game view has focus. Textures, game objects, and lighting effects will all be rendered once the game view is run.

2.2 Objects In Unity3D

When you break down a Unity3D program it consists of scenes, components, assets, game objects, and prefabs. Scenes are "levels" in the game. There can be multiple scenes or levels in a game. Scenes contain all the parts listed above and combine them into one functional level. The scene is what the player will see when the game is started.

Components are the scripts that are executed when the game is started. These scripts can be attached to objects such as the terrain or the camera but mainly are used for game objects. They act like instructions or pieces of code on what the object it is attached to should do. These scripts or components could control where a game object could move next, how fast it should move, and when it should stop.
Assets are models or meshes that can be applied to an object. They are mainly used to apply a texture or animation to a model or game object. Assets can also be sound files such as birds chirping in the background or a fiery explosion.

Game objects are containers that will eventually store data, components, and interact with the scene. They can be used as the virtual representation of the player, an NPC (non playable character), or even a stationary object such as a tree or rock. Game objects are fully customizable, they inherit components and assets and can become part of a prefab. They also may be non-rendered objects that are invisible in the game. For example, they could act as checkpoints for some other game object to follow or trigger an event when another game object approaches it. All of these objects could be used to create prefabs.

Prefabs are components, assets, and game objects put together. Prefabs are like blueprints for an object in the scene. The components, assets, and game object attached to them are the properties of the prefab. They contain the code and the textures that are connected to a game object. They then become a simple entity of the scene combining components, assets and game objects. Prefabs increase reusability because they can easily be dragged and dropped into the scene. Dragging and dropping creates copies (instances) of the original prefab. Prefabs are also easy to maintain. For example, you can drag and drop one thousand monsters or prefabs in the scene and if you need to make a change to the monsters all you have to do is edit the prefab rather than editing each monster individually. You can drop as many prefabs on a game object or scene as needed and create new ones for future game objects to use.
2.3 Elements of Unity3D

In Unity3d a transform is a component that is attached to all game objects such as a camera or terrain. It controls the position, rotation, and scale of an object. The position of an object is based on an x, y, z axis. By default all transform positions are set to 0,0,0 which is the bottom left of the grid (left handed system). When the transform moves around the x, y, and z will either decrement or increment depending on the direction the transform moves. Rotation controls the transform's rotation in the game. Rotation on the x,y,z axis is also called yaw, pitch, and roll. An airplane is the best example to display how it works. Refer to Figure 2.3-1. Like the default of the transform the default rotation is 0,0,0.

Figure 2.3-1: Rotation Plane. (A example of yaw, pitch, and roll using a plane.)

From: http://www.allstar.fiu.edu/aero/ftmidcont.htm

Scale relates to the width, length, and height of a mesh or model. Scale is important because it relates to any physics components attached to the mesh. If the scale is set to 2,2,2 than the mesh will be double the size and in physics component calculations it will be double the weight. All these properties can be edited with the
inspector pane with inputs or in the scene view visually by rotating the mouse when the object is selected.

2.3.1 Collider

Colliders are invisible spheres, cubes, and other shapes that surround a game object. Colliders are used for interactions between other game objects that have colliders. When colliders from two different objects intersect a trigger occurs. The triggers signal the game objects of where they collided (what areas of the colliders were hit) than can be scripted to perform an action. One use for colliders is to prevent one game object to run through another one. When colliders intersect data is passed between the two objects and in this case will prevent the objects from going through each other. Both colliders can access the components attached to the game object they collided to and the properties attached to those components if their set to public. Depending on the type and shape of the collider detecting collisions can be very pin point. A mesh collider will follow the shape of the model or act as its skin. This provides accurate collision data but at the cost of slowing the game down. Primitive colliders such as spheres, cubes, and capsules require less calculation during collision but aren't as pinpoint as mesh. It comes down to the developer to see if they want to sacrifice detection coverage for optimization or vice versa. All game objects that have colliders require a component called a rigidbody to move.

2.3.2 Rigidbody

If the game object is a static object such as a tree or a mountain and never moves than a rigidbody isn't necessary. But if you want the laws of physics applied to your game object than a rigidbody is required. A rigidbody is a component that deals with all the
physics in your virtual environment. Unity3D has a physics component that deals with properties such as mass, drag, and gravity. Rigidbodies require colliders to make these calculations. These are calculated in run time and depending on the type of collider in use it could slow the game down. An example of a collider and a rigidbody working together is a wheel. The wheel collider will create the detection and the rigidbody will keep the wheel to the ground with gravity and tumble up and down if the terrain is rugged. In this experiment a rigidbody is used to apply gravity to the NPC preventing it from floating. Rigidbodies allow the developer to save a lot of time dealing with the physics of their virtual world without actually having to create their own physics engine.

2.3.3 Camera and Terrain

Cameras play a pivotal role in a scene. Obviously if there is no camera than players of the game won't see anything. The camera is a mandatory object in the scene and can be manipulated like any game object. The camera has a view area of what the camera is shooting or viewing. This is what the player will see. The camera being a game object can be dragged and dropped in the scene and rotated in the direction where the developer wants the user to view. There can be multiple cameras on a scene and they can be called by their name or tag. These cameras can be controlled by a script and move around the terrain for scenic view or static cameras that activate and deactivate when an event is triggered. Terrain is the ground a players object will walk on. It's the hills, roads, and grass that make the virtual environment come alive and provide the user with entertaining game play. Unity3D comes with a terrain engine that generates a terrain based on set properties the user can tinker with. The user can set the texture of the terrain from Unity3Ds custom made textures or import one of your own.
2.3.4 Triggers Handling in Unity3D

Triggers in Unity3D are the scripted actions that occur when a condition is met. The triggers are usually activated by collisions between two game objects. The functions OnTriggerEnter(Collider c) and OnTriggerExit(Collider c) are part of Unity3D's API that deal with the actions once a collision occurs. The OnTriggerEnter function listens for collisions using the collider. When a collision occurs or an object walks inside the radius of the collider, the OnTriggerEnter is fired. Any script in this function is run as long as the object doesn't exit the radius. OnTriggerExit works similarly to OnTriggerEnter except any code or scripts in this function is run when the object is outside or not colliding with the other object. Each object can have both functions so when a collision occurs each object could handle OnTriggerEnter and OnTriggerExit differently.

There may be many game objects colliding with other game objects in the scene. In order to differentiate between them, every object has a name and a tag associated to each object. This can be set in a script but can also be done in the inspector view. Once a tag or name is established the OnTriggerExit and OnTriggerEnter contains a parameter of type Collider. This is the object that collided with the object. Using the argument that was passed to those functions you can get the tag, name, and components of the object. Using collider.CompareTag(tag name here) will get you the object's tag. This helps distinguish which trigger should be run. For example there is a game object that has the tag name "ball". When a game object collides with an object with the tag ball it first checks its tag and then runs its OnTriggerEnter function and checks if(collider.CompareTag("ball") == true). If this returns true than the developer could script the ball object to roll. When the object doesn't collide with the ball anymore the OnTriggerExit will run the script for the
ball to stop rolling. The name is used exactly the same way a tag is used to get objects except names have to be unique where as tags do not. With the ability to access the components of the object you collided with, this allows the current object to gather needed data from the other object. Using these built in functions and components allows for smoother handling of events and triggers.

2.3.5 Time in Unity3D

The amount of frames per second is based on the hardware that will be playing the game or in this case running the experiment. If you have a very high end system than frames will process quicker in your game than in a low end system which will process the frames slower. More frames will lead to more updates allowing the higher end systems to "see" things before the slower systems can. This problem can be dealt with using Time.deltaTime. Time.deltaTime keeps track of the time it took to finish the previous frame. Using this makes the game time dependent rather than frame dependent. One example of this is in moving an object. Let's say a rock travels five meters. During these five meters the rock gets to its destination. Without Time.deltaTime the rock moved five meters in a single frame. With Time.deltaTime the rock travelled five meters a second. This removes any advantage a higher end system has over a slower one by going by time instead of frames. The higher end system could have moved twice as fast or more due to all the updates and frames it could execute.

2.3.6 Execution Order Functions in Unity3D

In Unity3D there are functions that keep the game running. Some functions initialize variables, some functions update those variables, and other functions execute other functions until they are stopped. Before a game updates game variables like health
it has to instantiate the game objects, components, and other variables into the scene with the awake function.

The awake function is the first function called before the game begins. Unity3D will randomly go through all the active game objects and scripts in search of the awake function and run those scripts. This ensures that variables and scripts that need to be created or executed are done before the game begins. One example is the creation of game objects or prefabs. You can't set the health of an object before it is instantiated or apply a texture to a null object.

The start function is similar to the awake function except that it is called when the component is instantiated into the game rather than randomly called. It is called after the awake function. The start function's use is to initialize variables and properties when a script or a component using that script is activated. For example, there are two game objects that are created by the awake function and they are named rock and player. When these objects are created they each have their own start function. Rocks start function may contain nothing because it's part of the terrain in the scene but the start function will still be executed. The players start function will contain health, energy, and other properties that need to be set before any updates of the object can occur.

The update function is the main update loop that runs the game. This update is where all the game updates occur. Updating health or locations of the game objects are done here. Since each script contains its own update function each script could be updating their positions and other properties as other objects update theirs. These updates can also be used for triggers and any OnTriggerEnter or OnTriggerExit calculations that are made. OnTriggerEnter calculates what points made contact between colliders and the
update function will use those calculations and run any code that developer has once the trigger fires. The update loop can check the distance between an object and a location and trigger an event such as an explosion when the object is nearby. Almost everything that involves any type of update whether its rendering a texture, player interaction, or AI scripted behavior is done here.

Coroutines are functions that are called and then stopped. The routine will run a function that has been created by the developer and it will run it repeatedly. The coroutine stops when you call the function stopcoroutine(String function). When and how coroutines execute is key here. If you use the keyword yield on the coroutine than the coroutines function will wait for the next frame and after all the update functions to run and then execute. The function yield waitforseconds(int) will wait till the update function is run and then wait the specified amount of time to execute the coroutine function. Using the yield startcoroutine(function) will run the current coroutine after the function finishes executing. An example of using coroutines is in a game that has timed events. Each timed event can be run on its own coroutine. When time runs out on one event the developer can call stopcoroutine and stop that specific event while the others are still running.

In summary the execution order of scripts begins with all the awake function calls then all the start calls. After that OnTriggerEnter, and OnTriggerExit are executed for all scripts that have them. Next comes the update function which executes or uses any calculations from the above functions to perform tasks such as movement. The final execution is the rendering of game objects and meshes in the scene. After this it repeats until the game is over, paused, or stopped.
Unity3D provides a powerful gaming engine that will supply any developer with a clean and uncluttered API and an easy to use drag and drop interface. These tools provide a smoother and time saving way to develop the game of your dreams. Unity3D was easy to adapt to and very intuitive in terms of what you expect from a game engine. It contained many libraries and components that made creating this project far simpler than I thought it would. I didn't have to code every single aspect of this project from scratch (for example the colliders). Most of the important aspects of Unity3D were discussed here but Unity3D contains far more tools and components that developers will find useful. For more information on Unity3D please go to their website at http://unity3d.com/.
This experiment was created to determine the life expectancy of the NPC based on total distance traveled and food eaten using different memory structures and recall in an experimental environment. Using common memory structures such as LIFO and memory recall (percentage) this experiment will determine the total distance an NPC will travel before running out of food and how much food the NPC has eaten. A list of questions will be answered such as: Will a closest (distance) memory structure with 100% recall result in the highest distance travelled and food eaten? Will FIFO on 33% have the lowest distance travelled and food eaten? Will the order of the memory structures be closest, LIFO, random, and FIFO (closest being highest distance travelled and FIFO being the lowest)? Will the order of recall percent be 100%, 66%, and 33% (100% having highest food sources eaten and 33% having the lowest)? Will memory structure and recall be dependent or independent of each other? Will total distance travelled correlate to food eaten. (More distance travelled, more food eaten)? These questions will be tested and answered based on my design of the experiment.

The experiment will contain two variables: memory structure, and recall. What will be measured in this experiment will be total distance travelled and food eaten. Distance travelled is based on Unity3D's measurement of distance. Unity3D refers to distance in meters but I refer to them as units to avoid any confusion between Unity3D's meter and the metric system's meter. Food eaten is determined by how many food sources the NPC has exhausted in the experiment. Memory structures will serve as the variable on the order the NPC will recall location of food sources. Multiple memory structures will be tested in this experiment. These structures are LIFO, FIFO, random, and closest.
Recall is the percentage that the NPC will recall a food source in its memory. Those percentages are 33%, 66%, and 100% chance of recalling a food source location. A key component to this experiment is hunger. Hunger will decrease the NPCs food meter and when it is depleted to zero the NPC will die. The NPC will become hungry when the food meter is 50% or below and begin searching for food. When hungry the NPC will seek food and use its memory and recall to stay alive as long as possible. When the NPC is not hungry it will explore the terrain and discover food sources for its memory. Without any new food sources the NPC can only travel a total of 300 units.

Since the experiment is implemented with Unity3D everything will use Unity3D components. The experiment itself will need an NPC, food sources, food meter, terrain, and hunger. The NPC will be the object that will locate the food sources and try to survive using it's memory structure and recall. A grid of "invisible" nodes will be created. The NPC and the food sources will be created randomly on these nodes and when exploring it will randomly select and use these nodes as waypoints to its destination. Depending on where the location of the NPC is, it has at most eight neighbor nodes or directions it can go: up, down, left, right, and four diagonal directions. An A*[11] path finding algorithm is used to calculate the path to the main destination for the NPC. This occurs both when the NPC is randomly travelling (explore state) and when it moves toward a food source it "remembered" (forage state). They both operate the same way, when the NPC reaches a node that becomes the start node. From there A* is used to calculate a path from the start node to the destination node which may or may not contain a food source. Initially the NPC will be exploring the terrain randomly discovering food sources as it encounters them.
The NPC will be displayed as a red cube mesh. Refer to Figure 3.0-1. Food sources are what keeps the NPC alive by satisfying its hunger. These will be represented with green sphere meshes on the terrain. Refer to Figure 3.0-1. The food meter keeps track of the NPCs hunger. It starts at one hundred and if it reaches zero than the NPC dies and the run is over. The terrain is the "land" the NPC will walk on and where the food sources will be randomly placed. The land is completely flat to prevent any physics calculations that may taint the results.

![Figure 3.0-1: NPC-Food Source Interaction.](image)

Based on the questions stated earlier a hypothesis was developed that the NPC will travel the farthest distance or live longer using the closest memory structure and 100% recall. The NPC will use closest to locate the closest food source based on its current location and will always recall where the food source location is. This will allow it to eat quicker and travel less to stay alive longer. The same hypothesis will apply with the highest number of food eaten. The combination that will result in lowest distance travel will be FIFO with 33% recall. Since FIFO will place the most recent food source in
the back of the memory list this will most likely place farther food sources to the front of the memory list. With recall at 33% its more likely that the NPC may not recall anything in its entire memory list. When this happens the NPC will switch to explore state for its next step. It will still be hungry, so the subsequent step will switch back to the forage state. Lowest amount food eaten will follow the same assumption which is FIFO on 33%.

In terms of memory structure and the highest total distance travelled, closest should provide the highest amount of distance travelled followed by LIFO, random, and FIFO. The reason closest is first is because the NPC will travel less and get to the food source the quickest. LIFO was next because the most recent food source may be the closest resource but depending on the location of the NPC this may not always be true which is why it is second on the list. Random was third because the chances the NPC will remember a food source close to it is as good as remembering a food source that is really far from it. Since random will randomize food source location in the memory it’s possible the NPC may get an entire list with closest food sources first. FIFO was placed last because any food source it finds first will be placed last. This will cause the NPC to explore more, possibly creating more distance between that resource and the NPC. The further that NPC is from the resource the more it expends it's food meter making the trip to that food source hurt the NPC more than help it.

Recall percentage at 100% will perform the best followed by 66% and 33% respectively. With recall at 100% the NPC will always be able to "remember" where the food source location is. This will further it's lifespan and distance travelled by preventing the NPC to go to food sources further down in its memory list therefore having the NPC travel farther. Memory structure and recall will be dependent on each other. Memory
structure will provide the order of the memory and play a factor whether the NPC will go to the closest food source or not. Recall will play a factor if the NPC actually goes to the food source. Both will affect each other whether the NPC is successful getting to the food source or not. Total distance travelled will correlate to food eaten. The longer the NPC lives the more food sources it will consume. Thus, the higher the total distance travelled the more food eaten.
In summary there are eight hypothesis

1. The combination that will result in highest distance travelled will be closest on 100% recall.

2. The combination that will result in the highest amount food eaten will be closest on 100% recall.

3. The combination that will result in lowest distance travelled will be FIFO on 33% recall.

4. The combination that will result in lowest amount of food eaten will be FIFO on 33% recall.

5. Closest should provide the highest amount of distance travelled followed by LIFO, random, and FIFO.

6. Recall percentage 100% will have the highest distance travelled followed by 66%, and 33%.

7. Memory structure and recall will be dependent on each other.

8. Total distance travelled will correlate to food eaten. (More distance travelled, more food eaten)
4 METHODOLOGY

The experiment was created with Unity3D. This allows a visual representation of what is happening in the experiment. This experiment will test how far an NPC will travel in terms of Unity3D’s units based on its memory structure and recall. The system that will be running this experiment is an Intel Pentium Dual CPU T3200 @ 2.00GHz with 3GB of DDR2 RAM operating on Windows Vista Home Basic 32 bit. This system was used to build the experiment as well as run it.

The four memory structures are LIFO, FIFO, random, and closest. The three recall percentages are 33%, 66%, and 100%. These parameters will be run in pairs. For instance running the memory structure LIFO on 100% recall. There will be twelve conditions. Refer to Table 4.0-1 for the conditions.

Each condition will be run at least fifteen times. For example, Closest on 0% recall will run until the NPC dies. After that a new run begins with the same parameters (Closest on 0%) until the fifteen runs are finished. This is to provide more accurate and reliable results. This also prevents any experimental error by eliminating any runs that ended abruptly due to an error. Having multiple runs will also show whether the data accumulated was by chance or not; to statistically test the hypothesis.

<table>
<thead>
<tr>
<th>Random on 0% recall</th>
<th>Random on 33% recall</th>
<th>Random on 100% recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO on 0% recall</td>
<td>FIFO on 33% recall</td>
<td>FIFO on 100% recall</td>
</tr>
<tr>
<td>LIFO on 0% recall</td>
<td>LIFO on 33% recall</td>
<td>LIFO on 100% recall</td>
</tr>
<tr>
<td>Closest on 0% recall</td>
<td>Closest on 33% recall</td>
<td>Closest on 100% recall</td>
</tr>
</tbody>
</table>

Table 4.0-1: Experimental Conditions Table. (Table of all the conditions being tested)
There will be sixteen food sources that are randomly placed on the terrain that the NPC can exhaust. This number was decided for two reasons. The first was to provide the NPC with enough food so that the NPC will always find a food source regardless of memory structure and recall. If the NPC never finds food than it won't use its memory and recall which will defeat the purpose of the experiment. The second was to prevent the NPC from living too long which would defeat the purpose of memory and recall because anywhere the NPC goes will contain food. Each food source will contain one hundred units of food. Food sources will not regenerate. If a food source has one hundred units and the NPC eats eighty units than the food source will decrease to twenty units and stay that way until the NPC eats from it again or the run ends. Food sources that are depleted or dropped to zero units will be removed from the terrain and from the NPC's memory.

Should the hungry NPC walk close to a food source even if it wasn't its initial destination the NPC will consume as much as it can from the food source. If still hungry it will continue on to its destination. If not hungry it will explore.

The NPC will begin the experiment full with one hundred food in its food meter. The NPC will explore the terrain looking for food sources and place the location of these food sources in its memory. Over time the NPC will lose food from its food meter. At 50% hunger the NPC will stop exploring and begin using its memory to locate previously found food sources. If the NPC has no food sources in its memory list at 50% or below or the NPCs meter goes above 50% the NPC will go back to exploration for its next step. If the NPCs food meter reaches zero than the NPC will die and a new run will begin.

Every time a run ends the data is recorded and written to a file. The file will store the experimental conditions (total distance and food left). This can also be used as the
amount of food sources the NPC fully consumed (16-food left on terrain). The results of the entire experiment will be analyzed with SPSS.
5 AN EXPERIMENTAL RUN

A run is a single test trial of a memory structure and a recall percentage. It is used to measure the amount of distance the NPC will travel under these conditions. It is also another way to measure how long the NPC lives via distance. A run consists of an NPC that randomly moves around the terrain in search of food sources to increase its survival. To achieve this, the experiment can be broken into three processes: initialization, scene execution, and data recording. Initialization will handle instantiating all the variables and components before the scene begins running the experiment. After initialization of the variables the scene will use the components that have been created to perform the run. Once the run finishes the results will be recorded for later analysis.

5.1 Initialization

The initialization process instantiates and creates all necessary variables and components before the experiment begins. Nodes will be generated 11 units apart horizontally and vertically and 16 units apart diagonally. There will be a total of 81 nodes on the terrain which will be a 9 by 9 grid. Refer to Figure 5.1-1 for the textured scene view and Figure 5.1-2 for the wireframe scene view. These nodes are game objects that will act as guides to the movement of the NPC. The nodes are invisible and will cause no collisions during run time and are only used for keeping track of NPC and food source positions during the run of the experiment. These nodes will provide the NPC movement choices of going one node up, one node down, one node left, one node right, or one node diagonally. The NPC travels node to node from one destination to the other. These nodes are also responsible for creating random positions for the NPC as well as the 16 food sources.
Figure 5.1-1: Texture Scene View. (Texture scene view with NPC and food sources)
The NPC will be created as a collidable game object with a red cube mesh. It will be created on a random empty node (a node that doesn't contain a food source) and move between nodes as a method of transportation. This type of movement was used to control the NPCs movement on the terrain. One reason to do this was to prevent the NPC from accidental collisions between two food sources at once by providing adequate spacing between them. Since the nodes are separated by distance between other nodes this prevents that problem from occurring.
This process also sets the food sources to one hundred units. This is to ensure that the food sources have identical amounts for all the runs regardless of the memory structure and recall percentage. The food sources will be green sphere meshes that are randomly placed on empty nodes. These are collidable game objects that appear when they are above one hundred units and disappear when the NPC is using it or when the food source is zero. Memory structure and recall percentage is set here as well. The memory structure can be set to LIFO, FIFO, random, and closest. The recall percentage can be set to 33%, 66%, and 100%. The memory storage itself will be using the List structure in C#. This list will have no limit to the amount of food sources it can carry but the max it can have is based on the max amount of food sources that are loaded into the scene. After the fifteen runs are completed another test case will be used. For example, the experiment will start off with closest at 33% recall after fifteen runs the next will be closest with 66% recall and after that it will be closest at 100%. After that the memory structure will change following the same pattern until all combinations of memory structure and recall percentages have been tested for fifteen runs each.

The food meter that keeps track of the NPCs hunger will be set to one hundred units. The NPCs food meter decrease rate will also be set in this process. The current rate of decrease is 1 unit per second and the NPC moves 3 units per second. This number was used because it allowed the NPC to live long enough to locate food and at the same time didn't allow the NPC to live long enough that food sources weren't needed. After the food meter is setup as well as all the other components the experiment can be begin.
5.2 Running the Scene

When the initialization process is finished the scene will run the experiment. This is where the experiment will test the various memory structures (LIFO, FIFO, random, and closest) and recall percentages (33%, 66%, 100%). The testing will consist of many triggers and event handling through a run. An example of these events or triggers will deal with removing food sources when they reach zero units and keeping the memory list of the NPC updated to ensure the list is functioning with the latest data set of food sources. The NPC will be in one of three states and will interact with food sources as it moves. During the run the NPCs food meter will increase and decrease as it travels around the scene. All these interactions and triggers occur in the experiment.

The NPC can be in three states during the experiment: explore, forage, and death. Refer to Figure 5.2-1. Explore is the starting state of the NPC. The explore state is where the NPC randomly moves around the terrain node by node searching for food sources. The NPC can move left, right, up, down, and diagonally. It may even back track to areas that it previously explored. When the NPCs food meter reaches 50% the NPC will transition into the forage state. This state is where the NPC uses its memory and recall to get food sources. If the NPC has no food source locations in its memory it will transition back to the explore state. The death state can be reached from those two states when the NPCs food meter drops to zero. This is when the run ends and a new one will begin.
During the run there will be colored line indicators to help visualize where the NPC will move. These are lines that appear in front of the NPC to direct where it is going. Refer to Figure 5.2-2. There are two types of lines. The red line indicates the next node the NPC will go to. These nodes are also referred to as sub nodes. Since the NPC moves node to node it will select one of the immediate nodes around it and the red line will display a line between the NPC and the selected node. This line will stay rendered until the NPC reaches the node and then targets another node. The second line shows node of the food source selected from the NPC’s memory. A yellow line is used to indicate this node’s location. Refer to Figure 5.2-2. The node or location may or may not be an immediate node. The yellow line connects a line from the NPC to the main destination node.
As the NPC moves around the scene its food meter will decrease one unit per second. If the food meter reaches zero the NPC is dead and a new run begins. To avoid death the NPC will have to move around the scene to locate food sources to extend its life. The NPC moves about 3 units a second and moves a total of 300 units before death if it never locates any food sources. As the NPC moves it will encounter food sources. These food sources can be a location in the NPCs memory, a food source that is on the way to a location in the NPCs memory, or a randomly found food source. When the NPC encounters a food source multiple events occur. The first event will cause the food source to hide from the scene and cause the NPC to stay still as it is eating. This was designed to show that the NPC is consuming the food source. The next event will increase the NPCs
food meter as much as the food source can give. For example if a food source has 100 units and the NPC has 40 units in its food meter than the NPC will eat 60 units to get to 100 units and the food source will decrease to 40 units. This allows the NPC to return to eat from the food source when it needs it again. The third event places the location of the food source in the NPCs memory list and performs any sorting if necessary (explained further below). The alternative event is if the food source was at 20 units and the NPC needs 60 to be full. The NPC will eat the food source and increase its food meter by 20 units to 60. The NPC's food meter is greater than 50 so it's not hungry anymore. This also means that the food source will be removed from the scene as well as in the NPCs memory list (forgetting) because it dropped to zero.

The NPC memory list will contain the locations of the food sources. The NPC will always try to remember the first item in the list and move its way down. The NPCs memory list structure will be either LIFO, FIFO, random, and closest for the entire run. LIFO (Last In First Out) will keep the most recent food source location in the memory list and the least recent in the back. When the NPC finds a food source the most recent will be in the first position of the list and the next food source found later will be inserted to the first slot and push down the other food source further down the list and will continue this until the run is over or the memory structure is changed. An example of LIFO is when the NPC finds x3,x2,x1 in that order, than the NPCs list will be x1,x2,x3 with x1 being the most recent food source it found and x3 being the least recent. The next memory structure is FIFO (First In First Out) and the first food source discovered in the list will always be the first in the list. The second food source will always be second in the list. A FIFO example is if the NPC found these food sources in this order: x2,x3,x4
than the list will be x2,x3,x4. Since x2 was the first found it will stay the first and try to
be the first food source the NPC will try to remember and move to. Random will "sort" or
"order" the NPCs memory list randomly. This occurs when the NPC finds a new food
source or comes across an old one. Closest sorts the memory list based on the location of
the NPC and the food sources in its memory measuring based on units. If the list was
x3,x4,x1 and the NPC was closest to x1 followed by x3 than the list will be sorted to
x1,x3,x4 with x1 being the closest to the NPCs current location and x4 being the farthest.
For all the memory structures any food source in the list that is zero is removed from the
NPCs list and the list gets pushed up.

Percentage recall 33%, 66%, 100% will be related to the NPCs ability to recall the
food source location. It will go down the list from the first slot to the last trying to recall a
food source location. For example, if the recall percent was 33% the NPC has a 33%
chance to remember the first slot in the list. If the NPC fails it will try the next slot in the
list. If the NPC tried recalling everything in the list and failed than it will enter the
explore state and move to a random node and try remembering from the beginning of the
list to the end again.

5.3 Recording Results

Throughout the entire run distance will be recorded in terms of units. The NPC
travels approximately three units a second. The distance will increment and update every
time the NPC moves during each frame. At the end of the run the distance will be
recorded to a file along with the memory structure, percent recall, and food left in the
scene. The food sources left in the scene will be converted to how many food sources did
the NPC consume. Distance and food sources left (food sources consumed) will be used
for the statistical analysis of the experimental hypothesis. Once the writing ends a new run will begin and this process repeats itself until the end of the fifteenth run under the current conditions. After that, new conditions will be tested for another fifteen rounds repeating this process until all conditions have been tested.

5.4 Implementing with Unity3D

Using Unity3D's MonoDevelop IDE this experiment was constructed with 9 classes. The game objects of the experiment are the NPC, food sources, nodes, and terrain. The NPC consists of both Unity3D's classes as well as my own. These classes are: transform, collider, mesh renderer, rigidbody, AI, and memory. The food sources had the exact same classes except rather than AI and memory it had the class food data. Nodes only had the transform class attached to it. The terrain had scripts attached to it as well to help keep track of the NPC and the food sources that were on it. The terrain had the transform, collider, and terrain grid classes attached to it. Some classes extended from MonoBehaviour [10] which is Unity3D's base class that allows the class to use Unity3D's functions and library. Figure 5.4-1 displays all the classes that I created and their purposes in this experiment.
<table>
<thead>
<tr>
<th>Class Name</th>
<th>Lines of Code</th>
<th>Functionality</th>
<th>Extended From</th>
<th>Component Attached To</th>
</tr>
</thead>
<tbody>
<tr>
<td>NPC</td>
<td>146</td>
<td>Creates the game object NPC</td>
<td></td>
<td>TerrainGrid</td>
</tr>
<tr>
<td>AI</td>
<td>2589</td>
<td>Controls the movement and state of the NPC</td>
<td>MonoBehaviour</td>
<td>NPC</td>
</tr>
<tr>
<td>Food</td>
<td>70</td>
<td>Creates the game object for food source</td>
<td></td>
<td>TerrainGrid</td>
</tr>
<tr>
<td>FoodData</td>
<td>190</td>
<td>Controls the data of food sources (food amount)</td>
<td>MonoBehaviour</td>
<td>Food Source</td>
</tr>
<tr>
<td>GridNode</td>
<td>480</td>
<td>Controls the data (adjacent nodes) for the nodes that the NPC moves on</td>
<td></td>
<td>TerrainGrid</td>
</tr>
<tr>
<td>IGridNode&lt;T&gt;</td>
<td>10</td>
<td>Interface for GridNode</td>
<td></td>
<td>PathFinding</td>
</tr>
<tr>
<td>Memory</td>
<td>474</td>
<td>Contains the memory structures and recall of the NPC</td>
<td>MonoBehaviour</td>
<td>NPC</td>
</tr>
<tr>
<td>PathFinding</td>
<td>144</td>
<td>Generates the path of the NPC</td>
<td></td>
<td>AI</td>
</tr>
<tr>
<td>TerrainGrid</td>
<td>864</td>
<td>Generates the terrain and the game objects (NPC and food sources)</td>
<td>MonoBehaviour</td>
<td>Terrain</td>
</tr>
</tbody>
</table>

Figure 5.4-1: Table of Classes.
6 RESULTS

Using SPSS the results of the experiment were calculated. The distance travelled by the NPC and number of food eaten were averaged based on the memory structure and recall percentage. The two dependent variables were distance travelled and food eaten. Table 6.0-1 displays the averages of the experiment variables for memory structures and recall percentages.

<table>
<thead>
<tr>
<th>Recall</th>
<th>Distance Travelled:</th>
<th>Memory Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Closest</td>
<td>Random</td>
</tr>
<tr>
<td>33%</td>
<td>3055.563 12.00</td>
<td>3104.788 11.269</td>
</tr>
<tr>
<td>100%</td>
<td>3421.763 12.200</td>
<td>3146.215 12.333</td>
</tr>
</tbody>
</table>

Table 6.0-1: Distance Travelled and Food Eaten Averages.

Table 6.0-1 shows that memory structure closest on 100% recall percentage had the highest average of distance travelled of 3421.763. Memory structure LIFO on 33% recall percentage had the lowest average distance travelled at 2584.019. The memory structure that consumed the most food sources on average was random on 66% recall and consumed about 13.350. The lowest average of number of food sources consumed was LIFO on 33% recall at 8.519.

The highest average distance travelled in the closest memory structure including all the recall percentages is 3421.763 at 100% recall, the lowest was 3055.563 with 33% recall. Random memory structure had up to 3350.373 distance travelled using 66% recall. The lowest distance travelled for random is 3104.788 at recall 33%. For FIFO the highest distance was 3138.603 using 66% recall and the lowest was 2804.252 with 100% recall. In LIFO the highest distance travelled in that group was 2806.163 at 100% recall the lowest distance in LIFO was 2584.019 with recall 33%.
The highest average of food eaten for closest is 12.875 at 66% recall and the lowest is 12.000 with 100% recall. For LIFO the highest was 9.813 at 100% recall and 8.519 at recall 33% for the lowest. In FIFO 12.063 was the highest average using 66% recall, 10.063 was the lowest using 100% recall. Random memory structure had 13.350 with 66% recall as its highest average and 11.269 with 33% recall for its lowest.

Table 6.0-2 shows the statistical significance tests of the hypothesis in the experiment. Significance provides an estimation of the test’s reliability or repeatability. If a test has a 0.05 or lower "Sig" value than the results are significant. This means that if the experiment were to run again similar results are to be expected. This provides a high confidence level in the results and prediction based on the result. If the "Sig" was above 0.05 than the results are not repeatable and therefore aren't reliable. In this report significant results will be bolded in the tables.
Table 6.0-2: Univariate Analysis.

Table 6.0-2 shows the univariate analysis of variance and the impact and importance of memory structures and recall percentages for distance travelled and food eaten. The table is grayed except for the relevant hypothesis tests. When it comes to memory structure it had a significant impact on distance travelled with $F = 8.422$ and $p < .01$. Memory structure also greatly impacted food eaten with $F$ being $11.865$ and $p < .01$. Percent recall did not affect distance travelled but had a significant effect on food eaten with $F = 3.11$ and $p < .05$. With memory structure and recall percent working together
(the interaction of memory structure and recall) there was no effect on both distance travelled or food eaten. Based on the analysis shown in Table 6.0-2 memory structure and recall percentage are both independent variables and their effects do not rely on each other in a way that would raise or lower the distance travelled or food eaten.

<table>
<thead>
<tr>
<th>Dependent Variable</th>
<th>Distance Traveled</th>
<th>Mean Difference (I-J)</th>
<th>Std. Error</th>
<th>Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Closest</td>
<td>LIFO recency</td>
<td>603.392688</td>
<td>123.073</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>FIFO primacy</td>
<td>318.584908</td>
<td>137.600</td>
<td>0.021</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>99.90</td>
<td>127.922</td>
<td>0.436</td>
</tr>
<tr>
<td>LIFO recency</td>
<td>Closest</td>
<td>-603.392688</td>
<td>123.073</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>FIFO primacy</td>
<td>-284.807780</td>
<td>129.498</td>
<td>0.029</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>-503.489329</td>
<td>119.165</td>
<td>0.000</td>
</tr>
<tr>
<td>FIFO primacy</td>
<td>Closest</td>
<td>-318.584908</td>
<td>137.600</td>
<td>0.021</td>
</tr>
<tr>
<td></td>
<td>LIFO recency</td>
<td>284.807780</td>
<td>129.498</td>
<td>0.029</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>-218.68</td>
<td>134.116</td>
<td>0.104</td>
</tr>
<tr>
<td>Random</td>
<td>Closest</td>
<td>-99.90</td>
<td>127.922</td>
<td>0.436</td>
</tr>
<tr>
<td></td>
<td>LIFO recency</td>
<td>503.489329</td>
<td>119.165</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>FIFO primacy</td>
<td>218.68</td>
<td>134.116</td>
<td>0.104</td>
</tr>
</tbody>
</table>

| Food Eaten         | Closest           | LIFO recency          | 3.11       | 0.594 | 0.000|
|                    | FIFO primacy      | 1.40                  | 0.664      | 0.037|
|                    | Random            | 0.11                  | 0.617      | 0.853|
| LIFO recency       | Closest           | -3.11                 | 0.594      | 0.000|
|                    | FIFO primacy      | -1.71                 | 0.625      | 0.007|
|                    | Random            | -3.00                 | 0.575      | 0.000|
| FIFO primacy       | Closest           | -1.40                 | 0.664      | 0.037|
|                    | LIFO recency      | 1.71                  | 0.625      | 0.007|
|                    | Random            | -1.28                 | 0.647      | 0.049|
| Random             | Closest           | -0.11                 | 0.617      | 0.853|
|                    | LIFO recency      | 3.00                  | 0.575      | 0.000|
|                    | FIFO primacy      | 1.28                  | 0.647      | 0.049|

Table 6.0-3: Multiple Comparisons for Memory Structure

Table 6.0-3 shows the significance between the memory structures split in two groups: distance travelled and food eaten. For distance travelled closest and LIFO had a significance of 0.00 which means that closest and LIFO differ significantly when it comes to overall distance travelled. Closest and FIFO had a 0.021 significance value.
LIFO and FIFO had a 0.029 significance level. LIFO and random are at 0.000. For food eaten closest and LIFO had a 0.000 value meaning there is a reliable (repeatable) gap between the amount food eaten between the two. Closest and FIFO has a significance value of 0.037. LIFO and FIFO had a value of 0.007 significance. LIFO and random were at 0.000. FIFO and random was at a 0.049 level.

<table>
<thead>
<tr>
<th>Dependent Variable</th>
<th>Mean Difference (I-J)</th>
<th>Std. Error</th>
<th>Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance Traveled</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33 % 66 %</td>
<td>-155.003</td>
<td>107.742</td>
<td>0.152</td>
</tr>
<tr>
<td>100 % 66 %</td>
<td>-185.833</td>
<td>111.276</td>
<td>0.096</td>
</tr>
<tr>
<td>66 % 33 %</td>
<td>155.003</td>
<td>107.742</td>
<td>0.152</td>
</tr>
<tr>
<td>100 % 33 %</td>
<td>-30.830</td>
<td>111.276</td>
<td>0.782</td>
</tr>
<tr>
<td>100 % 33 %</td>
<td>185.833</td>
<td>111.276</td>
<td>0.096</td>
</tr>
<tr>
<td>66 % 33 %</td>
<td>30.830</td>
<td>111.276</td>
<td>0.782</td>
</tr>
<tr>
<td>Food Eaten</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33 % 66 %</td>
<td>-1.11</td>
<td>0.520</td>
<td>0.034</td>
</tr>
<tr>
<td>100 % 66 %</td>
<td>-0.843</td>
<td>0.537</td>
<td>0.118</td>
</tr>
<tr>
<td>66 % 33 %</td>
<td>1.11</td>
<td>0.520</td>
<td>0.034</td>
</tr>
<tr>
<td>100 % 33 %</td>
<td>0.263</td>
<td>0.537</td>
<td>0.625</td>
</tr>
<tr>
<td>100 % 33 %</td>
<td>0.843</td>
<td>0.537</td>
<td>0.118</td>
</tr>
<tr>
<td>66 % 33 %</td>
<td>-0.263</td>
<td>0.537</td>
<td>0.625</td>
</tr>
</tbody>
</table>

Table 6.0-4: Multiple Comparisons for Percent Recall

Table 6.0-4 shows the significance between the recall percentages split into the same two groups: distance travelled and food eaten. The only significant value was with 33% and 66% with a "Sig" value of 0.034 in food eaten.

Figure 6.0-4 displays the average of the total distance travelled associated with each memory structure and recall conditions. Using Figure 6.0-4 the averages were calculated combining the averages of all recall percentages for that memory structure to create an overall average of distance travelled. For example, the closest memory structure overall distance travelled average included all averages for 33%, 66%, and 100% recall percent categories was 3268.961. Coming in second was random with an overall average of 3200.459 distance travelled. After random FIFO came in third with an overall average
of 2974.503. LIFO with an overall average of 2703.29 came in last with the least distance travelled of the four memory structures.

Figure 6.0-4: Average Total Distance for Memory Structures and Recall.
Figure 6.0-5: Average Food Eaten for Memory Structures and Recall.

In Figure 6.0-5 the amount of overall food eaten by memory structure is displayed. Much like Figure 6.0-5 the averages of all three recall percentages for each memory structure will be calculated. For example, closest using its values at 33%, 66%, and 100% averaged to 12.358 food eaten. This places closest the highest amount of food eaten over the other three memory structures. Random is second with a 12.317 average among its recall percentages. FIFO is third with an average of 10.938 and LIFO is last with an average of 9.282.
Figure 6.0-6: Average of Food Eaten with Recall Only.

Figure 6.0-6 shows the overall average of number of food eaten based on the recall percentage. This number contains all runs including all the memory structures that are under their respective recall percent. For example, all the food eaten averages for closest, random, FIFO, and LIFO will all be averaged and grouped under 33%, 66%, and 100%. The recall percentage that had the highest average for food consumed was 66% at 11.951 food eaten. The second highest food eaten average recall percent was 100% with 11.102 amount of food eaten. The last and the least amount of food eaten on average was 10.619 and it was 33% recall percentage.
Based on the results of the experiment some of the hypothesis that were made at the beginning were true and others were false. I wanted to begin this section of my report beginning with this hypothesis: "Memory structure and recall will be dependent on each other ". Results showed this wasn't the case. The calculations done by SPSS illustrated that memory structure and recall percentage had no repeatable effect on each other and that they could be interpreted separately or seen as independent variables. This means that recall percentage wouldn't have had an effect in conjunction with the memory structures in terms of food eaten and distance travelled and vice versa. This could have been due to what their roles in the experiment were. Memory structure controlled the order of the memory list and recall percentage was in charge of the chance of "remembering" where a food source was. If memory structure changed the recall percent, it will still operate the same. If recall percent changed, than memory structure order will continue to function the same as well. Both variables don't interact with each other in a way that would affect their functionality. Since memory structure and recall percent are independent variables, some of my hypotheses made were based on this hypothesis being true. This only changes what was being observed. Ideally I could have broken my hypotheses up by memory structure and recall rather than using them both as dependent on one another or even completely left them out. With this said I will continue on my discussion with the hypotheses in the order they were presented.

The first hypothesis that was stated was that "The combination that will result in highest distance travelled will be closest on 100% recall". With the results concluded this is true but it wasn't by a large amount. Closest on 100% recall averaged 3394.389 and
random on 66% recall was at 3350.373. It was probably due to the NPC always recalling where the closest food source is based on its current location which allows it to have a quick meal so it can build up its food meter to find newer food sources to eat from and therefore travel farther.

The combination that resulted in highest amount of food eaten based on the results was random on 66% at a total average of 13.350 food eaten. The hypothesis assumed that closest on 100% recall was going to provide the highest amount of food eaten. This hypothesis ended up to be false. Closest on 100% had an average of 11.900 compared to random on 66% with 13.350 food eaten. This may have been because of the randomness of the travelling. The NPC would randomly eat and discover more food sources on the way to its destination which could traverse across the entire terrain. This would cause the NPC to consume more food sources to keep it alive and diminishing multiple food sources on its way to its destination. The NPC may have more food sources in its memory to eat from and therefore eat more due to randomly encountering other food sources on its way to a random food source in its memory. Closest might have more local travel and will guide the NPC to the closest food source possibly preventing it from discovering newer food resources on the way to its main destination and therefore eating less food.

The hypothesis that predicted that FIFO on 33% with average distance travelled of 2980.653 will provide the lowest distance travelled was incorrect. The results showed that the lowest average distance travelled was 2584.019 with LIFO being the memory structure and 33% was the recall percentage. For food eaten the combination that resulted in the lowest amount of food eaten was LIFO on 33% recall with 8.519 food eaten. This
result opposes the hypothesis that the combination of FIFO with 33% recall with an average of 10.688 food eaten would have the lowest amount of food eaten. Both these results might have been due to the positioning of the NPC when it gets hungry.

On LIFO the NPC will go to the most recent food source which generally means that it goes to the closest (most recent) which is why positioning is important. The NPC will go to the same recent food source repeatedly while only traveling a minimal amount or in the local area until it returns to the same resource. This might lead to the NPC consuming all local (based on its current position) food sources because it keeps going to the most recent ones it found. After consuming all the local resources it may not have found any newer food sources to go to or the food sources are too far for the NPC to get to and eventually die before it gets there. FIFO should operate the opposite way. The first food source the NPC finds is the first source it goes to. After exploring, this may cause the NPC to travel farther to its destination food source (compared to LIFO's local food source area) as well as discover other newer food sources to prolong its life and cause it to travel longer and eat more.

The hypothesis that "closest should provide the highest amount of distance travelled followed by LIFO, random, and FIFO" was incorrect according to the results of the experiment. Based on the results the order in highest average distance travelled first is: closest, random, FIFO, and LIFO. Since the order of the hypothesis compared to the results order isn't exactly the same, the hypothesis is incorrect. Closest matched both the hypothesis and the results. This was probably due to the NPC always finding food as soon as possible which will prolong its life. The longer the NPC lives the more distance it will travel. The next highest distance travelled in the hypothesis was LIFO. In the results
it was random. A possible reason why random had a higher distance travelled over LIFO could have been due to the amount of food the NPC had to consume compared to the food it gains. When the NPC selects a random food source out of memory than there is a higher chance it may select a closer food source without sacrificing more food than it gains. The third and fourth in the list for the hypothesis was random and FIFO, in the result it was FIFO and LIFO. A possible explanation why FIFO came in third in the results and why it wasn't random like it was originally hypothesized could have been the amount of discovery of food sources when it travels back to the first food source it found. Once it gets hungry it may stumble across these food sources again. This gives the NPC a more efficient way of eating by spreading it more evenly among the food sources. After it exhausts the first food source it found, the NPC can use the food sources that it discovered from going to the first food source and consume those.

For recall percentage and average distance travelled it was hypothesized that 100% would have the highest distance travelled following 66% and 33% with the least amount of distance travelled. Calculating the average including all memory structures between recall percentages, 66% did have the highest average of distance travelled. But this result isn't reliable according to Figure 6.0-2 which shows the result wasn't significant between recall and distance travelled. There was significance between food eaten and recall percentages so I will be discussing that instead. Since recall percent and food eaten were significant the results showed 66% recall had the highest amount of food eaten, following 100%, and than 33%. 66% recall had one more food eaten than 100% on average. When the NPC fails to recall a food source it moves on to the next and so on until it recalls a food source or runs out of food sources in the list to recall. For memory
structures such as FIFO and LIFO this may be a positive thing to have. If there is a 66% chance of recalling a food source than that means there is a 34% of failing to recall one. If the first food source happens to be the farthest away from the NPC and the second food source in the list is much closer to the NPC, a failure to recall the first food source maybe a good thing. This may prevent the NPC from using up more of its food meter for little gain. This is also why the NPC is consuming more food at 66%. In times where the NPC should have died if it attempted a long trip to a low unit food source, it went to the more closer one keeping it alive to consume and exhaust more food sources.

The hypothesis that there will be a correlation between distance travelled and food eaten ended up to be true. The results showed that the more food that was eaten the higher the distance travelled by the NPC. This supports that the more the NPC ate the longer it lives and the longer it lives the more distance it travelled.
8 CONCLUSION

This experiment has proven to be a challenging and enlightening experience. It helped refine my skills in running experiments such as identifying variables to test and organization. It also helped in analyzing results and how to interpret them. Learning how to program using a gaming engine was also a bonus while working on this project. This experiment utilized many components to implement, test, and analyze. It all started with a hypothesis, a design, and the use of Unity3D.

The tools of Unity3D such as: scenes, game objects, components, assets, and prefabs were the building blocks of this experiment. Utilizing these tools with the drag and drop system and the views interface, make a more intuitive and simpler environment to create this experiment. The drag and drop system allowed prefabs and game objects to be dragged directly into the scene reducing any unnecessary code. The views: console, scene, project, hierarchy, inspector, and game add to Unity3Ds ease of use by providing quicker ways to access objects such as the food sources. The elements used in this experiment involved many triggers and colliders which were key to the interactions of the NPC with the food sources. The terrain contained a camera that was positioned to provide a top view of the terrain. This allowed maximum view of the entire experiment to track where the NPC was going and made it easy to see what was happening in the experiment. Time was used in the experiment to make the project frame rate independent as it was running. This caused both high end and low end systems to execute triggers and events at the same time rather than by frame. When the experiment begins, everything is loaded into the scene and is executed in this order: awake, start, fixed update, update, late update, and co routines. With all these tools the experiment was created.
This experiment was to test the life expectancy of the NPC based on the type of memory structure and recall percentage the NPC was running. To determine how long the NPC lived, total distance travelled and food eaten was used as a measurement. The hypothesis that was created at the beginning of this experiment answered multiple questions that lead to a core conclusion on what combination or condition will keep the NPC alive the longest. Memory structures LIFO, FIFO, random, and closest in combination with 33%, 66%, and 100% recall were used to conduct this experiment. This experiment was broken up into twelve conditions such as LIFO on 66% recall and each condition was ran at least fifteen times. During a run the NPC will randomly select nodes to walk to and on these nodes there potentially can be a food source for the NPC to discover. The food meter for the NPC will decrement by 1 each second. Once the initialization process is complete then the scene can start the experiment. The NPC has one of three states it can be in at one time: explore, forage, and death. When running the experiment the NPC starts in explore. During this state the NPC chooses random nodes in search of food sources. As the NPC encounters food sources it places it into its memory list. This memory list is controlled by the memory structure and recall percent which are the variables that is being tested. As the NPC explores, its food meter decrements and when it reaches 50% it enters the forage state. This state is when the NPC uses its memory list to look for food it has already found. Depending on the memory structure the first food source the NPC will try to go to can be the most recent, the first one it initially found, the closest, or a randomly selected food source in its memory list. At the same time the chance the NPC will recall the food source could either be 33%, 66%, and 100%. This is what will test the amount of distance the NPC will travel as well as food
eaten against each condition. Eventually the NPC's food meter will reach zero. This is when it reaches the death state. Once the NPC dies the results are recorded to a file and another run can begin. The last process is to record the results of the run. This saved a lot of micro managing for this project because the project itself runs on its own. Once all the data was gathered for all the runs of each condition it was time to analyze the results.

The results that were gathered from this experiment provided interesting data. Some of the results stuck out more than others. The first result that surprised me was that memory structure and recall percent were both independent variables. They didn't interact with each other in terms of distance travelled or recall percent. Some of my hypotheses used memory structure and recall percent as dependent variables. I knew I couldn't change the hypotheses due to this result so I continued on using the same hypotheses regardless of this result. What I learned from this result is that I should never make a hypothesis that is based on another hypothesis being true.

The next piece of data that I found interesting was that in almost all recall conditions (excluding recall 100% where they are almost even) FIFO had a higher average than LIFO when it came to distance travelled and food eaten across all recall percentages. I thought this result was important because in this experiment I thought most recent usually meant most closest. If food sources were close, the NPC would usually have something to eat and live longer but for this experiment this didn't happen. This further supports that the type of memory structure will affect NPCs in a game. If I turned this experiment into a game, LIFO would be used for the "easy" difficulty mode of the game.
Another interesting result was related to the correlation of distance travelled and food eaten. The order was always closest, random, FIFO, and LIFO from highest to lowest for both distance travelled and food eaten. Based on the overall averages this is true. But based on my observation the list could very well be (closest = random = FIFO) > LIFO for distance travelled and closest = random > FIFO > LIFO for food eaten. I'm using 66% recall as the average in my observation since it's the average of all the recall percentages. I wanted to use 66% to show that there is always some "forgetting" when it comes to memory structures. I would say that even though the type of memory structure maybe different the results can be very similar. This doesn't mean that the type used isn't important in a game but you may be left with a trade off to think about. Option 1, is to use the memory structure that causes the NPC to function flawlessly in your game but has higher system requirements reducing the amount of systems that can run the game. Option two, is to use the "decent" memory structure that uses less resources but allows more systems to run the game. Based off my results closest memory structure is option 1 and random is option 2. Closest did have better results than random but random was quicker because it didn’t have to sort itself like the closest did. If this experiment was a game I would most likely make the trade off of option 2 (random) to allow more users to play my game.

The result that I find surprising and puzzling is that 66% recall had the highest average of food eaten. This was very unexpected on why forgetting locations of food sources causes the NPC to consume more food. This result could lead to a more ideal recall percent between 100% and 66% that will consume even more food and increase NPC life. It also could mean that the first choice isn't always the best choice. I noticed
that during this experiment a decision to go to the closest was the worse decision the NPC could have made because that food source contained a minimal amount of food. The second food source in the NPC's memory structure had the maximum amount and it was only 2 to 3 nodes away. On 100% the NPC would probably have died and consumed less food sources because it recalled the low food source location where as 66% could have recalled and gone to the higher food source location increasing life and therefore increasing more food eaten. In a game this can be used to add more realistic recall for an NPC. If the NPC can't recall most of the locations a player to track down, it can provide for dull game play. In this experiment "forgetting" instructions 34% of the time created more realistic and challenging game play among the other two recall percents. With all this said, the reason why 66% had more food eaten is still a question I'm not sure how to answer.

This project can be used to construct future projects and modified for other memory structures and recall percentages. It can be used with custom memory structures or existing ones and recall percentages between 0% to 100%. It can also be modified to change the amount of food sources that are on the terrain. The decrease rate of the NPC's food meter can be changed as well. Some other things that can be done to this project could involve resizing the terrain for a bigger scene or change the speed or the movement of the NPC. It is also possible to run multiple NPCs too. Most of these changes can be done by just changing a few variables and some code in the project.

This experiment was a learning experience from the idea to the design and implementation. Whether it's for the tutorial on Unity3D, the data collected on this experiment, or the program itself the reusability of this project can lead to future
improvements or other experiments. I hope it may be of use or informative to anyone else that comes across this project or has any interest in developing an experiment.
BIBLIOGRAPHY


<http://www.w3schools.com/js/>.


<http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Non-player_character.html>.


