2003-09

Algorithmic approaches to finding cover in three-dimensional, virtual environments

Morgan, David J.

Monterey, California. Naval Postgraduate School

NAVAL
POSTGRADUATE
SCHOOL

MONTEREY, CALIFORNIA

THESIS

ALGORITHMIC APPROACHES TO FINDING COVER IN THREE-DIMENSIONAL, VIRTUAL ENVIRONMENTS

by

David J. Morgan

September 2003

Thesis Advisor:	Christian J. Darken
Second Reader:	Joseph A. Sullivan

This thesis done in cooperation with the MOVES Institute

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2003	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Algorithmic Approaches to Finding Cover in Three-Dimensional, Virtual Environments		**5. FUNDING NUMBERS**	
6. AUTHOR(S) Major David J. Morgan, U.S. Army			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**	
11. SUPPLEMENTARY NOTES: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited		**12b. DISTRIBUTION CODE**	

13. ABSTRACT (maximum 200 words)

In order for an agent to be credible in simulating a human opponent in a first-person combat simulation, it must be able to find and use cover from direct fire weapons. The ability to find cover is fairly intuitive for humans, but current attempts at replicating this ability in computer simulations and video games have been either simplistic or totally missing. This thesis explores a range of algorithms which computer agents can use for finding cover from direct-fire weapons in high-detail, dynamic, three-dimensional environments. The first method treats the enemy as a point light source and uses binary space partition trees to create shadow volumes to find areas of cover. The second method uses a depth-mapping technique to find potential areas where the agent could get behind cover. The third method uses a sensor grid centered on the agent that allows it to check the area around it for cover locations. We implemented the sensor grid technique inside of the first-person shooter computer game America's Army: Operations as a proof of concept.

14. SUBJECT TERMS COVER, CONCEALMENT, AGENTS, REACTIVE AGENTS, VIRTUAL ENVIRONMENTS, SIMULATION, ARMY GAME PROJECT, BINARY SPACE PARTITION TREES, DEPTH MAPPING, SENSOR GRID			**15. NUMBER OF PAGES** 112
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

ALGORITHMIC APPROACHES TO FINDING COVER IN THREE-DIMENSIONAL, VIRTUAL ENVIRONMENTS

David J. Morgan
Major, United States Army
B.S., United States Military Academy, 1991

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN MODELING, VIRTUAL ENVIRONMENTS AND SIMULATIONS

from the

NAVAL POSTGRADUATE SCHOOL
September 2003

Author: Major David J. Morgan, U.S. Army

Approved by: Dr. Christian J. Darken
Thesis Advisor

Commander Joseph A. Sullivan, U.S. Navy
Second Reader/Co-Advisor

Dr. Rudolph P. Darken
MOVES Academic Committee Chair

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

In order for an agent to be credible in simulating a human opponent in a first-person combat simulation, it must be able to find and use cover from direct fire weapons. The ability to find cover is fairly intuitive for humans, but current attempts at replicating this ability in computer simulations and video games have been either simplistic or totally missing. This thesis explores a range of algorithms which computer agents can use for finding cover from direct-fire weapons in high-detail, dynamic, three-dimensional environments. The first method treats the enemy as a point light source and uses binary space partition trees to create shadow volumes to find areas of cover. The second method uses a depth-mapping technique to find potential areas where the agent could get behind cover. The third method uses a sensor grid centered on the agent that allows it to check the area around it for cover locations. We implemented the sensor grid technique inside of the first-person shooter computer game America's Army: Operations as a proof of concept.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

There several people that I must thank for their support of my efforts in researching and writing this thesis:

First, and foremost, my wife, Mi, for putting up with the late nights, giving up a lot of our time together and always encouraging me. Thanks for putting up with me and still .

Chris Darken for being the one person besides me, who was excited about this topic from the very beginning and who saw great potential in the research. Thank you for all the great, thought-provoking discussions we had about theory and implementation of these algorithms.

Rudy Darken for teaching great classes that provide excellent background for this kind of research. Thank you for a great learning experience.

Joe Sullivan for providing great support and training simulations. Without you life in the lab would not have been near as much fun.

Finally I'd like to thank Christian Buhl and Greg Paull, from the Army Game Project, for answering a million and one questions about America's Army: Operations. Without their help, I'd still be searching through the code trying to figure out how to make it work.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. THESIS STATEMENT

A range of algorithms exists that can be used to improve the current ability of agents to find cover from direct fire in dynamic, three-dimensional, dynamic, simulated environments.

Agents in current three-dimensional, dynamic, simulated environments lack the ability to take cover from direct fire due to a lack of information about their immediate surroundings. A range of algorithms exists that can provide an agent with additional information about its surroundings, making it possible to produce more realistic behavior.

B. MOTIVATION

The United States Military has and will continue to increase its use of simulations to train its soldiers and to perform operations analysis. While simulations cannot completely replace live training currently, they do offer some significant benefits. Simulations are less expensive than live training overall when all associated costs are included. Simulations are safer than live training, especially in force-on-force situations where two live units actively fight each other. Simulations do not have the environmental impact of live training. Simulations also allow for training in a controlled environment where the conditions of the exercise can be exactly controlled and repeated as many times as it is necessary to reach the trainer's goals. For these reasons and more, the United States Military has turned to simulations in order to maintain the training level of its units and soldiers.

In many cases, it is preferable to use computer-controlled agents to act as the enemy forces in simulations rather than place them under the control of another person. Many times the hardest part of a military task is coordinating actions between units to get combined effects on the target. If you have two units to train, computer-controlled agents can allow you to train them together rather than opposing each other. This also allows you to run higher-level

scenarios. Instead of eight live simulators opposing eight live simulators, you can have sixteen live simulators opposing sixteen computer-controlled agents. This is a more effective use of simulation assets, which may be in high demand.

In order for computer-controlled agents to provide a positive training effect, they must look and behave as closely as possible to their real-life counterparts. If the behavior of the agent is significantly different from real-life, the trainee, in essence, is training on a different task from the one that they are supposed to learn. While there may be some cross-over in related skills, it is unlikely that they will be able to reach full proficiency in the target task.

One of the areas where current computer-controlled agent behavior is unrealistic is in the ability for them to take cover when fired upon by direct fire weapons. The basic ability to hide and take cover is something that everyone learns as they grow up. The military builds on this knowledge and trains its soldiers even further how to find and use cover. It is an essential survival skill on modern battlefields dominated by projectile weapons. However, in most current simulations and games, the computer-controlled forces have only a simplified understanding of cover or none at all. This leads to computer-controlled opponents that stray out into the open making themselves easy targets. This does not provide the trainee with the necessary tough, realistic training.

In order to provide effective training in computer simulations, the computer-controlled agents should be able to find and take cover like a human would in the same situation. We can do this through a variety of methods that increase the agent's knowledge of the world around him. That knowledge can then be used to produce believable behaviors.

C. THESIS ORGANIZATION

The remainder of this thesis is organized as follows:

- **Chapter II: Description of the Application Area.** Detailed description of cover and concealment, an overview of some of the processes involved in how we find cover, and information on computer processes useful in finding cover.

- **Chapter III: Related Work.** Some of the techniques currently used in military simulations and computer games for finding cover.
- **Chapter IV: Conceptual Models.** Descriptions of the three algorithms that we developed in order to provide an agent with knowledge of cover around it. These are the Binary Space Partition Shadow Volume Method, the Depth Mapping Method, and the Sensor Grid Method.
- **Chapter V: Implementation of the Sensor Grid Model.** A detailed description of how we implemented the Sensor Grid Method in the computer game America's Army: Operations.
- **Chapter VI: Conclusions and Future Work.** A discussion of general conclusions of all the methods we researched and suggestions for future research to improve them.

THIS PAGE INTENTIONALLY LEFT BLANK

II. DESCRIPTION OF THE APPLICATION AREA

A. INTRODUCTION

This chapter provides a description of some of the issues that are important when developing cover algorithms for use in computer simulations. First we must understand what cover is and how it is different from concealment. Second, we look at human performance issues related to the issue of how we are able to find cover in real life. Finally, we must consider the tools available in computer simulations to help our agents find cover and the limitations of these tools.

B. COVER AND CONCEALMENT

1. Cover

Cover is a physical object that can prevent a weapon system from causing damage to you by deflecting or absorbing its energy [8]. Common examples of real-life cover from direct-fire weapons include rocks, trees, earthen berms, and solid walls. A good piece of cover should be large enough for you to get your entire body behind and should be able to protect you from the weapon being fired at you.

An object that provides cover for one weapon system may not provide cover against another one. For example, a cinderblock wall provides good cover from most small arms fire from rifles. However, it will provide no cover against heavy machine guns that are able to fire right through it. In this case, the cinderblock wall would only provide concealment. In order for cover to be effective, it must be able to stand up to the force exerted on it by the weapon.

Interestingly enough, several feet of packed earth provides better protection against bullets than rock does. When bullets strike a sufficiently large amount of earth, their impact is absorbed and they stop. When bullets strike rock they tend to ricochet and also fracture the rock sending small pieces of secondary shrapnel flying all over the place. The ricochets and the flying pieces

of rock can wound you or other people near you. If hit with sufficient force, rock can also crumble to the point where it provides no cover at all.

Cover is also direction dependant. As the position of the weapon system changes the area of cover provided by an object also changes. Depending on the shape of the object and the position of the weapon, the object may provide no cover at all. If you take cover behind a log from someone firing at you from the ground floor of a building, you may not have any cover if that person moves up to the second or third floor where they can see over the log.

2. Concealment

Concealment is anything that has a negative impact on the ability of someone to accurately target you with a weapon system by affecting their ability to see you. Common examples of concealment are brush, tall grass, smoke, and fog. These are all objects that you can place between you and a person firing at you to keep them from accurately targeting you. Less common examples of concealment are shadows and personal camouflage. In these cases, there is no physical object between you and the person firing at you, but they can still prevent the enemy from accurately targeting you.

3. Cover Versus Concealment

Cover and concealment are closely related to each other. The vast majority of objects that provide cover will also provide concealment. Generally, objects that are large enough and strong enough to stop a bullet will also prevent the enemy from seeing you. However, the opposite is not generally true. Objects that provide concealment do not always provide cover. It is a common saying in the U.S. Army that “cover provides concealment, but concealment does not provide cover”. This makes it very easy for most people to keep them straight.

The only notable example that we were able to find of an object that provides cover, but not concealment, is bulletproof glass. Bulletproof glass is specifically designed to stop bullets and allow you to see through it. Even so, it can only stop bullets up to a certain size and for only a certain number of shots before it ceases to provide protection.

4. Cover from Other Types of Weapons

One important distinction in types of cover is the difference in cover from direct-fire weapons and indirect-fire weapons. Direct-fire weapons are generally designed to cause damage by directly striking the target. Their energy travels over a relatively flat and narrow trajectory. Examples of direct-fire weapons are rifles and lasers. In order to take cover from this type of weapon, you want to place the object providing you cover between you and the person firing at you. This is different for indirect-fire weapons and area-effect weapons.

Indirect-fire weapons are those that do not follow a direct path to the target. They generally travel in a high, arching trajectory that takes them over intervening objects to attack the target from above. Examples of common indirect-fire weapons are artillery shells, mortars, and bombs. In order to take cover from indirect-fire you need to put the cover-producing object between you and the flight path of the indirect-fire weapon. This is what the military calls “overhead cover”.

Since indirect-fire weapons usually have a very low chance of directly hitting their targets, they generally have explosives in them that let them cause damage over an area. In order to take cover against area-effect weapons, the position of the enemy is not as important as where round lands. This is where the damage will be coming from. If a hand grenade lands behind you, you need to have cover between you and the hand grenade, not between you and the enemy that threw it. In many cases with indirect-fire weapons, you will know that you are under attack, but not know where the round will land. In this case, your best bet is to find a location that provides cover from the largest number of probable landing spots.

C. HUMAN PERFORMANCE

1. How Humans Find Cover

In order to program an agent to find cover, we first had to look at how humans find cover. Unfortunately, there is not a lot of existing research in this area. Even basic military training manuals assume that a person already has an understanding of how to take cover. They only list several things that can

provide cover and tell you to protect yourself from enemy fire by using cover. In order to determine how humans actually find cover, we relied on our experience in the military and common sense.

A person wishing to take cover generally follows these steps:

- Determine where the threat is coming from.
- Look for objects or features that may provide cover from the threat.
- Determine if these locations can be reached.
- Choose a place to take cover.
- Move to that location.

Due to the parallel nature of brain processes, these steps do not have to execute sequentially. In fact, they may not always occur in this order and at times, some of the steps may be left out. What we are attempting to list here are the minimum, necessary, logical steps to find a complete solution to the cover problem. There may be shorter and quicker solutions, such as simply dropping to the ground, but they will not guarantee that you are in cover, if cover is available.

The steps listed above may seem like a lot to think about when your life may be in danger, but humans seem to be able to do it all in a fraction of a second. In addition, there may be a significant amount of preprocessing going on when a person knows that they are in a dangerous situation where they may need to take cover.

As we move around, we continuously create a mental model of the area around us. This is why we can do things like back up without looking behind us or walk up stairs without looking at our feet for every step. We use our mental model of the surroundings to plan our interaction with the environment. This mental model is not as high resolution as a deliberate scan and interpretation of the area, but it allows us to take quick action when necessary.

Experienced infantrymen also tend to plan their use of cover ahead of time. While out of direct combat with the enemy, they will note good locations to take cover as they move along. When moving while in direct combat with the

enemy, they will plan short moves where they limit their exposure before taking cover at a new location.

a. Determine the Direction of the Threat

In order to make the most effective use of cover you must know the direction that the threat is coming from. If you directly observe the threat, then this is very easy. If you do not directly observe the threat, then you must analyze secondary factors to determine the most likely position of the enemy. Some of these factors include: sound, impact of rounds, and previous intelligence on possible enemy locations. In the absence of any information at all on the direction of the threat, we tend to pick the piece of cover that offers the best protection from the widest area. Another option in this situation is to move back to the area where we came from under the assumption that we must have just come into view prompting the enemy to fire.

b. Determine Which Positions Provide Cover

Once the position of the threat has been determined, you must determine which objects provide cover from that threat. There are actually two steps in this process, determining the area of cover offered by the object and determining if the object provides cover or concealment. There is no clear evidence of the any order for these two steps.

Determining the area of cover offered by the object involves mental simulation. Through mental simulation, you project lines from the threat location to the object. These lines allow you to determine the area on the far side of the object from the threat where there will be cover. Fortunately, humans are fairly adept at doing this type of mental simulation [14]. It has been an important survival mechanism for us to be able to keep a three-dimensional map of our surroundings in our heads. In military terms, this is sometimes referred to as “situational awareness”. It is an understanding of the area around you, your position in that area, and the potential things that can affect you in that area.

Determining whether an object offers cover or concealment against a threat is a matter of learning and experience. You must make a quick judgment about the composition of the object to determine if it is sufficient to stop

the threat. During military basic training when recruits first learn to take cover, they often make poor choices. They correctly move themselves out of line of sight from the threat, but they try to take cover behind objects that provide little or no real protection. After becoming more familiar with the capabilities of different weapon systems, they are much more adept at quickly determining good cover locations.

c. Determine Which Positions Can Be Reached

Once you have determined which positions provide cover, you must determine which ones are reachable and how long it will take to reach them. A cover position that you cannot reach or takes too long to reach will not be useful.

d. Chose a Cover Position to Take

In order to choose the best cover position, you must consider many factors. You will have to make this decision very quickly and under extreme duress. Your life may well depend on it.

You must consider the quality of cover provided by the object or feature providing cover. You must determine if the object is sufficient to stop the threat from which you are taking cover.

You must consider the size of the area of cover provided by the object. A larger area provides you more maneuverability and is easier to get behind. A larger area decreases the chance that you try to take cover and find that you cannot fit yourself completely inside the area. The ability to move within the currently covered area also means that you have more room to adjust if the threat changes position.

A good cover location will also have a good egress route. A good egress route provides you with a way to move out of the immediate area with continuous cover from the threat. Taking cover behind an object without a good egress route will can leave you pinned down in that position and unable to move without taking serious damage. When this happens, the enemy is often able to move to another position where the object provides you no cover at all.

You must consider how long it will take you to reach the cover location. You want to minimize the amount of time that you expose yourself to the threat. Often this is a simple matter of distance to the location. However, difficult terrain can slow your movement and have a huge impact on the time it takes to reach the position. A good cover location must be quickly reachable.

e. Move to the Cover Position

The last step in taking cover is to move to the cover location you chose. The key here is to minimize your exposure to enemy fire. Sometimes this means you should move as quickly as possible. Sometimes this means you need to drop to the ground and crawl into the cover. The situation will dictate which method you should use.

2. Taking Cover from Multiple Attackers

In most of the situations we have described above, we have talked about taking cover from one threat. However, it is a more common situation in combat that you will be facing multiple threats at the same time. These threats may be all in the same direction or in several directions. One solution is to analyze the cover from each threat and find the intersection of these locations. While this will provide you with the best answer, the computational expense could be enormous. A better solution may be to analyze the cover from the threats that are on the far ends of the group. We believe that this a reasonable approach whenever all of the threats are in the same general direction, but we can give no guarantee that it will work in all cases.

3. The Bottom Line

In the end, a good choice made quickly is often better than the perfect choice that takes too long. There is no way to accurately estimate the amount of time between your perception of the threat and when it is able to effectively engage you. With modern weapon systems, the life expectancy of a fully exposed target is very short. The only prudent course of action is to assume that you have no time available and make the initial decision as quickly as possible. After you make your initial decision and you have some cover, you can take the time to look for better locations.

D. FINDING COVER IN COMPUTER SIMULATIONS

1. Checking Line of Sight

Most simulated environments have some sort of line of sight checking built into the system. While we call it a “line of sight" check, we are actually determining if a line drawn between any two points in the simulated environment intersects anything. One of most common uses for this check is to determine if one entity can see another one (thus the name), but programmers also use it in a wide variety of applications including navigation and determining projectile impact points.

The results returned by line-of-sight functions vary greatly from program to program. It is important when designing your system to know all of the information that it can provide and then use that to your advantage when developing your cover algorithms. Some of the information returned may include:

- True / False answer indicating if an object was intersected
- Coordinates to the point of intersection
- Surface normal of the point of intersection
- A reference to the object intersected

Some systems go even further and offer a set of functions that trade information for speed: A fast trace that just returns “True” or “False”, a medium speed trace that returns information on the first intersection, and a slow trace that returns information on all intersections along the line segment.

Checking line of sight requires a significant amount of computational effort on the part of the computer and programmers always try to minimize the use of line of sight checks in real-time simulations. The process normally involves several dot products, cross products, and comparisons for each polygon tested. Programmers use various culling algorithms to limit the number of polygons in the environment that are actually tested. However, typical environments can be composed of several million of triangles, which can be difficult to process quickly.

One problem with most line of sight checks is that they do not accurately determine if an object can be seen. The location of an object in an environment

is most often recorded as a three-number vector. If we check line of sight to this location and the result says that we can see it, then we can definitely see the specific point on the object. However, if the result says that the line of sight is blocked, this does not necessarily mean that we cannot see another part of the object. Other portions of the object may be plainly visible.

Additional line traces can increase the probability that the object is completely out of sight, but this technique is still prone to error and may not be computationally efficient. For example, you could use the four corners of a bounding box around the object to perform additional checks. If all of the checks say that line of sight is blocked, then you can be reasonably sure that the position gives the object cover. Then you might have the opposite problem. The object could be completely in cover, but a corner of the bounding box sticking out in the open. In this case, the function would report that line of sight existed to the object, when it really does not. On top of this, you have just increased your requirement for line of sight checks by five times; one for the location of the object plus four more for the corners of the bounding box.

Even with all of their problems, line-of-sight checks provide a useful approximation for determining if a threat can see a target. It may be the case that the only way to make a more accurate system would be to implement some form of a computer vision system. While line-of-sight checks are computationally expensive, they are much less expensive than even primitive computer vision systems. They key is to tune your system to get a good balance between computational performance and task performance.

2. A Computer Agent's View of the World

An agent views the world through a set of sensors. By combining this information with internal logic, perception, and previous knowledge, it creates an internal representation of the world. Generally, this is not a complete representation of the world due to the amount of storage space required. It is debatable what level of representation humans retain in their memories. Often the agent processes the detailed information and keeps a summarized version on hand. It only keeps enough information on hand to complete its task.

For an example of what information is available to a typical computer game agent we will use the computer game Unreal Tournament [22]. The computer agents that you compete against in the single player version of Unreal Tournament are commonly called “Bots”. Unreal Bots are able to navigate through their world and interact with it in all the same ways that players can. The Bots provide challenging game-play for players and exhibit a range of behaviors that are very similar in some ways to that of human players in multi-player games.

The behavior of Unreal Bots is controlled by state code [21]. The code that controls the Bot is separated into sections called states. Execution continues inside of the current section until something happens that causes execution to jump to another state. Some functions may be redefined inside of certain states so that their execution becomes state-dependant. That is, the current state of the Bot determines which version of the function is executed when it is called. This produces behavior that depends on the current state of the Bot as well as the actual stimulus that it receives.

The Bot starts in a default state that ensures everything is initialized correctly and then proceeds into other states based on its goals, perceptions, and state transition code. For example a Bot could start in a “Hunting” state where it would travel around the map looking for players to fight. Once it sees a player, it will decide whether it should attack the player or flee. If it decides to attack, it will move to one of its “Attack” states and engage the player. If the player runs away the Bot will chase them. If it takes too much damage, it may decide that it needs to run away and it will enter a “Flee” state. In the “Flee” state it will move away from a player and attempt to find objects that it can pick up to heal itself. This seems like a robust set of behaviors, but the agent produces these behaviors with a bare minimum of information about the actual environment around it.

An Unreal Agent has no idea what the world around it actually looks like. It has no information about the size or shape of the rooms. It does not know

anything about the vast majority of the objects in the environment. When it needs something like a First Aid packet, ammunition, or a gun it queries the game engine and stores the result.

Unreal Bots navigate through their world by using a waypoint graph. Waypoints are placed throughout the environment by level designers. The waypoints are connected by links that tell the system which waypoints can be reached from which other waypoints. The links between waypoints are directional. For two waypoints to be connected by a link, they must be within direct line of sight of each other and a Bot must be able to travel directly between them. The series of waypoints and the links between them form a navigational graph that allows the Bot to move around the environment. However, the Bot does not keep even this information internally. The game engine keeps track of the actual waypoint graph. When the Bot wants to go somewhere, it sends a request to the game engine, which in turn tells it which direction to move.

An Unreal Bot knows about the player through a reference. When a player comes into line of sight of a Bot the game engine sends the Bot a "seePlayer" message that includes a reference to the player. The Bot can then use this reference to gather additional information about the player, such as their position, health, and weapons they carry. The Bot does not retrieve any of this information until it is needed. When the player moves out of line of sight the Bot receives another message telling it that it can no longer see the player.

In order to take cover the Bot will need more information about the environment.

3. Penetration of Objects

Most computer simulations do not accurately represent the penetration of objects by projectiles. Most often whether an object stops weapons fire or not is a True/False Boolean value stored somewhere with the object. In real life the penetration of an object by a projectile is affected by a large number of factors including the material the object is composed of, the thickness of the material,

the type of projectile, and the angle of incidence. Whether or not an object can be penetrated by a weapon has a direct impact on the quality of cover it provides.

Different materials behave differently when struck by a projectile. Soft materials may absorb the energy of the projectile, while hard materials may reflect it. Some materials may deform when struck while others may not. A brittle material may shatter when struck. These all can have an effect on the resistance offered by the material and the effects of repeated shots against the material.

The type of projectile has a major impact on whether or not it penetrates an object. An object that stops regular bullets well may offer little protection against armor piercing rounds that are designed to have greater penetration. Against exotic weapons such as lasers, some materials may absorb the energy while others offer no protection at all. Blast-effect weapons can be even more problematic as their energy spreads over the entire surface of the object instead of a discrete point.

Once you determine if a round penetrates an object or not, you still have to determine what happens to the round. If the round does not penetrate the object, it can either be absorbed or reflected. If it does penetrate the object then its characteristics need to be modified to account for the energy lost during penetration as well as any changes in its flight path. Since objects in the vast majority of simulations are not actually solid, one method that has been employed is to re-spawn the projectile. The original projectile is destroyed at the front surface of the object and a new projectile is created on the far side of the object with appropriately modified properties.

III. RELATED WORK

A. INTRODUCTION

This chapter discusses how military simulations and computer games have commonly addressed the problem of finding cover. We also discuss some of the limitations of their solutions and why we need a better solution.

B. MILITARY SIMULATIONS

1. Terrain Annotation

Many older military simulations calculated cover based on the terrain that the target occupies. For example, a unit in a section of the map with woods would have more cover than a unit that was in the open. There is no actual attempt to draw line of sight, because elevation data is not included in the maps. However, because these simulations aggregate individual combatants together into units this makes sense. It is impossible to do detailed line of sight checks when the simulation does not track the exact location of individual combatants.

As military simulations have developed, they have become more and more detailed. A greater level of detail gives them the potential to be more accurate and provide us with more information that is useful. Many of the current military simulations use terrain elevation data and consider this when determining line of sight. In these systems, the cover that a unit has is determined by a combination of line of sight (as determined by the terrain elevation data) and the type of terrain that the unit currently occupies.

At the far end of this ever-increasing level of detail are the agent-based simulations. In these simulations, combatants are required to think for themselves and react to their current surroundings. Units are modeled all the way down to the individual. In order to accurately model how an individual behaves on the battlefield, they will need to be able to take cover in highly detailed environments.

2. Terrain Cell Line of Sight

An interesting modification of terrain annotation is based on line of sight to the cells instead of just the underlying terrain. Horn and Baxter published a

paper describing their development of a tool that can be used to automatically plan tank squadron assaults that uses this technique [13].

Their program preprocessed the map in order to find the cover locations. The program samples the map data every 100 meters to form depth map of the area of operations. The user, based on the best current intelligence on their positions, assigns the enemy an area on the map. The user places the friendly forces their initial positions and then the program analyzes the situation. Cover is used in both directions during this process: it determines both what the enemy cannot see and which positions provide good visibility of the enemy area.

The program determines the cover that each location provides by calculating how much of the enemy area can draw line of sight to that location. Only the terrain elevation data is taken into account; information on trees, buildings, and other objects are not considered. Line of sight is drawn from each location in the enemy area back to the location we are considering. The fewer places that can draw line of sight to the location, the better the cover it provides.

While this is an effective method for determining cover, it does not suit the purposes of this paper. At one measurement per 100 meters of map area, it is too low detail for our use. Trees, building, and other detailed objects are also not considered. Finally, this is not a “run-time” model. It is designed to preprocess the map, not to calculate cover while the simulation is in progress. Processing line of sight from every position on the map to every other position on the map is very calculation intensive, which may make it undesirable for use in simulations that have to function in real time.

C. COMPUTER GAME AI

1. Scripting and Channeling

Many of the methods used by computer game developers for making their agents use cover could be collectively called “scripting and channeling”. Scripting is when an agent is given a predefined set of actions to follow, like a movie script. Forcing a player take a certain path through the level is called what we call channeling.

Scripting and channeling are a form of preprocessing that is totally dependent on the level designer's ability to find cover. The steps vary, but in general the level designer first plans what type of encounter, they wish to have. Then they pick a location for the encounter in the level. The level designer then places objects in the level that restrict the movement of the player into a channel toward the location of the encounter. This determines the direction from which the agents will need to take cover. The level designer can then find good cover locations and write a script telling the agents how and when to use them.

Used together, scripting and channeling can make computer-controlled agents appear to be very good at finding and using cover. In the computer game Medal of Honor: Allied Assault by EA Games there are examples where enemies fire around the corners of walls and even kick over tables to hide behind. This makes them seem extremely capable of finding cover, but in actuality, the situation is very tightly controlled. If you play the same level several times you will be forced into the same situation and the agents will do the same thing every time. It quickly becomes apparent that they are not really thinking about the terrain.

While scripting and channeling work perfectly well in most games, they completely fall apart in any large, unconstrained environment. In an unconstrained, free-play environment it will be impossible to tell which direction the enemy will be coming from so choosing cover locations becomes impossible. Most computer game levels are also very small compared to any military simulation. At some point it becomes too large for the level designer to consider every locations' cover value.

2. Waypoint Annotation

One of the most popular current approaches to solving almost any problem dealing with interaction between the agent and its environment is to use waypoints. Waypoint graphs were developed as a means of allowing the agents to easily navigate through a level. A series of points are placed throughout the map anywhere the agent may need to go. All the points are connected to the points around them by links. In order to be linked, two points must have direct

line of sight to each other. Once this is done, instead of doing costly navigation processing of the environment, the agents simply move around on the graph like a car following a road.

Game developers use the waypoint graph for more than just navigation. By storing additional information at the nodes on the waypoint graph, they can preprocess the information and store it at the node. For instance, instead of having an agent calculate if it can jump across a small gap during the game, you can use waypoints to store the information. Determine where the agent can make the jump and place a waypoints at the take off and landing points. Store some additional information at the nodes that tells the agent it is supposed to jump when it gets to the nodes. Now when the agent needs to find out if it can make the jump, it queries the waypoint graph and the information is right there for it.

Waypoints can also be used as sensor points to gather information about the environment. The environment around them can be sampled and the information stored with the waypoint. This information is then readily available for the agent to use in deciding where to go. The waypoints can store a wide variety of information including local lighting levels, accessibility, and line of sight.

When we attended the 2003 Game Developers Conference, we talked to many game designers about the problem of finding cover. Every single one of them that we talked to immediately proposed using waypoints as a solution to the problem. In a way, this makes perfect sense. The use of waypoints is well defined and easy to code. It also takes very little processing power during execution, which is very important to maintain high frame rates. In this case, the ability of the agent to find good cover locations becomes almost totally dependent on the level designer's ability to place the waypoints in logical locations. Unfortunately, no matter how carefully the level designer chooses points for cover, all points on the level may provide cover in some circumstance.

One very well developed example of using waypoints to find cover is a paper by Van der Sterren about analyzing terrain to pick sniper locations [24].

Van der Sterren's algorithm functions during execution and processes the waypoint graph against ten separate criteria to determine the best current sniping position. Cover is one of the considerations in choosing a waypoint as a sniping position. Cover in Van der Sterren's algorithm is calculated waypoint to waypoint. A position is considered to provide cover if it has an adjacent waypoint or waypoints that are out of line of sight from the waypoints in the target area.

This algorithm has two main benefits: it is fast and it is adaptable to the current situation in the game. The algorithm is kept fast by minimizing the number of necessary line traces. The set of all waypoints can be easily culled down to a set of interest and line traces are minimized because they are only done from waypoint to waypoint. The algorithm is able to adapt to the current situation by executing during run-time.

Even though Van der Sterrne's algorithm executes during run-time, it still requires significant amounts of pre-computation. In the example that he provides, he uses around 480 waypoints on a small level that appears to be less than 500 meters square. A level designer must place each of these waypoints. Because of this, the performance of the algorithm will be highly dependent on the density of the waypoints and the quality of their placement.

One problem that waypoint-based cover algorithms have in dynamic environments is that they are static. Checking the same location every time means that they will not always offer the best solution. Consider Figure 1 below. The circle represent waypoints and the dashed lines the graph between them. A large object in the center of the room provides cover to everything on the other side of it. This area of cover is represented by the cross-hatched area. If the agent searches the waypoint graph it will find that there are two waypoints that are in cover. I could choose either of these as places to move to and be in cover from the threat. However, neither of them offers an optimal solution. The agent can get into cover faster by moving directly to a point inside the cover that is not part of the waypoint graph.

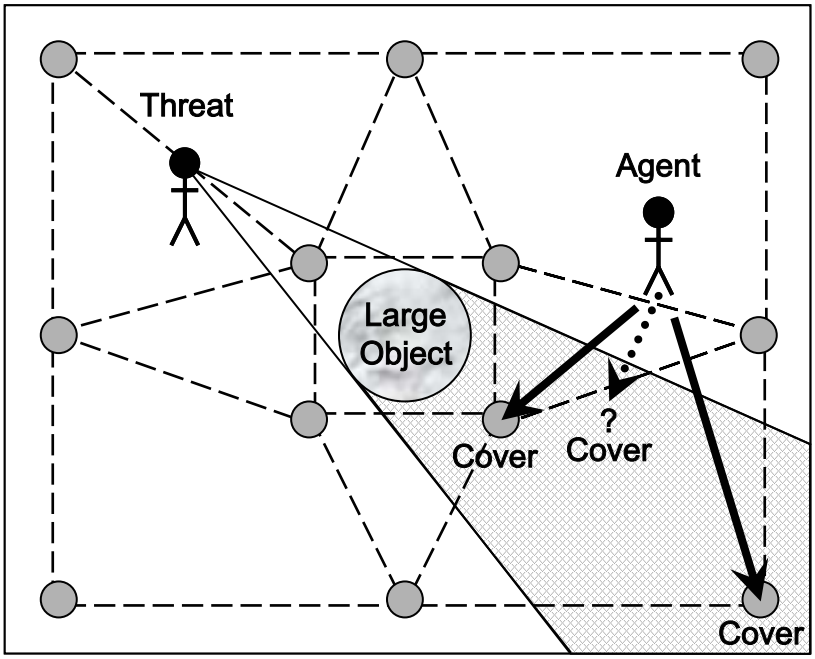


Figure 1. Using Waypoints Can Miss Cover Locations

Another problem with using waypoint graphs is that they cannot handle dynamic terrain. The level designer positions the waypoints long before the game begins. The placement of each waypoint is carefully considered, but there is no way to anticipate changes that may occur in a dynamic environment. As the terrain changes, cover opportunities may be missed.

Consider Figure 2 below. The left hand side shows the initial situation. The agent has taken cover at the center waypoint based on line of sight checks from the threat to the waypoints. The right had side shows the situation after an explosion that has moved the large object. None of the waypoints are now in cover and the object even blocks part of the waypoint graph. Even though cover is still available, the agent is unable to find it by using the waypoint graph.

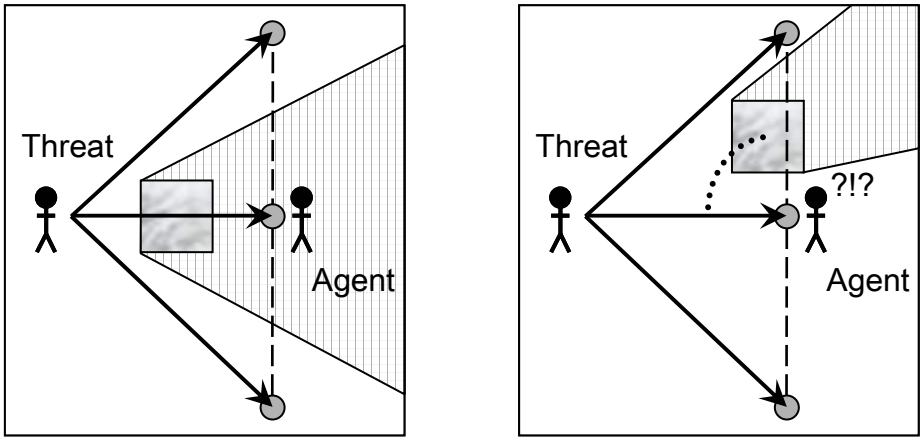


Figure 2. Waypoint Methods May Not Handle Dynamic Environments

THIS PAGE INTENTIONALLY LEFT BLANK

IV. CONCEPTUAL MODELS

A. INTRODUCTION

This chapter discusses the three conceptual models that we developed for finding cover in dynamic, three-dimensional environments. The Binary Space Partition Shadow Volume Tree method uses light-source shading methods to find areas of cover. The Depth Mapping method uses a rough approximation of computer vision techniques to find places where the agent can potentially take cover. The Sensor Grid method places a series of sensors around the agent and uses them to detect potential cover locations.

B. ASSUMPTIONS

We made the following assumptions in order to simplify the algorithms in development. If needed it should be possible to modify any of the algorithms so that any of these assumptions are no longer necessary.

1. We are Only Concerned About Taking Cover from Direct Fire

This assumption allows us to focus on one type of cover. Cover from direct-fire weapons is the simplest case to analyze because we can assume that the projectile travels along a straight line from the threat to the target.

We can add cover from indirect-fire weapons later by determining likely impact points. Adding cover for area-of-effect type weapons will depend on the behavior of the particular threat. Changes for cover from area-of-effect type weapons could involve signification changes in the algorithms if the effects of the weapon behave like a fluid. To accurately calculate the area of effect could require complex calculations and could also depend on the geometry of the area of detonation.

2. Any Object that Blocks Line of Sight Provides Cover

This assumption allows us to avoid analyzing material properties while calculating cover locations. Material properties are often missing in many simulations so this information would not be available. In some simulations, this information may be stored at the object level. This would require a capability to determine which object individual polygons belong to.

Even when an object does not provide cover, it may still provide concealment. When cover is not available, concealment is better than nothing at all. Therefore, while we are focusing on finding cover, this assumption still provides reasonable behavior for our agent.

To design a system where we do not make this assumption requires that we consider the penetration of objects and their value as concealment. Such a system would need a line-of-sight algorithm that was able to return every intersection along the weapon's potential flight path. The line-of-sight algorithm would also need to provide access to the material properties of the objects intersected.

The decision logic for an agent in such a system would need to be much more complicated also. In determining where to move, the agent would have to weigh the value of cover verses concealment. It would need to be able to decide between a close position that provides concealment verses a far position that provides cover. It would also need to be able to determine if an object provides enough cover to stop the threat the agent is facing.

A full system that can find both cover and concealment will most likely require separate algorithms for cover and concealment. Cover and concealment are fundamentally different in that concealment does not require an intervening object. Since concealment involves anything that hinders the enemy's ability to see the agent, it involves as much signal detection theory as it does line of sight algorithms. Most cover algorithms will not be able to handle cases where you can take concealment by "hiding in plain sight" (e.g. in shadows or through camouflage where there are no intervening objects between the threat and the target).

3. There is Only One Enemy

This assumption allows us to explore the simplest case where there is only one threat. We need to be able to handle one enemy before being able to handle multiple threats.

There are several ways that an algorithm could be implemented to consider multiple threats. The most straightforward option is to calculate cover for every threat individually and then combine the results. However, this option can be very computationally expensive. Another option is take some subset of the enemy and calculate cover based on their locations. This option is less expensive computationally, but its performance is highly dependent on the actual arrangement of the enemy forces. A good, general-purpose method of handling an arbitrary number of threats from several different directions will require significant effort to develop properly.

4. The Agent Taking Cover Knows the Location of the Enemy

This assumption allows us to focus on finding cover instead of determining the agent's perception of threats.

To develop an algorithm where the location of the enemy may not be known requires modeling of the agent's perception of the enemy's location. If there is any information at all the agent would have to be able to hypothesize the location of the threat in order to decide where to take cover. If there is no information at all, then the situation is almost reversed. In this case, the agent will want to find the cover location that provides the best protection from the widest area.

5. The Agent Taking Cover Knows What the Enemy Can See

This assumption reduces our computational complexity by avoiding calculation of natural biases in human exocentric perception. Doing one calculation with perfect knowledge at the beginning helps us avoid additional processing requirements.

In order to implement a system with imperfect knowledge of what the enemy can see we would need to model the natural biases in human exocentric perception. In real life, we compensate for these biases by constant reevaluation of the cover location as we move toward and into the position. Accurately replicating this in an agent would greatly increase the computational cost of running any cover algorithms.

6. The Agent Taking Cover has Perfect Knowledge of the Area of Interest Around It

This assumption allows us to find cover without maintaining an internal representation inside the agent of the surrounding area. This saves memory space, the processing power that would be required to build the internal representation, and the processing power for hypothesizing about areas where the agent only has partial information.

To build an algorithm where the agent had imperfect knowledge of the area of interest around it would require extensive sensor modeling, a detailed internal representation of the surrounding area, an ability for the agent to hypothesize about unseen areas, and a conflict-resolution capability.

The agent would need to be able to process detailed information on the portions of the area that it had observed and build it into a detailed internal representation. The cover calculations would then be run on the internal model of the world instead of the actual environment. All of the algorithms presented in this paper require detailed line of sight information. The only way to accurately process the same information inside of the agent's perception would require an almost perfect copy of the environment down to the polygon level. The memory requirements for this would be enormous.

The agent would also need to be able to hypothesize about areas that it has not directly observed. Given that the agent sees a box on the floor from one angle, it would need to be able to determine whether it thinks that the unobserved side of the box would be a suitable cover location or not.

The agent would also need to have decision logic to deal with conflicts in its internal representation and the actual environment. Continuing the example in the paragraph above, suppose that the agent has decided to take cover behind the box. It moves around the box to find that it cannot get behind the box for some reason. The information that it based its decision on has now changed. The agent must be able to deal with these situations.

C. THEORY

This section deals with the theory we developed for guiding the development of our algorithms. When searching for cover in virtual environments, we must ask three basic questions:

- Where is there cover?
- Where can I reach?
- Where can I stand?

Covered areas offer protection from direct fire. As stated in the assumptions section we will consider any area that is out of direct line of sight from the threat to provide cover.

Reachable areas are places that I can move to. Some path must exist that the agent can use to reach the location. A cover location that we cannot reach does us no good.

Standable areas are places where the slope of the terrain and the composition of the surface material allow us to stand, kneel, or lay prone. This is important because not all reachable areas are suitable as places where we can stop and take cover. You might be able to get into a place behind cover by jumping, but there may not be suitable surface there that allows you to remain in that position.

What we are looking for is the intersection of these three areas: Covered Reachable, and Standable. Since there are many methods currently available for finding the Reachable and Standable areas, we have focused our algorithms on finding Covered areas. However, each algorithm presented in the following sections will detail any advantages or disadvantages that it offers in finding the Reachable and Standable areas.

D. BINARY SPACE PARTITION SHADOW VOLUME TREES

1. Definition

a. BSP Trees

A Binary Space Partition tree (or BSP tree) is a hierarchical subdivision of an n dimensional space into homogeneous regions [3]. Fuchs,

Kedem, and Naylor developed BSP trees in 1979-1980 as a way of determining visibility priority in three-dimensional scenes [10][11]. Since then they have been used for a variety of applications including collision detection, ray tracing acceleration, partitioning of polygons into convex sub-polygons, and shadow generation [3].

BSP Trees use n-1 dimensional shapes to partition an n dimensional space into two convex subspaces. Each subspace can be partitioned into two more subspaces until the scene is completely partitioned into elemental, convex subspaces. All of the information about the partitioning is stored in the structure of the binary tree. Each node contains information on the n-1 dimensional shape used for the partitioning. Each node also has a front and a back leaf. The binary structure of the tree allows for efficient traversal and quick information retrieval.

The BSP tree is built by adding elements from the scene one at a time and placing them into the tree appropriately. There are four cases that each element can fall in: in front, behind, coincident, and spanning. Consider the case of a two-dimensional area with objects A, B, C, D, and E as shown in Figure 3. The two-dimensional space is partitioned into subspaces by one-dimensional lines. The tree is started by picking a root node which in this case is A. A line is created through A partitioning the space into two sub-spaces. One side of the partition is chosen to be the in-front area (marked by a + sign) and the other is the behind area (marked by a – sign). The original information on A is stored inside the A node. We add B next and find that it B falls in the positive subspace of A. B gets added as a positive leaf node to A. Object C is in the negative region of A so it is added as a negative leaf node of A. When Object D is added, we find that it is coincident with object B, so its information is stored in the same node where object B's information is stored. When object E is added we find that it spans the partition created by C. E is split along the partition into E1 and E2. E1 is added as C's positive child and E2 is added as C's negative child. We have now partitioned the entire space into convex subspaces.

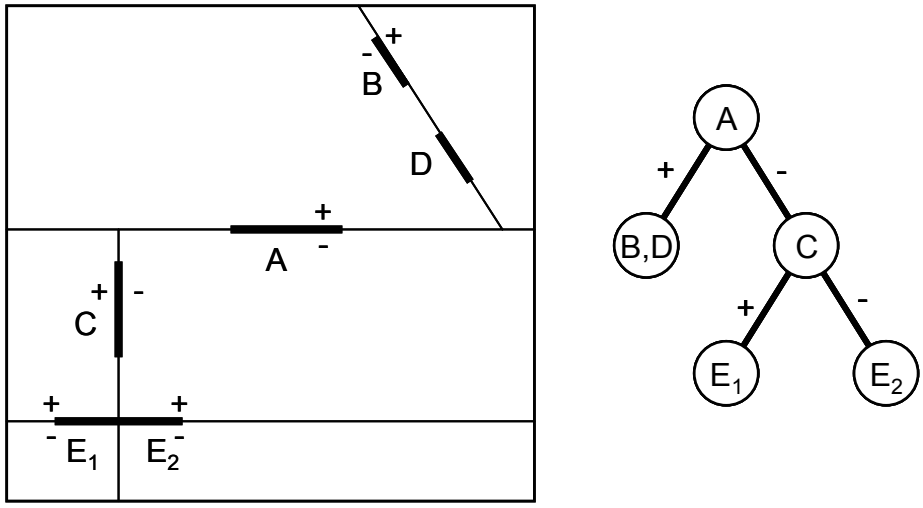


Figure 3. BSP tree for a two-dimensional space.

When used in computer graphics BSP trees are very useful for displaying static images. We can use a back-to-front, painters algorithm or a front-to-back, scanline algorithm to render the view. All of the visibility information can be quickly extracted by traversing the BSP tree. Once we build the tree, the camera can be moved freely without rebuilding the tree; it only requires that the tree be traversed in a different order.

BSP trees have also been developed for use in dynamic scenes with moving objects. As objects move, their nodes are removed from the tree and reinserted. Since the tree does not have to be completely rebuilt for each render, performance is greatly improved. Normally all of the objects in the scene are classified as static or dynamic. The tree is built with the static objects first and then the dynamic objects are add last. Adding the dynamic objects last puts them in the leaf nodes of the BSP tree, which minimizes costly internal changes to the tree structure.

b. Shadow Volumes

A shadow volume is a semi-infinite volume that denotes an area that is blocked from a light source [16]. Objects that are inside this area are in shadow. Objects that are outside this area are not. A shadow volume is enclosed by shadow planes, which are formed by using the edge vertices of a

polygon and a point light source. The direction of the normal of the shadow plane determines which side of the plane is in shadow or out of shadow. To create the shadow volume for a polygon, we create a shadow plane for each edge of a polygon and then top it off with a plane through the polygon itself. Figure 4 shows an example of a shadow volume.

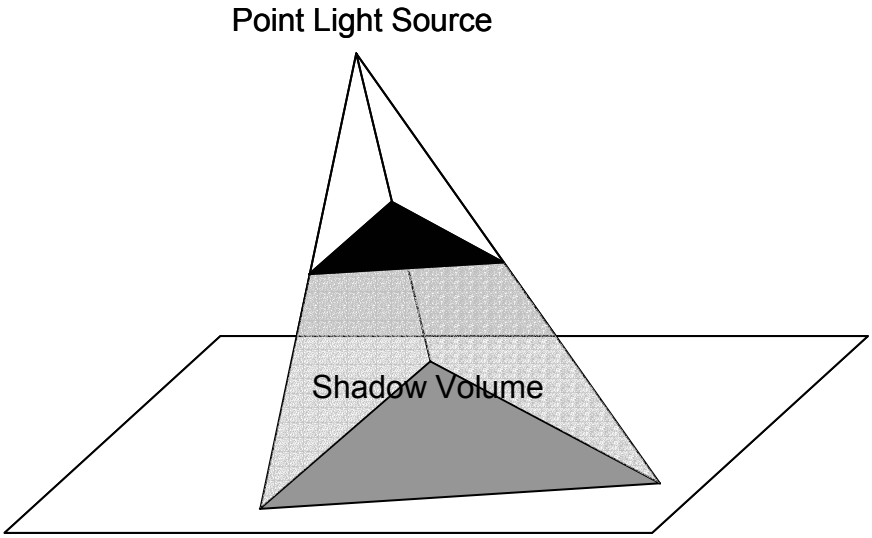


Figure 4. A Shadow Volume

c. BSP Shadow Volume Trees

Chin and Feiner introduced Shadow Volume Binary Space Partition Trees (or SVBSP Trees) as a means of computing shadows from point light sources in static scenes [6]. SVBSP trees use the structure of the BSP tree to create a merged shadow volume for the scene. When each polygon is added to the SVBSP tree the subspace is partitioned along the edges and in the plane of the polygon to create the shadow volume for that polygon. In their algorithm, Chin and Feiner imposed a strict, front to back insertion of the polygons so that the faces of all polygons in the SVBSP tree were guaranteed to be lit. As more polygons are added, the merged shadow volume evolves. Due to the special nature of BSP trees, the camera can be moved around the scene without having to recomputed the scene.

Since their introduction, there have been many improvements to the basic BSP tree and SVBSP algorithms. Chin and Feiner have developed

methods to deal with multiple light sources as well as area light sources that generate realistic umbra and penumbra regions [5]. Chrysanthou and Slater developed a method for building unordered SVBSP trees that can handle dynamic environments [7]. Their method allows for the transformation of objects in the scene without entirely rebuilding the SVBSP tree.

2. Concept Applied to Finding Cover

If we treat the threat as a point light source, then areas that are "in shadow" can be considered to be in cover. Create a SVBSP tree of the scene using the threat as a point or area light source. The merged shadow volume of the SVBSP tree shows all of the areas where cover may be found.

One important modification of the standard SVBSP tree algorithm is to build it only using rear-facing polygons. Normally the SVBSP tree algorithm only considers polygons that are facing the light source. Rear facing polygons are not visible and cannot be lit so they are culled early in the process. If we use the forward facing polygons in building our SVBSP trees, many of the shadow areas will be inside of objects (see Figure 5 below). Since we cannot move there, it does us no good to consider these areas as cover.

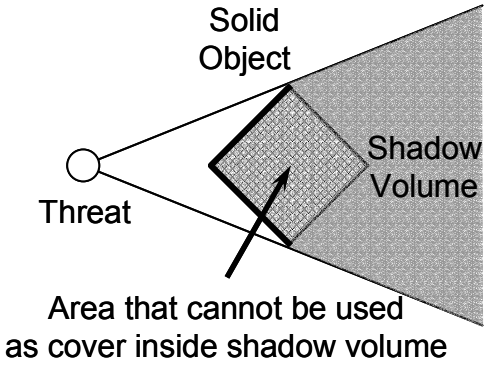


Figure 5. Shadow Volume Created Using Forward-Facing Polygons

By only using the rear-facing polygons, we eliminate the inside of solid objects from our shadow volume (see Figure 6 below). This has the potential to increase our greatly efficiency when attempting to find a point in cover.

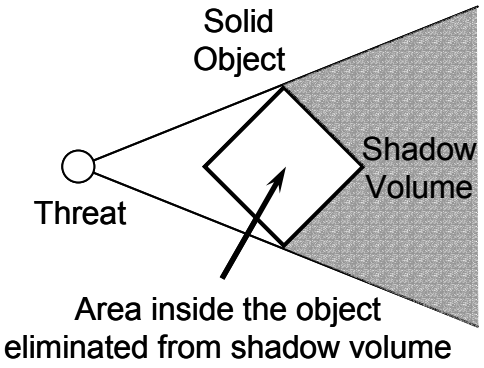


Figure 6. Shadow Volume Created Using Rear-Facing Polygons

3. Steps in the Algorithm

Our algorithm for finding cover using SVBSP trees has the following steps:

- Determine the location of the threat.
- Create a view frustrum from the threat, centered on your location, which is large enough to encompass the area where you want to find cover.
- Build the SVBSP tree using all of the polygons in the scene graph culling those that are outside of the view frustrum and those that are forward facing.
- Find the intersection of the shadow volume and the standable surfaces.
- Determine which surfaces in this area are reachable.
- Choose a reachable point and move there.

4. Benefits

Using SVBSP trees to find cover has two main benefits: it finds areas of cover and it uses no approximations in finding cover.

By finding areas of cover instead of cover points, it gives the agent more complete information about its surroundings. The agent can plan movement routes that stay inside of the covered area. The agent can also identify broken areas of cover and large uniform areas of cover, which could be part of its decision criteria in choosing an exact destination for its move.

SVBSP tree use no approximations when determining cover. Since it uses the exact polygon model for the scene and mathematical equations for the

shadow planes there are no approximations that need to be made when determining the areas that are out of line of sight from the threat.

5. Problems

BSP trees require a large amount of computational power and may be too slow for some applications. According to the BSP Tree Frequently Asked Questions section of the OpenGL website, complexity (time and space) for BSP trees is $O(n^2)$ upper bound and O(n) expected for n polygons [3]. With the ever-increasing polygon counts in three-dimensional environments, the processing power may not be available to make the SVBSP tree algorithm feasible for use in real-time applications.

Another problem with the SVBSP tree method for finding cover is that it can be very difficult to determine if your agent will fit behind a piece of cover and be totally hidden. If a model uses lots of large polygons the shadow volumes will also be large and your agent can fit entirely inside of one. However, say you have a rock which has 100 rearward facing polygons from the threats location. Each polygon will create its own shadow volume. None of them alone may be large enough for your agent to fit in, but collectively they provide plenty of room. There must be some method for handling this situation.

While we have listed finding areas of cover as a benefit for this method, it can also be a problem when trying to find a movement destination for the agent. An area defined by several planes essentially has an infinite number of locations inside. Choose any two points inside the area and you can produce a point in between them. So you must develop a method for choosing a single point as a destination inside of the cover area once it is computed.

E. DEPTH MAPPING

1. Definition

Depth Mapping is a means of representing n dimensional data in an n-1 dimensional form. Each element in the n-1 dimensional format holds the data on the n^{th} dimension. As the name implies, this is normally depth or distance information. For example a depth map of a three dimensional scene from a specific viewpoint would be a two dimensional grid with numbers in each square

representing the distance from the viewpoint to the first object struck in that square.

2. Concept Applied to Finding Cover

Depth mapping is similar to computer vision techniques in that it transforms the scene into a numerical representation of what the threat can see and processes the information. It produces a reduced-detail resolution approximation of the depth of objects in the scene from the threat's point of view. We must then process the information stored in the depth buffer to find cover in the scene.

Depth mapping creates a layer of information on the scene. This is very similar to shadow mapping in that only the information on the closest object to the threat is stored in each cell of the grid. There is no information on the objects that lay behind this point. In fact, depth maps have recently been proposed as a method of creating shadow volumes.

The depth map is built by constructing a two-dimensional grid at some virtual location between the threat and the target. A line is traced from the threat's eye position through the center of each cell on the grid. The distance to the first object encountered is stored as the value for the cell. If the trace does not encounter any objects, then some null value is stored.

Building the depth map with rear-facing polygons provides the same benefits as it did with SVBSP trees. We want to find things that we can get behind, not inside. Only considering rear-facing polygons in the area of interest can eliminate some points inside solid objects from consideration.

Once we build the depth map, we can use templates of our agent to determine where there is cover. The templates tell us how many cells our agent covers on the depth map for a certain posture at a certain distance. So for instance, we may have one template that tells us that our agent appears to be one cell wide and two cells tall at 120 meters while standing. Another template would tell us that our agent appears to be three cells wide and five cells tall at a distance of 30 meters while standing. We must build one set of templates for

each posture that our agent is capable of assuming (i.e. standing, kneeling, and prone).

In order to find all cover locations we must check every cell in our depth map against the templates for each posture. Starting with the furthest template, check its distance against the distance of the cell. If the distance stored in the cell is less than the distance for the template, then we check all of the other cells covered by the template. If all of these cells are also closer than the distance to the template, then we have cover at that location for that posture and distance. The next step is to check each of the closer templates until we find the closest template of that stance that allows us to be in cover. The initial locations of our cover positions can be computed using a vector to the bottom center of each element and the distance stored there. (See Figure 7)

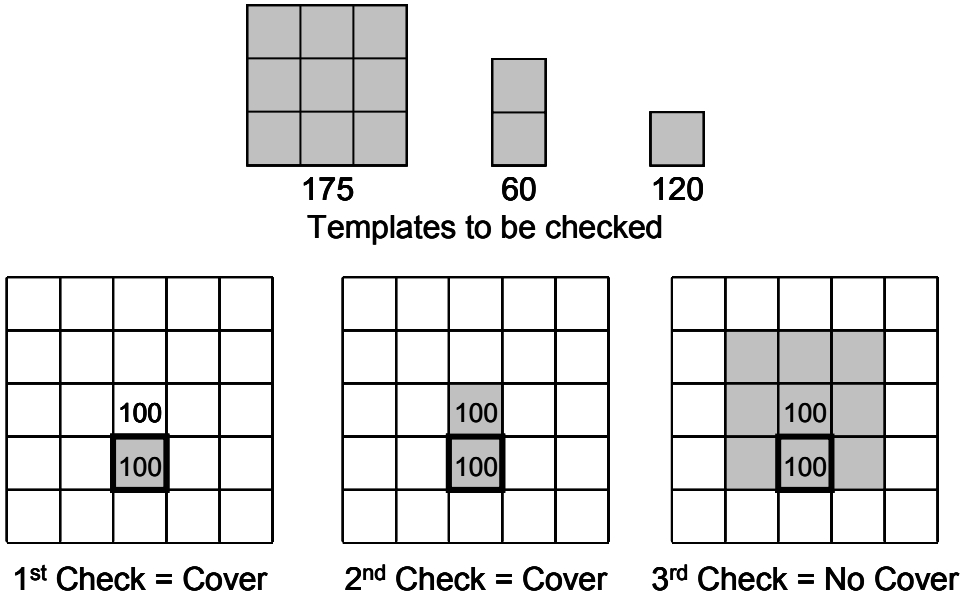


Figure 7. Checking a Sensor Grid Element for Cover

After we have determined the cover positions for the sensor grid, we must determine if there is a corresponding covered place we can stand. Due to the way that we have constructed the sensor grid so far, it is entirely possible that the locations we have stored are actually floating in the air or under the surface of

the terrain. Since we have stored the closest possible cover position, we can trace a line from the threat's eye position, through the bottom of the sensor grid element until it hits the terrain. If a standable position exists along this line, we move our initial cover locations back to this point. (See Figure 8)

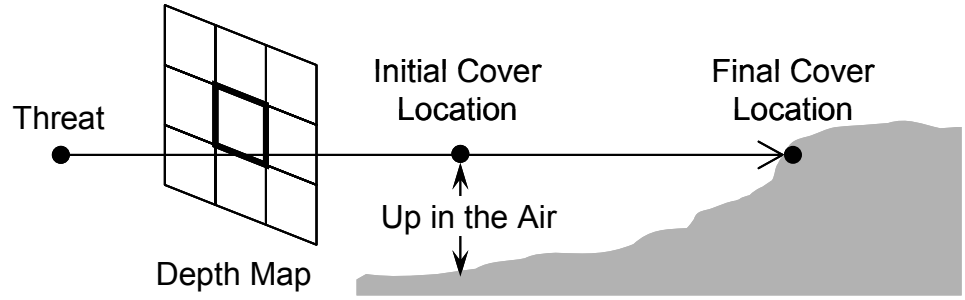


Figure 8. Moving Cover Locations to Standable Positions

After iterating over all squares in the depth map with all postures, we will have a list of points that are in cover, and on a standable surface. Each position in the list will also have a posture associated with it. From this point, we can determine which points are reachable and which point we want to move to.

3. Steps in the Algorithm

Our algorithm for finding cover using depth mapping has the following steps:

- Determine the location of the threat.
- Build the depth map for the scene from the threat's point of view.
- For each element in the depth map find the closest prone, kneeling, and standing template that is completely covered.
- For each prone, kneeling and standing position identified so far, move the point away from the threat till it rests on standable terrain. Discard any points where standable terrain does not exist.
- Test the remaining points and discard any that are not reachable.
- Choose one of the remaining points and move there.

4. Benefits

One benefit of the Depth Mapping method is that you do not have to perform any separate checks to determine if you will fit inside the area of cover.

Because of the way we processed the depth map information we are guaranteed to fit in the cover at that point or some other point further away.

Another benefit is that Depth Mapping also gives you a definite location to move to. Even though this point may not be navigable, we know that we can search for navigable points along the line away from the threat and still remain in cover.

Depth mapping has no problems with objects made of large numbers of polygons. Since only one check is made in each area, the number of polygons in the scene have do not affect the functioning of the algorithm.

5. Problems

Depth maps are approximations of the surface area of the cover. As such they give you very little information about anything behind the surface. There may be many objects that are stacked close enough together that you cannot move there, but the Depth Map does not provide you with any additional information about it. The hardest part of this algorithm will be determining if the area behind the front of edge of the depth map provides enough room for you to navigate.

Depth maps are less computationally expensive than the shadow volume method, but still very expensive. The actual expense for performing the algorithm will depend on the size of the grid and the efficiency of the line of sight algorithm. As the number of points in the horizontal and vertical planes increases linearly, the number of points that must be tested and calculated over the whole grid increase exponentially. Increasing the number of points to be tested also increases the number of templates that must be stored for testing.

F. SENSOR GRID

1. Definition

A sensor grid is a group of objects that are used to gather information about the surrounding area. The types of information that the sensors gather depends on the type of sensor and the information available from the environment. The sensors are arranged in some pattern, or grid, that gives them

systematic coverage of the area of interest. Often this pattern is some sort of even spacing, but it can also be modified so that there is a higher density of coverage in areas where more detailed information is necessary.

2. Concept Applied to Finding Cover

A sensor grid can be used to sample the area around an agent to find cover in its environment. When the agent needs to find cover, it deploys the grid and uses the sensors to gather information on cover locations. It can then use this information to choose a cover location and then move to that sensor's location.

The sensors used in this method can be very simple. At the most basic level all that the sensor needs to be is a location in space. We can determine if a sensor is in cover if we do a line of sight check between the threat and the location of the sensor. If the line of sight check is blocked, then we can assume that the location offers cover. Additional information that might be useful to store in the sensor would be if the location is over standable terrain and the posture that the agent will need to take to benefit from the cover.

The deployment and arrangement of the sensor grid can have a large effect on the number of sensors required to cover an area and the quality of the results returned by the algorithm. While it is perfectly possible to arrange the sensors in a grid that uniformly covers an area around the agent in the x, y, and z directions, it is not necessary. Under most conditions, the vast majority of the movement of the agent will be in horizontal directions. Ground is generally flat and we tend to think about ground movement in horizontal planes. Our elevation is a matter of gravity holding us to the current ground level. We can take advantage of this in the development of our sensor grid pattern.

One horizontal layer of sensors, clamped to ground level, can reduce the number of required sensors while still finding cover around the agent. If the sensor grid extends both in the horizontal and vertical directions with no regard to ground level, a lot of them will provide meaningless information. Sensors that are too far above ground level will be testing points up in the air that we cannot

reach. Sensors that are below ground level will also be in unreachable positions. (See Figure 9 below) Eliminating these sensors greatly reduces the number of checks that the algorithm must perform when they are not likely to produce useful results anyway.

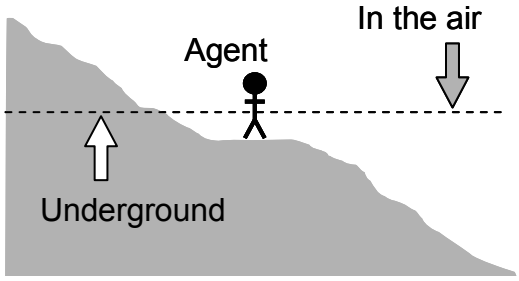


Figure 9. Problems with a Flat Horizontal Layer of Sensors

We must carefully consider the arrangement of the sensors in the horizontal plane of the sensor grid to ensure that it produces good results. The first thing that we must determine is how large of an area we want the sensor grid to cover. It should be large enough that all cover locations within a reasonable distance are considered. The second item we must consider is the density of the sensor locations; that is how close together we want them. If the sensors are too far apart, they will not detect cover opportunities. If the sensors are too close together, the number of sensors can grow so large that the algorithm is too slow for real-time operation. There must be a balance between resolution and speed. The last item that we must consider is whether the sensors have an even spread. Spreading the sensors in an uneven fashion will leave gaps in coverage and some valid cover locations will not be considered.

When testing a sensor location for cover there are four cases that we must consider: no cover, cover while prone, cover while kneeling, cover while standing. This assumes that all three of these postures are available to the agent. The easiest way to determine which case applies to the sensor's location is to start from the ground up. Since almost all objects that provide cover are on the ground (or are the ground itself) it makes sense to start there. There are also some situations where a person's leg may be exposed, but their upper bodies

are in cover. Checking from the ground up allows us to quickly label places with no cover and stop checking. If that location has some cover, then we can do a more detailed examination of the location to determine which posture is the highest we can maintain and still be in cover.

3. Steps in the Algorithm

Our algorithm for finding cover with a sensor grid has the following steps:

- Determine the location of the threat.
- Create the sensor grid in the predetermined pattern.
- Clamp the sensors to ground level.
- Determine which sensors are on standable terrain.
- Test line of sight to the sensors and classify them.
- Of the sensors that are in cover and on standable terrain, determine which ones are reachable.
- Choose a sensor that is in cover, on standable terrain, and reachable as your destination.
- Move to that sensor's location.

4. Benefits

Of all the algorithms presented so far, the sensor grid is the most computationally efficient. As the polygon count of the scene increases, the algorithm the only thing that affects the speed of the algorithm is the efficiency of the line of sight algorithms that it depends on. It handles objects made of single polygons or millions of polygons just as efficiently. The biggest impact on the speed of the algorithm is the number of sensors used. As the number of sensors increases, the number of line of sight checks for finding cover only increases linearly.

The Sensor Grid algorithm gives you half of a navigation solution before you even begin to calculate which points are reachable. Since algorithm already requires us to clamp the sensors ground level in a regular pattern, we can use them as temporary waypoints. This allows us to efficiently navigate around objects into cover locations.

The Sensor Grid considers all of the area around the agent equally. The other two algorithms we have considered create a "cover surface" where

anything that is behind it is in cover. This algorithm can deal with objects placed one behind the other in arbitrarily complicated scenes.

5. Problems

The Sensor Grid method conducts a low detail sampling of the area. As such it may miss some cover opportunities. Points that are in between two of the sensor may provide good cover, but will never be considered.

The Sensor Grid method is prone to false reporting in situations with large numbers of small objects. This is because the line of sight checks are conducted from point to point, not point to area. It may be the case that an object is large enough and in the right position to block the line of sight check while other parts of the target are clearly visible.

The Sensor Grid method reports the insides of objects as valid cover locations. Since the cover determination is based purely on line of sight, the insides of solid objects will be reported as providing cover. It is not until we attempt to navigate to these locations that we find that they cannot be reached.

THIS PAGE INTENTIONALLY LEFT BLANK

V. IMPLEMENTATION OF THE SENSOR GRID MODEL

A. INTRODUCTION

This chapter describes our implementation of the sensor grid model inside of the computer game America's Army: Operations (AAO). AAO is based on the Unreal Engine so the first section gives some background on it and what it has to offer as a three-dimensional virtual environment. The second section describes the Army Game Project and some of the modifications that they have made to the Unreal Engine. The last section details how we built an agent in AAO that uses the sensor grid method to find cover.

B. THE UNREAL ENGINE

Tim Sweeney, the founder of Epic Games, developed the Unreal Engine. He began research in 1994 and published the engine in 1998 [21]. His design goals were to allow developers to create true three-dimensional environments and to enable programmers to control and customize all aspects of the environment and characters. The game engine has been constantly improved over the last five years and is one of the best of its kind. It is not only featured in the Unreal series of games, but has also been licensed by many other companies to use in their games.

Programming in the Unreal Engine is written in accomplished through the use of UnrealScript and C++. It supports full four-way function calling between C++ and UnrealScript, so developers can freely mix code between the two. While Unrealscript runs many times slower than C++ code in the game, it is more simple to use in some cases. UnrealScript looks like a cross between Java and C++ with some additional added features. Two important capabilities of UnrealScript for agent programming are state-based execution and time-based execution. Agents behaviors are controlled by their current state and changes in state. Time-based execution allows the modeling of actions that take a certain amount of real-life time to complete, without constantly checking to see if it is complete.

The Unreal Engine is an idea environment for exploring agent-based research. It provides a highly detailed three-dimensional environment with realistic physics. It has built-in support for artificial intelligence programming.

C. THE ARMY GAME PROJECT

The Army Game Project is an attempt by the U.S. Army to allow civilians to learn about the Army through a computer game called “America’s Army: Operations” [1]. The Assistant Secretary of the Army for Manpower and Reserve Affairs selected the Modeling Virtual Environments and Simulation (MOVES) Institute at the Naval Post Graduate School to develop the game. Development began in January 2000 and it was first published in July 2002. Since then over 1.67 million people have downloaded the software for free. Users have played over 130 million missions logging a staggering 13 million hours of game play.

America’s Army: Operations (AAO) has many features that made it our first choice for this research. AAO is based on the Unreal Engine, which is easy to learn and provides a rich, detailed three-dimensional environment. Since the MOVES program developed the game, we had easy access to the code and many people with significant experience with Unreal programming experience.

One of the most significant reasons for using AA:O is that the computer-controlled agents can go prone. Many of the games published today do not have this feature. In fact, the base Unreal Engine itself does not support the ability for people to go prone. We feel that the ability to go prone is an essential part of making realistic use of cover, so it was essential that our development environment supported this ability.

We used version 1.6 of America’s Army: Operations for all of our research.

D. COVERBOT

1. General

CoverBot is our implementation of the sensor grid method of finding cover inside of America’s Army: Operations. There are two versions of CoverBot: a step-by-step version called CoverBot and a full-speed version called

CoverBotTwo. CoverBot stops after each major step of the algorithm to allow verification of its results. CoverBotTwo executes without any stops to allow us to see it run at full speed.

2. Flow of Execution

Both of the CoverBots have essentially the same flow of execution. The only real difference is that the sensor grid for the step-by-step version is deployed immediately while the agent is in the waiting state. This makes it easier to test the Bot because we can see where the sensors are that it will be using.

Both of the CoverBots start in a standing position and go into a waiting state. We use the “takeDamage” message from the game engine to trigger the agents to advance through each phase of the algorithm. The game engine sends this message to the agent each time we shoot it so we can use our rifle to control the flow of the algorithm inside the simulation.

The next step is for the agent to create the sensor grid. The agent determines the placement of each sensor based on a pattern centered on the its current location. When the initial position of each sensor is determined, the agent attempts to clamp the sensor to the terrain at that position. If it is successful, it also checks if the terrain at that location is standable. If there is a standable surface at that location, the agent creates a sensor there.

After the agent places all of the sensors in the pattern, it determines which of them are in cover. For each sensor, the agent checks line of sight to three different positions. The agent moves the sensor from ground level to the prone height of the agent, to the kneeling height, to the standing height. At each height it checks the line of sight between the enemy and the sensor’s location. The agent then picks one of four cover categories for the sensor’s location: no cover, cover while prone, cover while kneeling, or cover while standing.

After the agent has categorized the cover at all of the sensor locations, it tries to determine the shortest path to any of the points that have cover. For each sensor in cover, the agent checks to see if there is a straight-line path to the sensor’s location. If no direct path exists, it checks to see if it can reach the

cover location by traveling through any of the other sensor locations first. Essentially, it is using the other sensors as waypoints for a two-part path to the cover location.

The agent selects a movement destination, by finding the shortest path into a sensor position that provides cover. If a valid path is found, then agent moves to that position and then changes its posture based on how the cover was classified for that location. The agent always remains at the tallest posture that it can assume in that location and still be in cover.

Once the agent reaches cover, we can reset it by shooting it again. The agent will stand up and prepare itself to deploy the sensor grid again.

3. Solutions to Problems

The following paragraphs provide additional details on how we implemented various parts of the algorithm.

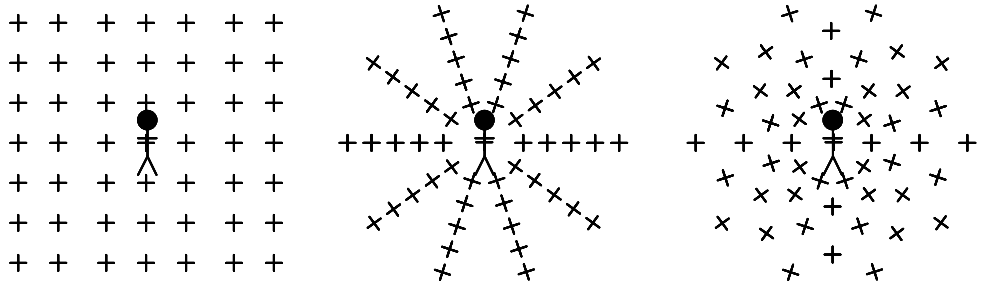
a. Determining the Location of the Enemy

We give CoverBot the exact location of the enemy. The "takeDamage" message that is used to control the flow of the algorithm also passes the agent a reference to us (as its enemy). From this it can determine where our eye's are and uses this location for determining line of sight to the sensors. The agent assumes that we stay in the position where we initially shot it from while it was in the waiting state. This allows us to move around to better viewpoints during execution.

b. Building the Sensor Grid

We found that offset rings of sensors provide excellent coverage of the area searched for cover. A traditional grid formation would place the sensors equidistant from each other over a square area. (See Figure 10, left side) However if we place the sensors in equidistant rings around the agent with each ring containing the same number of sensors we get a variable density coverage of the area. The density of sensor points is highest close to the agent and lowers further away from it. (See Figure 10, center) However, this variable density is actually productive to the algorithm. It allows the agent to make a more detailed check nearby for cover while still considering points that are further away. This

seems to parallel parts of the human cover-finding process. We give more consideration to nearby locations than those that are further away because we can reach the nearby locations faster. Finally looking at the rings of sensor locations, we see that it does not give us very even coverage of the area. Large gaps in coverage have developed between rows of sensors as you travel outward from the center of the rings. By simply offsetting the rotation of every other ring by half the angle between adjacent sensors of the same ring, we can even out our coverage and still maintain a pattern that is denser on the inside than the outside.



Standard Grid
Uniform Spacing
(49 Points)

Circular Grid
Uniform Rings
(51 Points)

Circular Grid
Uniform, Offset Rings
(51 Points)

Figure 10. Sensor Grid Patterns

c. Clamping the Sensor Grid to Ground Level

We developed the following method for clamping our sensors to ground level. First, the sensors were deployed in a horizontal plane from the foot level of the agent. A trace was done directly down from each sensor to determine if the surface of the terrain was below the horizontal plane. If this check failed to produce a surface, a second check was performed from above ground level down to the horizontal plane to find a surface. If no surface was detected by either check then a sensor was not created. (See Figure 11 below) The reason that we check was split into two parts is that if you only do one check from top to bottom, there is a tendency for the points to be clamped to the tops of buildings and other objects. Checking from ground level down first also gives the agent a tendency to go downhill which is common when taking cover because it is easier to move downhill.

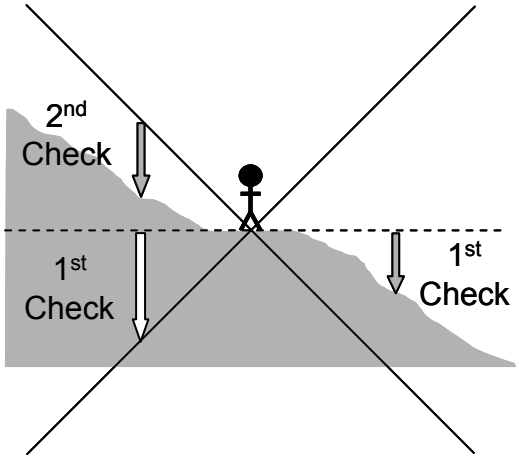


Figure 11. Clamping Sensors to Ground Level

Lines that extend upwards and downwards from the agent at 45-degree angles limit the length of the two traces. This keeps the sensors from being placed at a distance that was too far up or down for the agent to be able to reasonably navigate to. We do not want our agent jumping off cliffs to take cover or trying to climb up areas that are too steep to climb. However, the agent should be able to make reasonable small jumps down, so we modified the lower boundary line to allow it to jump down up to its own height. (See Figure 12 below)

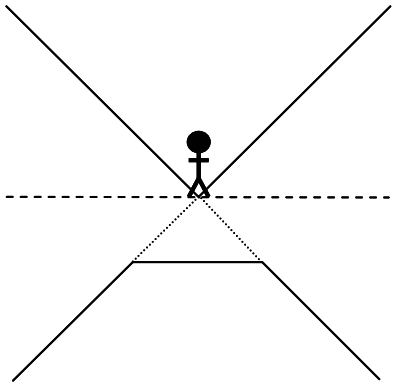


Figure 12. Adjusting the Lower Boundary to Allow Small Drops

While clamping the sensors to ground level, we should also determine if the surface is standable or not. We are already tracing a line to the surface while clamping so the information about whether or not the surface is standable should be readily available. If the surface is not standable, then we can remove the sensor from consideration as a cover location and save some computation cycles.

d. Determining if a Point is Standable

In order to determine if a point was standable, we checked to see if it had a slope of less than 45 degrees. The trace function that we used in clamping the points to ground level returns the surface normal of the first polygon it intersects. The surface normal has a length of one. At a slope of 0 degrees the z component of the surface normal is equal to one. At a slope of 45 degrees the z component of the surface normal is equal to the square root of two. So if the Z component of the surface normal is greater than the square root of two, we consider the surface to be standable. (See Figure 13 below)

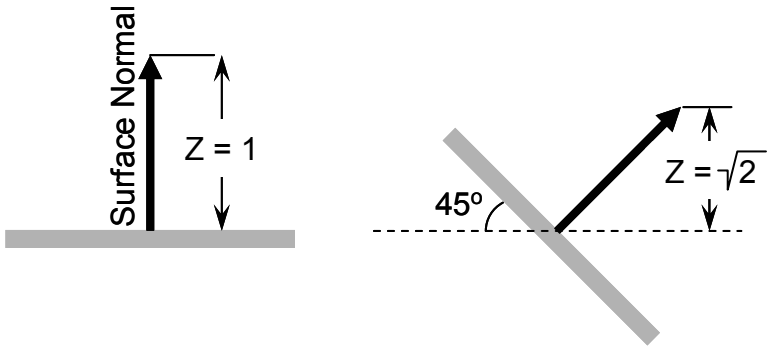


Figure 13. Determining Standability

e. Finding Cover

We check for cover by moving the sensor to the appropriate height above ground level and checking for line of sight. Start with the prone height for the agent and check if the sensor has line of sight. If line of sight exists, then there is no cover. If line of sight does not exist, the continue checking the next highest stance, until the line of sight check is clear or there is no line of sight to

the sensor at standing height. If the line of sight was clear to any sensor height, then the next lowest stance is the tallest stance that the agent can take at that location and still have cover. If line of sight is blocked even in the standing position, then the agent can take any posture at that location and still be in cover. Figure 14 below summarizes this procedure in pseudo code.

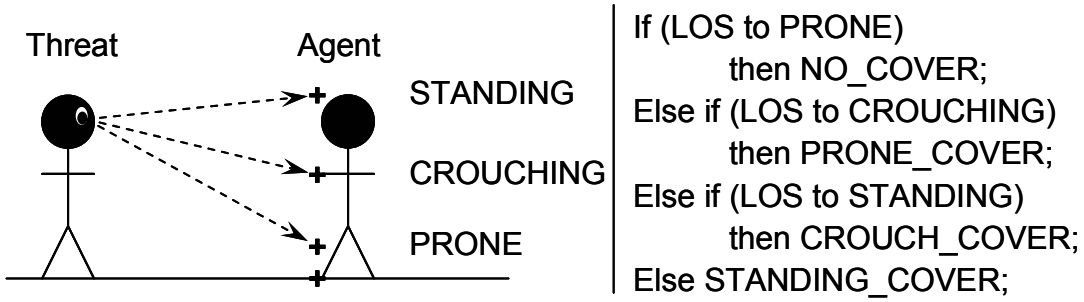


Figure 14. Determining Type of Cover with Sensors

f. Point-to-Point False Cover Results

Since we are checking line of sight from one point to another point, small objects and the edges of objects can appear to provide cover when they do not. This is because the point to point check does not take into account the area that the agent will occupy when it moves to that location. We avoided this problem by doing additional checks for locations that initially reported cover. We performed two more checks to points that we moved to the left and the right of the sensor by the collision radius of the agent. Since the collision cylinder for the agent is slightly larger than the actual polygon model for the object, this provides us with a conservative estimate of positions that will provide cover.

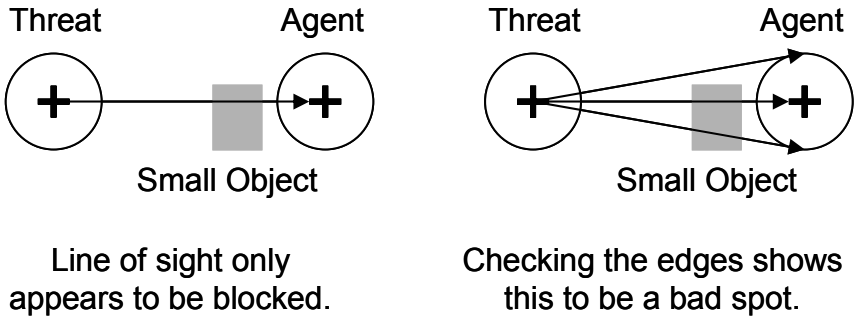


Figure 15. Checking for False Cover Results

g. Determining if a Point is Reachable

We considered a point reachable if a direct path or a path that went through one other sensor could reach it. We used two functions to determine if a point was reachable from another point. The first function, "pointReachable", is native to the Unreal Engine. It returns a Boolean value that tells you if the agent can move from its current location directly to a specified location in a straight line. However, this function cannot does not work if the starting point is not the agent's current location. To determine if the destination point could be reached from another sensor's location we used a volumetric line of sight check. This function determines if an upright cylinder that moves from one point to another intersects any objects.

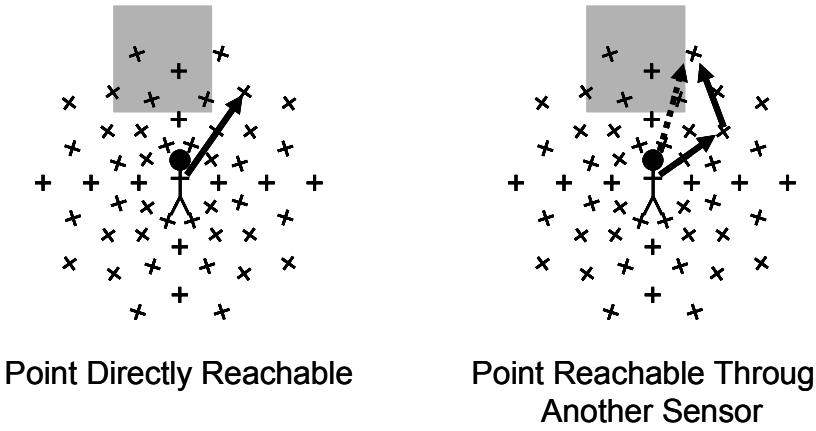


Figure 16. Checking if a Point is Reachable

h. Deciding Where to Go

Instead of implementing complicated decision logic about where the agent should take cover, we decided to go with a simple, effect solution. Our agent goes to the closest point in cover that it can reach.

4. Running the Demonstration

a. Loading the Environment

On the CD you will find a directory named "/FireTeam". Copy this directory directly to your hard drive. Once this is complete change to the "/system" directory and run the file called "ArmyOps.exe". This starts the game.

Once the menu screen is displayed, hit the "~" key to open a command prompt. To load the demonstration level, type "open test" at the command prompt. Once the level has loaded you will need to type "class r" to give yourself a rifle and unlock your movement controls.

b. Heads-Up Display

Figure 17. Heads-Up Display Features

c. Controls

Use the following commands to navigate through and interact with the environment:

ACTION	KEY
Move Forward	W
Move Back	S
Move Left	A
Move Right	D
Run	Ctrl
Jump	Spacebar
Look Left / Right / Up / Down	Move Mouse
Go From Standing or Prone to Kneeling	C
Go From Kneeling to Standing	C
Go From Standing or Kneeling to Prone	X
Go From Prone to Standing	X
Bring up Weapon Sights	Z
Fire Weapon	Left Mouse Button
Reload Weapon	R
Fix Jam	F
Open / Close the Command Consol	~
Toggle Between Main Menu and Demonstration	ESC

Table 1. Keyboard and Mouse Commands.

d. Consol Commands

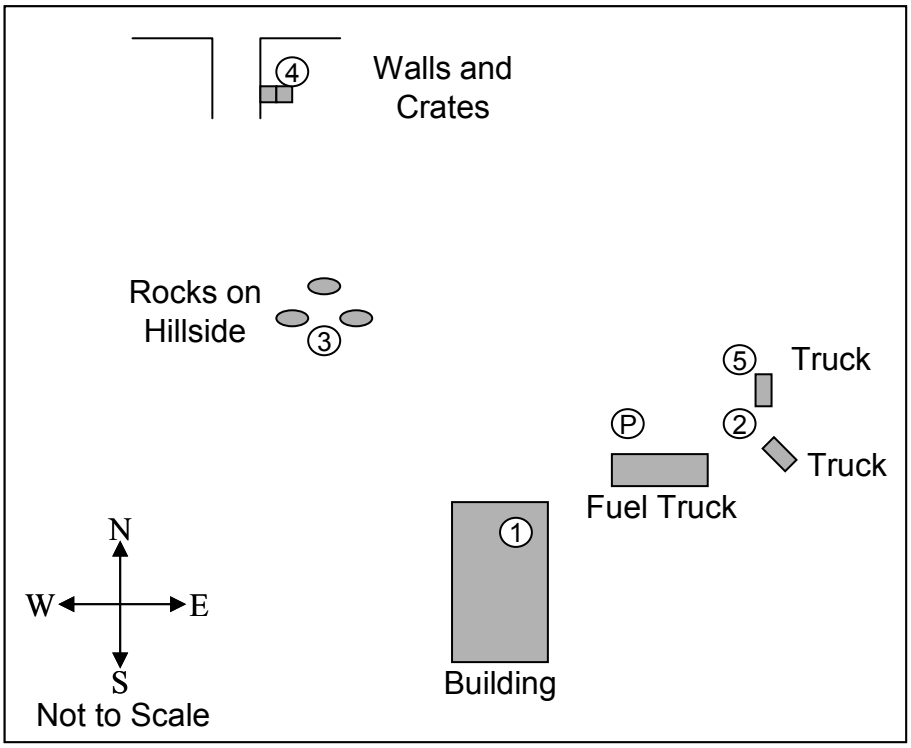
Use the following commands at the command prompt in the consol:

ACTION	COMMAND
Unlock Movement	playerlock 0
Change Player Class to Rifleman	class r
Give the Player Unlimited Ammo	mpcheat params ammo true
End the Demonstration	quit
Load the Demonstration	load test
Render Scene in Normal Mode	rmode 5
Render scene with polygons only	rmode 1

Table 2. Consol Commands.

e. Map of the Demonstration

The figure below shows a map of the demonstration area. This is not a map of the entire level. The Coverbots have been placed in a variety of locations so that its abilities can be tested. The compass in the lower left corner of the map shows where North is on the map.



P = Player starting position
1 = CoverBot
2 = CoverBot
3 = CoverBot
4 = CoverBot
5 = CoverBot2

Figure 18. Map of the Demonstration Level

f. CoverBot

CoverBot is a step-by-step implementation of the sensor grid algorithm. It starts in a waiting state with its sensor grid already deployed and clamped to the ground. The sensors appear as glowing balls of light that we call FireFlys. The trigger for making CoverBot execute the next step of the algorithm is to shoot him with your rifle.

The first time that you shoot CoverBot, he uses the sensor grid to find cover and plans a path to the shortest point in cover. He uses your location

from this first shot throughout the rest of the process until he resets. This allows you to move around him and see what is going on.

When CoverBot checks the sensor grid for line of sight the FireFlys, he makes the ones that are in your line of sight invisible. The FireFlys that are in cover are moved to a height that indicates whether that position provides cover prone, kneeling, or standing. The Firefly that is at the location where the CoverBot has decided to take cover is made six times larger than the others. If the CoverBot needs to go through one of the other FireFlys to get to that location it is made three times larger than the others and is shown regardless of whether or not it is in cover.

The second time that you shoot the CoverBot, it moves to the cover location that it decided on in the last step. It then assumes the posture necessary to have cover.

Shooting the CoverBot a third time resets it. It stands back up in its current position and redeploys its sensor grid.

g. CoverBotTwo

CoverBotTwo is a full speed version of CoverBot. Its sensor grid is not initially deployed and it does not use any FireFlys to show the location of its sensors. The first time you shoot CoverBotTwo, it immediately moves into cover and assumes the required posture. Shooting it a second time resets it.

Figure 19. CoverBot with Sensors Clamped to Ground Level

Figure 20. Sensors Showing Location Selected and Final Posture

Figure 21. CoverBot after Moving into Cover

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CODE

A. INTRODUCTION

This chapter includes all of the code that we used in our implementation of the sensor grid method of finding cover in the America's Army: Operation. All of the code is written in UnrealScript. Firefly.uc

B. FIREFLY.UC

```
/* ====================================================================
 * Filename:  FireFly.uc
 * Date:      19 SEP 2003
 * Author:    MAJ David J. Morgan, U.S. Army
 * ====================================================================
 * This displays a glowing ball of light in the level that does not
 * block anything.
 * ====================================================================
 */
class FireFly extends Light
    placeable;


defaultproperties
{
    bstatic=False
    bNoDelete=False
    bMovable=True
    bHidden=False
    bBlockBulletTraces=False
    bBlockNonZeroExtentTraces=False
    bBlockZeroExtentTraces=False
    Texture=Texture'T2-FX.Corona.fx2_cor_05'
    Style=STY_Translucent
}
```

C. NPC_COVERBOT.UC

```
/* ===================================================================
 * Filename:  NPC_CoverBot.uc
 * Date:      19 SEP 2003
 * Author:    MAJ David J. Morgan, U.S. Army
 * ===================================================================
 *  This class tells the engine which skins and what controller to use
 *  for the bot.
 * ===================================================================
 */
class NPC_CoverBot extends AGP_Character
    placeable;


function TakeDamage(int Damage, Pawn InstigatedBy, Vector HitLocation,
                    Vector Momentum, class<DamageType> DamageType,
                    optional BoneInfo Bone,
                    optional Controller KillerController)
{
    Controller.TakeDamage(Damage, InstigatedBy, HitLocation, Momentum,
                          DamageType, Bone, KillerController);
}


defaultproperties
{

    Skins[0]=Texture'T-Characters.Soldier.Soldier_PANTS_Tiger_Opfor'
    Skins[1]=Texture'T-Characters.Soldier.Soldier_SHIRT_Tiger_Opfor'
    Skins[2]=Texture'T-Characters.Soldier.Soldier_HAND_Blk_Gloves_1p'
    Skins[3]=Texture'T-Characters.Soldier.Soldier_FACE_Opfor_008'

    Mesh=Mesh'Soldier_3PMesh'

    bStaticSkinsAndMeshes = true;

    ControllerClass=class'AGP.CoverBotController'
}
```

D. COVERBOTCONTROLLER.UC

```
/* ===================================================================
 * Filename:  CoverBot2controller.uc
 * Date:      19 SEP 2003
 * Author:    MAJ David J. Morgan, U.S. Army
 * ===================================================================
 *    Base AI for CoverBots with step-by-step execution.  The bot
starts
 *  in a waiting state.  Glowing balls called "FireFlys" are used to
 *  indicate the position of the sensors in the sensor grid.  The
 *  trigger for the bot to change states is the takeDamage function.
 *  The first time the bot takes damage it searches the sensor grid for
 *  cover locations and  path into cover.  The sensors that are not in
 *  cover are made invisible.  The sensors that are in cover are moved
 *  to the height of the highest posture that the bot can assume at
 *  that location and still be in cover.  The FireFly for the sensor
 *  that the bot has chosen as its destination is made very large and
 *  if an intermediate sensor is needed to get there, it is made medium
 *  sized.  When the bot takes damage a second time it moves to the
 *  cover location that it chose initially and assumes the correct
 *  posture.  When it takes damage a third time it resets itself into
 *  the waiting state.
 * ===================================================================
 */

class CoverBotController extends AgentController;

/************************/
/*        VARIABLES     */
/************************/
var Color    red, green, blue, purple; // Colors for messages
var int      displayTime;              // Display time messages
var FireFly  fireFly[51];  // need (num_rings * num_points) + 1
var int      num_rings;    // number of rings of FireFlys
var int      num_points;   // number of FireFlys per ring
var int      num_fireFlys; // number of FireFlys actually created
var float    distance_between_rings;
var Actor    midpoint, endpoint; // path to the point in cover
var bool     foundCover, foundPath;
var int      middle, end;  // used for navigation

var enum CoverType
{
    Standing,
    Crouching,
    Prone,
    NoCover
} cover[51]; // need (num_rings * num_points) + 1 of these
// there is a 1-to-1 correspondence between the index of the cover and
// the sensor
```

```
/************************/
/*        PREGAMEPLAY   */
/************************/
function PreBeginPlay()
{
    Super.PreBeginPlay();
}

function BeginPlay()
{
    Super.BeginPlay();
}

function PostBeginPlay()
{
    Super.PostBeginPlay();
}


/*********************************************************************
* This method does some initialization of some variables when the
* CoverBot is created.
*********************************************************************/
function Possess(Pawn aPawn)
{
    Super.Possess(aPawn);

    // set colors for printing messages to the console
    red.R    = 255; red.G    = 0;   red.B    = 0;   red.A    = 255;
    green.R  = 0;   green.G  = 255; green.B  = 0;   green.A  = 255;
    blue.R   = 0;   blue.G   = 0;   blue.B   = 255; blue.A   = 255;
    purple.R = 255; purple.G = 0;   purple.B = 255; purple.A = 255;

    // set time for messages to be displayed in the console
    displayTime = 20;
}


/*********************************************************************
* This is a utility method that makes it easier to print console
* messages to the screen.
* param     message     the text message to display
* param     textColor   the text color to display the message in
* returns   nothing
*********************************************************************/
function PrintToConsole(String message, Color textColor)
{
    Level.GetClientController().Player.Console.Message(message,
        displayTime,textColor);
}
```

```
/*******************************************************************
* This function builds the sensor grid.  The sensors (FireFlys) are
* placed in their initial positions and clamped to ground level.
* param     none
* returns   none
*******************************************************************/
function CreateSensorGrid()
{
    local int     i, j;     // counters
    local rotator r;        // direction to current point
    local float   distance; // distance to the current ring
    local vector  fireFlyLocation;
    local float   elevation;

    // set the initital positions for all of the FireFlys
    distance = 0;

    // build one FireFly at my feet
    fireFly[0] = Spawn(class'FireFly', Pawn, ,
        Pawn.Location-MakeVect(0,0,Pawn.CollisionHeight-2),
        Pawn.Rotation);
    fireFly[0].SetDrawScale(0.1);        // makes them small
    fireFly[0].LifeSpan = 0;             // makes them permenant
    num_fireFlys = 1;

    // build the rest of the FireFlys in the grid
    for (i=0; i<num_rings; i++) // builds each ring
    {
        // set rotation to straight forward relative to Pawn
        r = rotation;
        // stagger the odd numbered rings
        if (i%2==1)
            r.yaw = r.yaw + (65500/num_points)/2;
        distance = distance + distance_between_rings;
        // place FireFlys for the current ring
        for (j=0; j<num_points; j++)
        {
            // find direction to next Firefly
            r.yaw = r.yaw + 65500/num_points;
            // move it out
            fireFlyLocation = Pawn.Location + distance*vector(r);
            // set it on the ground
            fireFlyLocation.z -= Pawn.collisionHeight;

            // don't make a FireFly unless there is something for it
            // to stand on
            if (clampSensorToGround(FireFlyLocation, distance,
                                    elevation))
            {
                fireFlyLocation.z = elevation;
                fireFly[num_FireFlys] = Spawn(class'FireFly', Pawn, ,
                    FireFlyLocation, Pawn.Rotation);
                // make them small
                fireFly[num_FireFlys].SetDrawScale(0.1);
                // make them permenant
                fireFly[num_FireFlys].LifeSpan = 0;
                num_fireFlys++;
```

```
            }
        }
    }
} // End function createSensorGrid()


/*********************************************************************
* This function clamps a sensor to ground level and tests if the
* surface is standable.  If no surface exists or the surface is
* not standable the function returns false to indicate that a
* sensor should not be created.
* param     ffLocation        starting location for the sensor
* param     distanceFromPawn  how far the sensor is from the Agent
* param     elevation         the final elevation of the sensor
* returns   true if a sensor should be created at the passed location
*********************************************************************/
function bool ClampSensorToGround(vector ffLocation,
                                  float distanceFromPawn,
                                  out float elevation)
{
    local vector hitLocation; // required by the trace funtion
    local vector hitNormal;   // required by the trace funtion
    local float  verticalDistance, pawnHeight;
    local vector verticalVector;
    local Actor  hitActor;

    // create a vector that can be used to determine where a line is
    // that extends from the agent at a 45 degree above and below the
    // horizontal plane.  0.785398163397 is 45 degrees in radians
    verticalDistance = distanceFromPawn*Tan(0.785398163397);
    verticalVector = MakeVect(0,0,verticalDistance);

    // have to use default collision height, current may be different
    PawnHeight = 2*Pawn.default.CollisionHeight;

    hitActor = None;

    // First Check:  Trace from the Sensor's Location down to the line
    if (verticalDistance < PawnHeight) // allow drops up to it's Height
        hitActor = Pawn.Trace(hitLocation, hitNormal,
                              ffLocation-MakeVect(0,0,PawnHeight),
                              ffLocation);
    else
        hitActor = Pawn.Trace(hitLocation, hitNormal,
                              ffLocation-verticalVector,
                              ffLocation);

    // If nothing was hit
    // Second Check:  Trace from upper bound down
    if (hitActor == None)
        hitActor = Pawn.Trace(hitLocation, hitNormal,
                              ffLocation-vect(0,0,1),
                              ffLocation+verticalVector);
```

```
    // If still haven't hit anything
    if (hitActor == None)
        return false;
    else
    {
        // check if the agent can stand on the surface
        if (Standable(HitNormal))
        {
            // The 2 is needed to make the FireFlys visible
            elevation = HitLocation.z+2;
            return true;
        }
        else
            return false;
    }
} // End function ClampSensorToGround


/*******************************************************************
* This function determines if the agent can stand on the surface.
* param     normal     the surface normal
* returns   true if the surface is flat enough for the bot to stand.
*******************************************************************/
function bool Standable(vector normal)
{
    // normal is a unit vector (length of 1), if the z component
    // of the normal is > 0.7071 then the surface has a slope
    // of less than 45 degrees
    if (normal.z>0.7071)
        return True;
    else
        return False;
} // End function Standable


/*******************************************************************
* This function determines if there is line of sight from the threat
* to the target.
* param     threat     location of the threat
* param     target     location of the target
* param     radius     the radius of the target
* returns   true if there is line of sight to the location of target
*           or points radius units to the left or right of the target
*******************************************************************/
function bool CanBeSeen(vector threat, vector target, float radius)
{
    local vector  AtoB;     // a vector from the threat to the target
    local vector  sideStep; // vector used to find points to the sides
    local rotator r;

    if (FastTrace(threat, target))
    {
        // Line of sight to center of target, return True
        return True;
    }
    else // Check points to the left and right
    {
```

```
        AtoB = target - threat;
        r = Rotator(AtoB);
        // Unreal has 65500 units in a circle

        // this turns the vector 90 degrees
        r.yaw = r.yaw + 65500/4;

        // Create a unit vector in the direction of r
        sideStep = Vector(r);

        // widen the sidestep to radius
        sideStep = sideStep * radius;

        if (FastTrace(threat, target+sideStep) ||
            FastTrace(threat,target-sideStep))
        {
            // if either of these can be seen then return True
            return True;
        }
        else
        {
            return False;
        }
    }
} // end function CanBeSeen


/*****************************************************************
* This function determines if any of the sensors in the grid are in
* cover.  If they are in cover, it determines the tallest posture
* that the agent can assume in that position.  The results are stored
* in the cover array.  All sensors are reset to waist level for the
* agent to aid in navigation in the next step.
* param     none
* returns   true if cover was found
*****************************************************************/
function bool FindCover()
{
    local int    i;         // counter
    local vector enemyEyes; // the location of the threat's viewpoint
    local bool   result;    // true if cover found

    result = false;

    enemyEyes = Enemy.EyePosition();
    for (i=0; i<num_FireFlys; i++)
    {
        // Check if there is cover, then set the height of the FireFlys
        // appropriately
        if (CanBeSeen(enemyEyes,
            FireFly[i].Location+MakeVect(0,0,2*Pawn.proneHeight-2),
            collisionRadius))
        {
           Cover[i]=NoCover; // if you can't take cover prone there is
                             // no cover
           FireFly[i].SetDrawScale(0.0); // don't want to see them
           // move the point up so it can be used for navigation
```

```
            FireFly[i].Move(MakeVect(0,0,Pawn.default.CollisionHeight));
        }
        else if (CanBeSeen(enemyEyes,
            FireFly[i].Location+MakeVect(0,0,2*Pawn.crouchHeight-2),
            collisionRadius))
        {
            result=true;
            Cover[i]=Prone;
            FireFly[i].Move(MakeVect(0,0,2*Pawn.ProneHeight-2));
        }
        else if (CanBeSeen(enemyEyes,
            FireFly[i].Location+MakeVect(0,0,
                2*Pawn.default.collisionHeight-2),collisionRadius))
        {
            result=true;
            Cover[i]=Crouching;
            FireFly[i].Move(MakeVect(0,0,2*Pawn.CrouchHeight-2));
        }
        else
        {
            result=true;
            Cover[i]=Standing;
            FireFly[i].Move(MakeVect(0,0,
                2*Pawn.default.CollisionHeight-2));
        }
    }
    return result;
} // End function FindCover()


/*********************************************************************
* This function decides where the coverbot should go based on which
* sensors are in cover and can be reached by either a straight line
* path or by a two part path through another sensor in the grid.  The
* coverbot always tries to reach the closest point in cover.
* param     none
* returns   true if a path to the cover was found
*********************************************************************/
function bool findPathToCover()
{
    local int    i, j;              // counters
    local float  min_distance;      // the current closest distance
    local float  distanceToPoint;
    local vector hitLocation;       // required by trace function
    local vector hitNormal;         // required by trace function
    local vector extent;            // size of the collision cylinder
    local int    directPathTo[51]; //0=False, 1 = True

    // set the colision volume for traces
    Extent = Pawn.GetCollisionExtent();

    // Find all the points that are directly reachable and mark them.
    // By doing this one time in the beginning and storing the result
    // we save repeating it over and over again when looking for
    // two-part paths
    for (i=0; i<num_fireFlys; i++)
    {
```

```
    if (pointReachable(fireFly[i].Location))
        directPathTo[i]=1;
    else
        directPathTo[i]=0;
}

// Find out which point we want to go to
min_distance = 9999.0;          // set initially very large
for (i=0; i<num_fireFlys; i++)
{
    // only check the point if it provides cover
    if (cover[i]!=NoCover)
    {
        //if there is a direct path
        if (directPathTo[i]==1)
        {
            distanceToPoint =
                VSize(fireFly[i].Location-Pawn.Location);
            if(distanceToPoint < min_Distance)
            {
            // if this is the best point so far, set destination
                middle = i;
                end    = i;
                return True; // Don't need to check anything else
            }
        }
        // else check if the point is reachable indirectly
        else
        {
            // go through all other points
            for (j=0; j<num_fireFlys; j++)
            {
                // only check midpoints that are reachable,
                // don't check a sensor against itself,
                // don't check the agent's current location
                if((directPathTo[j]==1) && (i!=j)  && (j!=0))
                {
                    // see if the second point is reachable
                    // from the first
                    if (Pawn.Trace(HitLocation, HitNormal,
                        FireFly[i].Location, FireFly[j].Location,
                        , , Extent)==None)
                    {
                        distanceToPoint =
                           VSize(Pawn.Location-FireFly[j].Location)
                  + VSize(FireFly[j].Location-FireFly[i].Location);
                        // if it is the closest point set it as the
                        // destination
                        if (distanceToPoint < min_Distance)
                        {
                            min_Distance = distanceToPoint;
                            middle = j;
                            end    = i;
                        }
                    } // end if
                } // end if
            } // end for j
```

```
            } // end if/else
        } // end if point provides cover
    } // end for i
    if (min_distance>9998.0) // > used to avoid floating point errors
        return False;
    else
        return True;
} // end function FindPathToCover


/*********************************************************************
* This function resets all of the fireFlys to the same size.
* param     newSize     the new size to make the fireFlys
* returns   nothing
*********************************************************************/
function ResizeFireFlys(float newSize)
{
    local int a;

    printToConsole("Resizing FireFlys: " $ newSize, Purple);

    for (a=0; a<num_FireFlys; a++)
        FireFly[a].SetDrawScale(newSize);
} // End function resizeFireFlys()


/*********************************************************************
* This function destroys all of the current fireFlys.
* returns   nothing
*********************************************************************/
function DestroyFireFlys()
{
    local int i;
    for (i=0; i<num_fireFlys; i++)
        fireFly[i].destroy();
    num_fireFlys = 0;
} // End function DestroyFireFlys


/*********************************************************************
* This function highlights the path chosen by the bot by increasing
* the size of the fireFlys that it is using to navigate.
* returns   nothing
*********************************************************************/
function ShowPath()
{
    fireFly[middle].SetDrawScale(0.3);
    fireFly[end].SetDrawScale(0.6);
}
```

```
/********************/
/*        STATES    */
/********************/

/*****************************************************************
* This the default starting state for the bot.  The fireFlys for the
* sensor grid are created and clamped to ground level.  The bot
* changes to the FindCoverState when it takes damage.
*****************************************************************/
auto state WaitingState
{
    ignores ShootTarget, NotifyTakeHit, Killed, NotifySeePawn, SeePawn,
        SeePlayer, SeeMonster, HearNoise;

    function TakeDamage(int damage, Pawn instigatedBy,
                        Vector hitLocation, Vector momentum,
                        class<DamageType> damageType,
                        optional BoneInfo bone,
                        optional Controller killerController )
    {
        enemy = instigatedBy;
        PrintToConsole("Took Damage from "$Enemy.OwnerName, red);
        GotoState('FindCoverState');
    }

Begin:
    PrintToConsole("WaitingState - Begin", red);
    Pawn.shouldStand(true);
    CreateSensorGrid();
End:
} // End WaitingState


/*****************************************************************
* When entering this state the bot searches the sensor grid for
* cover locations and decides which cover location it will move to.
* When the bot takes damage again it changes to the MoveToCoverState.
*****************************************************************/
state FindCoverState
{
    ignores ShootTarget, NotifyTakeHit, Killed, NotifySeePawn, SeePawn,
        SeePlayer, SeeMonster, HearNoise;

    function TakeDamage(int damage, Pawn instigatedBy,
                        Vector hitLocation, Vector momentum,
                        class<DamageType> damageType,
                        optional BoneInfo bone,
                        optional Controller killerController )
    {
        Enemy = instigatedBy;
        PrintToConsole("Took Damage from "$enemy.OwnerName, blue);
        GotoState('MoveToCoverState');
    }

Begin:
    PrintToConsole("FindCoverState - Begin", blue);
    foundCover = FindCover();
```

```
    if (foundCover)
        foundPath = FindPathToCover();
    else
        foundPath = False;

    if (foundCover)
         PrintToConsole("FoundCover = TRUE", Purple);
    else PrintToConsole("FoundCover = FALSE", Purple);
    if (foundPath)
    {
        PrintToConsole("FoundPath  = TRUE", Purple);
        ShowPath();
    }
    else PrintToConsole("FoundPath  = FALSE", Purple);
End:
} // End state FindCoverState


/*******************************************************************
* When entering this state the bot uses the path generated by the
* last state to move into cover.  When it takes damage again, it
* resets itself and moves back into the waitingState.
*******************************************************************/
state MoveToCoverState
{
    ignores ShootTarget, NotifyTakeHit, Killed, NotifySeePawn, SeePawn,
        SeePlayer, SeeMonster, HearNoise;

    function TakeDamage(int damage, Pawn instigatedBy,
                        Vector hitLocation, Vector momentum,
                        class<DamageType> damageType,
                        optional BoneInfo bone,
                        optional Controller killerController )
    {
        enemy = InstigatedBy;
        PrintToConsole("Took Damage from "$enemy.OwnerName, green);
        DestroyFireFlys();
        GotoState('WaitingState');
    }

Begin:
    PrintToConsole("MoveToCoverState - Begin", Green);
    if (FoundPath)
    {
        MoveTo(firefly[middle].Location);
        MoveTo(firefly[end].Location);

        focus = enemy;
        FinishRotation();

        if (cover[end]==Crouching)
            Pawn.shouldCrouch(true);
        else if (cover[end]==Prone)
            Pawn.shouldProne(true);
        else if (cover[end]==NoCover)
            PrintToConsole("Error - No Cover where I am moving to",
                           Purple);
```

```
    }
    else PrintToConsole("Nowhere to run to Baby!  Nowhere to Hide!",
                        Purple);

    Pawn.desiredRotation = Rotator(Enemy.Location-Pawn.Location);
End:
} // End state MoveToCoverState


/************************
    DEFAULT PROPERTIES
************************/
defaultproperties
{
    num_rings = 5;
    num_points = 10;
    distance_between_rings = 100.0;
}
```

E. NPC_COVERBOTTWO.UC

```
/* ===================================================================
 * Filename:  NPC_CoverBotTwo.uc
 * Date:      19 SEP 2003
 * Author:    MAJ Morgan
 * ===================================================================
 * This class tells the engine which skins and what controller to
 * use for the bot.
 * ===================================================================
 */
class NPC_CoverBotTwo extends AGP_Character
    placeable;


function TakeDamage(int Damage, Pawn InstigatedBy, Vector HitLocation,
                    Vector Momentum, class<DamageType> DamageType,
                    optional BoneInfo Bone,
                    optional Controller KillerController)
{
    Controller.TakeDamage(Damage, InstigatedBy, HitLocation, Momentum,
                          DamageType, Bone, KillerController);
}


defaultproperties
{
    Skins[0]=Texture'T-Characters.Soldier.Soldier_PANTS_Tiger_Opfor'
    Skins[1]=Texture'T-Characters.Soldier.Soldier_SHIRT_Tiger_Opfor'
    Skins[2]=Texture'T-Characters.Soldier.Soldier_HAND_Blk_Gloves_1p'
    Skins[3]=Texture'T-Characters.Soldier.Soldier_FACE_Opfor_008'

    Mesh=Mesh'Soldier_3PMesh'

    bStaticSkinsAndMeshes = true;

    ControllerClass=class'AGP.CoverBot2Controller'
}
```

F. COVERBOT2CONTROLLER.UC

```
/* ===================================================================
 * Filename:  CoverBot2controller.uc
 * Date:      19 SEP 2003
 * Author:    MAJ David J. Morgan, U.S. Army
 * ===================================================================
 *    Base AI for CoverBots with full speed execution.  The trigger for
 *  the bot to change states is shooting it.  When this bot is shot
 *  the first time it looks for cover and tries to move there.  If no
 *  cover is found, it stays where it is and prints a message to the
 *  screen.  When the bot is shot a second time it resets itself.
 * ===================================================================
 */

class CoverBot2Controller extends AgentController;

/************************/
/*        VARIABLES     */
/************************/
var Color       red, green, blue, purple; // Colors for messages
var int         displayTime;              // Display time for messages
var vector      sensor[51];
var int         num_rings;                // number of rings of sensors
var int         num_points;               // number of sensors per ring
var int         num_sensors;              // number of sensors created
var float       distance_between_rings;
var Actor       midpoint, endpoint;       // path to the point in cover
var bool        foundCover, foundPath;
var int         middle, end;

var enum CoverType
{
    Standing,
    Crouching,
    Prone,
    NoCover
} cover[51]; // need (num_rings * num_points) + 1 of these
// there is a 1-to-1 correspondance between the index of the cover and
// the sensor

/************************/
/*        PREGAMEPLAY   */
/************************/
function PreBeginPlay()
{
    Super.PreBeginPlay();
}

function BeginPlay()
{
    Super.BeginPlay();
}
```

```
function PostBeginPlay()
{
    Super.PostBeginPlay();
}

/*******************************************************************
* This method does some initialization of some variables when the
* CoverBot is created.
*******************************************************************/
function Possess(Pawn aPawn)
{
    Super.Possess(aPawn);

    // set colors for printing messages to the console
    red.R    = 255; red.G    = 0;   red.B    = 0;   red.A    = 255;
    green.R  = 0;   green.G  = 255; green.B  = 0;   green.A  = 255;
    blue.R   = 0;   blue.G   = 0;   blue.B   = 255; blue.A   = 255;
    purple.R = 255; purple.G = 0;   purple.B = 255; purple.A = 255;

    // set time for messages to be displayed in the console
    displayTime = 20;
}

/*******************************************************************
* This is a utility method that makes it easier to print console
* messages to the screen.
* param     message     the text message to display
* param     textColor   the text color to display the message in
* returns   nothing
*******************************************************************/
function PrintToConsole(String message, Color textColor)
{
    Level.GetClientController().Player.Console.Message(message,
        displayTime,textColor);
}

/*******************************************************************
* This function builds the sensor grid.  The sensors are placed in
* their initial positions and clamped to ground level.
* param     none
* returns   none
*******************************************************************/
function CreateSensorGrid()
{
    local int     i, j;           // counters
    local rotator r;              // direction to current point
    local float   distance;       // distance to the current ring
    local vector  sensorLocation;
    local float   elevation;

    // set the initital positions for all of the Sensors
    distance = 0;

    // build one Sensor at the feet of the agent
    sensor[0] = Pawn.Location-MakeVect(0,0,Pawn.CollisionHeight);
    num_Sensors = 1;
```

```
    // build the rest of the Sensors in the grid
    for (i=0; i<num_rings; i++) // builds each ring
    {
        // set rotation to straight forward relative to CoverBo
        r = rotation;
        // stagger the odd numbered rings
        if (i%2==1)
            r.yaw = r.yaw + (65500/num_points)/2;
        distance = distance + distance_between_rings;
        // places Sensors for the current ring
        for (j=0; j<num_points; j++)
        {
            // find direction to next Sensor
            r.yaw = r.yaw + 65500/num_points;
            // move it out from center
            sensorLocation = Pawn.Location + distance*vector(r);
            // set it at foot level
            sensorLocation.z -= Pawn.collisionHeight;

            // if the sensor successfully clamped to the ground,
            // build it
            if (ClampSensorToGround(sensorLocation,distance,elevation))
            {
                sensorLocation.z = elevation;
                Sensor[num_Sensors] = SensorLocation;
                num_sensors++;
            }
        }
    }
} // End function CreateSensorGrid()


/*******************************************************************
* This function clamps sensors to ground level and tests if the
* surface is standable.  If no surface exists or the surface is
* not standable the function returns false to indicate that a
* sensor should not be created.
* param     sensorLocation    starting location for the sensor
* param     distanceFromPawn  how far the sensor is from the Agent
* param     elevation         the final elevation of the sensor
* returns   true if a sensor should be created at the passed location
*******************************************************************/
function bool ClampSensorToGround(vector sensorLocation,
                                  float distanceFromPawn,
                                  out float elevation)
{
    local vector hitLocation, hitNormal;
    local float  verticalDistance, pawnHeight;
    local vector verticalVector;
    local Actor  hitActor;

    // create a vector that can be used to determine where a line is
    // that extends from the agent at a 45 degree above and below the
    // horizontal plane.  0.785398163397 is 45 degrees in radians
    verticalDistance = distanceFromPawn*Tan(0.785398163397);
    verticalVector = MakeVect(0,0,verticalDistance);
```

```
    // have to use default, current height may be different
    pawnHeight = 2*Pawn.default.CollisionHeight;

    hitActor = None;

    // First Check:  Trace from the Sensor's Location down to the line
    if (verticalDistance < pawnHeight) // allow drops up to it's height
        hitActor = Pawn.Trace(hitLocation, hitNormal,
                              sensorLocation-MakeVect(0,0,PawnHeight),
                              sensorLocation);
    else
        hitActor = Pawn.Trace(HitLocation, HitNormal,
                              sensorLocation-verticalVector,
                              sensorLocation);

    // If nothing was hit
    // Second Check:  Trace from upper bound down
    if (hitActor == None)
        hitActor = Pawn.Trace(HitLocation, HitNormal,
                              sensorLocation-vect(0,0,1),
                              sensorLocation+verticalVector);

    // If still haven't hit anything
    if (hitActor == None)
        return false;
    else
    {
        // check if the agent can stand on the surface
        if (Standable(HitNormal))
        {
            elevation = HitLocation.z;
            return true;
        }
        else
            return false;
    }
} // End function ClampSensorToGround


/*******************************************************************
* This function determines if the agent can stand on the surface.
* param     normal     the surface normal
* returns   true if the surface is flat enough for the bot to stand.
*******************************************************************/
function bool Standable(vector normal)
{
    // normal is a unit vector (length of 1), if the z component
    // of the normal is > 0.7071 then the surface has a slope
    // of less than 45 degrees
    if (normal.z>0.7071)
        return True;
    else
        return False;
} // End function Standable
```

```
/*******************************************************************
* This function determines if there is line of sight from the threat
* to the target.
* param     threat     location of the threat
* param     target     location of the target
* param     radius     the radius of the target
* returns   true if there is line of sight to the location of target
*           or points radius units to the left or right of the target
*******************************************************************/
function bool CanBeSeen(vector threat, vector target, float radius)
{
    local vector  AtoB;     // a vector from the threat to the target
    local vector  sideStep; // vector used to find points to the sides
    local rotator r;

    if (FastTrace(threat, target))
    {
        // Line of sight to center of target, return True
        return True;
    }
    else // Check points to the left and right
    {
        AtoB = target - threat;
        r = Rotator(AtoB);
        // Unreal has 65500 units in a circle

        // this turns the vector 90 degrees
        r.yaw = r.yaw + 65500/4;

        // Create a unit vector in the direction of r
        sideStep = Vector(r);

        // widen the sidestep to radius
        sideStep = sideStep * radius;

        if (FastTrace(threat, target+sideStep) ||
            FastTrace(threat,target-sideStep))
        {
            // if either of these can be seen then return True
            return True;
        }
        else
        {
            return False;
        }
    }
} // end function CanBeSeen
```

```
/*******************************************************************
* This function determines if any of the sensors in the grid are in
* cover.  If they are in cover, it determines the tallest posture
* that the agent can assume in that position.  The results are stored
* in the cover array.  All sensors are reset to waist level for the
* agent to aid in navigation in the next step.
* param     none
* returns   true if cover was found
*******************************************************************/
function bool FindCover()
{
    local int    i;
    local vector enemyEyes;
    local bool   result;

    result = false;

    enemyEyes = Enemy.EyePosition();
    for (i=0; i<num_Sensors; i++)
    {
        // Check if there is cover
        if (CanBeSeen(enemyEyes,
                      Sensor[i]+MakeVect(0,0,2*Pawn.proneHeight),
                      collisionRadius))
        {
            Cover[i]=NoCover; // if you can't take cover prone
                              // there is no cover
        }
        else if (CanBeSeen(enemyEyes,
                           Sensor[i]+MakeVect(0,0,2*Pawn.crouchHeight),
                           collisionRadius))
        {
            result=true;
            Cover[i]=Prone;
        }
        else if (CanBeSeen(enemyEyes,
                   Sensor[i]+MakeVect(0,0,2*Pawn.default.collisionHeight),
                   collisionRadius))
        {
            result=true;
            Cover[i]=Crouching;
        }
        else
        {
            result=true;
            Cover[i]=Standing;
        }
        // move up for navigation checks
        sensor[i].z+=Pawn.default.CollisionHeight;
    }
    return result;
} // End function FindCover()
```

```
/*****************************************************************
* This function decides where the coverbot should go based on which
* sensors are in cover and can be reached by either a straight line
* path or by a two part path through another sensor in the grid.  The
* coverbot always tries to reach the closest point in cover.
* param     none
* returns   true if a path to the cover was found
*****************************************************************/
function bool FindPathToCover()
{
    local int    i, j;          // counters
    local float  min_distance;
    local float  distanceToPoint;
    local vector hitLocation, hitNormal, extent;
    local int    directPathTo[51]; //0=False, 1 = True,

    // sets the collision volume for traces
    extent = Pawn.GetCollisionExtent();

    // Find all the points that are directly reachable and mark them.
    // By doing this one time in the beginning and storing the result
    // we save repeating it over and over again when looking for
    // two-part paths
    for (i=0; i<num_sensors; i++)
    {
        if (pointReachable(sensor[i]))
            directPathTo[i]=1;
        else
            directPathTo[i]=0;
    }

    // Find out which point we want to go to
    // initially set VERY high so it's east to test
    min_distance = 9999.0;
    for (i=0; i<num_Sensors; i++)
    {
        // only check the point if it provides cover
        if (cover[i]!=NoCover)
        {
            //if there is a direct path
            if (directPathTo[i]==1)
            {
                distanceToPoint = VSize(Sensor[i]-Pawn.Location);
                if(distanceToPoint < min_distance)
                {
                // if this is the best point so far, set destination
                    middle = i;
                    end    = i;
                    return True;  // Don't need to check anything else
                    // Since we check inside-out the first direct path
                    // that we find that is less than the current
                    // min_distance is guaranteed to be the closest
                    // point
                }
            }
            // else check if the point is reachable indirectly
            else
```

```
            {
                // go through all other points
                for (j=0; j<num_Sensors; j++)
                {
                    // only check midpoints that are reachable,
                    // don't check a point against itself
                    // don't check where the coverbot is already
                    // standing
                    if((directPathTo[j]==1) && (i!=j)  && (j!=0))
                    {
                        // see if the second point is reachable
                        // from the first
                        if (Pawn.Trace(HitLocation, HitNormal,
                                       sensor[i], sensor[j], , ,
                                       Extent)==None)
                        {
                            distanceToPoint =
                                VSize(Pawn.Location-sensor[j])+
                                VSize(sensor[j]-sensor[i]);
                            // if it is the closest point set it as
                            // the destination
                            if (distanceToPoint < min_distance)
                            {
                                min_distance = distanceToPoint;
                                middle = j;
                                end    = i;
                            }
                        } // end if
                    } // end if
                } // end for j
            } // end if/else
        } // end if point provides cover
    } // end for i
    if (min_distance>9998.0) // >9998 to avoid floating point errors
        return False;
    else
        return True;
} // end function FindPathToCover
```

```
/*******************/
/*        STATES    */
/*******************/

/*****************************************************************
* This is the initial starting state where the bot just stands there
* and does nothing.  If the bot takes damage from someone, it sets
* them to be its enemy and switches to TakeCoverState.
*****************************************************************/
auto state WaitingState
{
    ignores ShootTarget, NotifyTakeHit, Killed, NotifySeePawn, SeePawn,
        SeePlayer, SeeMonster, HearNoise;

    function TakeDamage(int Damage, Pawn InstigatedBy,
                        Vector HitLocation, Vector Momentum,
                        class<DamageType> DamageType,
                        optional BoneInfo Bone,
                        optional Controller KillerController )
    {
        Enemy = InstigatedBy;
        PrintToConsole("Took Damage from "$Enemy.OwnerName, Red);
        GotoState('FindCoverState');
    }

Begin:
    PrintToConsole("WaitingState - Begin", Red);
    Pawn.shouldStand(true);
End:

} // End WaitingState


/*****************************************************************
* In this state the bot attempts to find cover and move to that
* location.  If it does not find cover, it does nothing but print
* a message to the screen.  When the bot takes damage again it
* switches to WaitingState again.
*****************************************************************/
state FindCoverState
{
    ignores ShootTarget, NotifyTakeHit, Killed, NotifySeePawn, SeePawn,
        SeePlayer, SeeMonster, HearNoise;

    function TakeDamage(int Damage, Pawn InstigatedBy,
                        Vector HitLocation, Vector Momentum,
                        class<DamageType> DamageType,
                        optional BoneInfo Bone,
                        optional Controller KillerController )
    {
        Enemy = InstigatedBy;
        PrintToConsole("Took Damage from "$Enemy.OwnerName, Blue);
        GotoState('WaitingState');
    }

Begin:
    PrintToConsole("FindCoverState - Begin", Blue);
```

```
    CreateSensorGrid();
    FoundCover = FindCover();
    if (FoundCover)
        FoundPath = FindPathToCover();
    else
        FoundPath = False;

    if (FoundCover)
         PrintToConsole("FoundCover = TRUE", Purple);
    else PrintToConsole("FoundCover = FALSE", Purple);

    if (FoundPath)
    {
        PrintToConsole("FoundPath  = TRUE", Purple);
    }
    else PrintToConsole("FoundPath  = FALSE", Purple);

    if (FoundPath)
    {
        MoveTo(Sensor[middle]);
        MoveTo(Sensor[end]);

        Focus = Enemy;
        FinishRotation();

        if (cover[end]==Crouching)
            Pawn.shouldCrouch(true);
        else if (cover[end]==Prone)
            Pawn.shouldProne(true);
        else if (cover[end]==NoCover)
            PrintToConsole("Error - No Cover where I am moving to",
                           Purple);
    }
    else PrintToConsole("Nowhere to run to Baby!  Nowhere to Hide!",
                        Purple);

End:
} // End state FindCoverState


/************************
    DEFAULT PROPERTIES
************************/
defaultproperties
{
    num_rings = 5;
    num_points = 10;
    distance_between_rings = 100.0;
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

VII. CONCLUSIONS AND FUTURE WORK

A. INTRODUCTION

In this chapter, we summarize some of the things that we learned during the development of these algorithms and present several ideas for future areas of research.

B. CONCLUSIONS

1. Shadow Volume Binary Space Partition Tree

The study of the SVBSP Tree algorithm proved to be very rewarding even though it does not appear to be a viable solution at this time. It has the greatest potential of the three algorithms to provide an accurate solution to finding cover in a scene. However, it also has significant difficulties that may keep it from being implemented any time in the near future.

Generating a cover volume is both the strength and the weakness of the SVBSP tree. The way that the SVBSP tree builds the shadow volume provides the most accurate representation of the cover. At the same time, this process makes it extremely difficult to determine where the agent can fit inside of the area of cover. When the cover area is broken down into subspaces many of them will not be large enough for the agent to fit entirely inside of it. There is no clear and efficient solution to determining if the agent is in cover when it spans several subspaces.

At this time, we feel that more research is necessary before we can determine if the SVBSP tree method can be efficiently used in a real-time simulation for finding cover.

2. Depth Mapping

The Depth Mapping algorithm has potential as a viable solution for finding cover. It still has problems with determining how to deal with objects that are behind the surface of the cover area, but its similarity to computer vision techniques make it interesting for cognitive studies.

Depth Mapping may provide the best support for cognitive studies of finding cover. It may be possible to merge what the agent can see and what it thinks the threat can see into a common picture. To this we can add what the agent knows about areas that it cannot see (memory) and speculation of the areas that it knows nothing about (beliefs). This may provide a closer model of the actual cognitive process of finding cover.

It is not clear at this point whether the algorithm is computationally efficient or not. There is a trade-off between resolution (which determines how accurately it finds cover locations) and computational complexity (which determines how fast it operates). As the grid becomes finer, it does a better job of finding cover, but the calculations required grow exponentially. Further research will be necessary to determine if there is a balance point where the algorithm is both fast and accurate.

3. Sensor Grid

The Sensor Grid algorithm provides the most effective solution of the three for finding cover in our current context. That is, finding cover in a first-person shooter, on a single machine, using current computing technology. It is computationally efficient enough to operate in real time and accurate enough to find reasonable cover locations. It is easy to program and can be applied to a wide variety of simulations. It is able to deal with a wide variety of situations inside the virtual environment and still provide a solution. For now, it is the best solution for finding cover in dynamic, three-dimensional, virtual environments.

4. CoverBot

a. Machine Performance

CoverBot has good speed performance as currently written, but several modifications could easily improve its performance even more. We tested the CoverBot on a Dell Dimension 8200 with a Intel Pentium 4 2.53 GHz processor, 512 MB of RAM, and an NVIDIA GeForce4 Ti 4600 video card. When the CoverBot attempts to find cover there is a barely noticeable flicker during execution. By changing some of the code from UnrealScript to C++ and improving our path-finding algorithm even this should disappear. Another

performance-improving option is to split the computation over several animation frames.

Changing all the functions in the CoverBot from UnrealScript to C++ will greatly increase the speed of the algorithm. UnrealScript runs many times slower than C++ code inside the simulation. Due to the similarities in UnrealScript and C++ this should be very simple to do.

An improved path-planning algorithm would also greatly enhance the speed performance of CoverBot. When designing CoverBot we wanted to leave the number of sensor rings and the number of sensors per ring as variables. This allowed us to adjust them until we got a good mix between speed and accuracy. However, this required us to use a brute force method for path-finding.

In order to determine if there are any two-part paths that lead to the cover position, we search every single other sensor in the grid. Sometimes this does not make sense. For instance, when checking a point directly to your right, there really is no need to check the point directly to your left that will require you to go back through your current position to reach the point to your right.

If we fix the number of sensor rings and number of sensors per ring, we can design a more efficient means of searching for paths into the cover. The paths that we want to search can be predetermined based on the arrangement of the sensors in the grid. This would also allow the introduction of paths with three or more sections where appropriate

b. Task Performance

CoverBot reliably finds cover under a wide variety of situations. We tested it on flat terrain and hilly terrain, inside of buildings, near clusters of boulders, near stacks of boxes, and around parked vehicles that can be seen under. In all cases, it was correctly able to identify cover locations. It is able to deal with small objects and large, few objects and many.

One area where CoverBot does have problems however is navigating inside of buildings. Buildings typically have narrow doors. Since

CoverBot uses its sensor grid to navigate around its environment, it must have one sensor on either side of the doorway with a clear path between them for it to successfully make it through the doorway. In all of the tests that we performed, CoverBot never successfully located a path through the doorway.

An easy way to fix this would be to enable CoverBot to use a level's waypoint graph as well as it's sensor grid to navigate through the environment. When determining if a point is reachable, CoverBot could search its sensor grid first for a path. If that fails it could query the waypoint graph to see if another path exists. We did not implement this in our demonstration, because the level that we used did not have a waypoint graph built for it.

C. FUTURE WORK

There are many opportunities for future work in the area of cover algorithms. Here are our suggestions:

- Create a full implementation of the SVBSP Tree algorithm.
- Create a full implementation of the Depth Mapping algorithm.
- Re-implement the Sensor Grid algorithm and eliminate one or more of the assumptions.
- Conduct a study to determine if there is a difference between the way we choose cover locations when we conduct deliberately planning and when we must immediately chose one while under fire.
- Conduct a study to determine how the performance of CoverBot compares to a human player in the same simulation.

LIST OF REFERENCES

1. "America's Army Fact Sheet." [http://www.thearmygame.com/]. March 2003.

2. "Binary Space Partition Trees in 3d Worlds." OpenGL [http://www.cs.wpi.edu/~matt/courses/cs563/talks/bsp/document.html]. 1997. [cited 26 September 2003]

3. "Binary Space Partitioning Trees FAQ." [http://www.opengl.org/developers/code/bspfaq/]. June 1998 [cited 26 September 2003].

4. Board, B. and Ducker, M., "Area Navigation: Expanding the Path-Finding Paradigm." *Game Programming Gems 3.* pp. 240-255. Charles River Media, 2002.

5. Chin, N. and Feiner, S., "Fast Object-Precision Shadow Generation For Area Light Sources Using BSP Trees." *Proceedings of the 1992 Symposium on Interactive 3D Graphics.* March 1992.

6. Chin, N. and Feiner, S., "Near Real-Time Shadow Generation Using BSP Trees." *Computer Graphics*. v.23(3). pp. 99-106, July 1989.

7. Chrysanthou, Y. and Slater, M., "Shadow Volume BSP Trees for Computation of Shadows in Dynamic Scenes." paper presented at the 1995 Symposium on Interactive 3D Graphics. Monterey, CA, 1995.

8. Department of the Army. "FM 21-75 - Combat Skills of the Soldier." 3 August 1984.

9. Everitt, C. and Kilgard, M., "Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering." [developer.nvidia.com]. 12 March 2002.

10. Fuchs, H., Kedem, Z., and Naylor, B., "On Visible Surface Generation by A Priori Tree Structures." *Computer Graphics*, v.14(3). pp. 124-133. 1980.

11. Fuchs, H., Kedem, Z., and Naylor, B., "Predetermining Visibility Priority in 3D Scenes." *ACM Computer Graphics Proceedings*. v.13(2). pp. 175-181. August 1979.

12. Higgins, D., "Terrain Analysis in an RTS – The Hidden Giant." *Game Programming Gems 3*. pp. 268-284. Charles River Media, 2002.

13. Horn, G. and Baxter, J., “An Interactive Planner for Tank Squadron Assaults.” paper presented at the 2000 UK Planning and Scheduling Special Interest Group workshop. Open University. Milton Keynes. United Kingdom. 15 December 2000.

14. Kelly, J., Loomis, J., and Beall, A., “Judgments of Exocentric Direction in Large-Scale Space.” *Journal of Vision*, 2(7), 20 November 2002.

15. Liden, L., “Strategic and Tactical Reasoning with Waypoints.” *AI Game Programming Wisdom*. pp 211-220. Charles River Media, 2002.

16. Matzka, G., “Shadow Algorithms” [http://www.cg.tuwien.ac.at/courses/Seminar/SS2001/shadow/shadow.pdf] . 20 June 2001 [cited 26 September 2003].

17. McCool, M., “Shadow Volume Reconstruction from Depth Maps.” *ACM Transactions of Graphics*. v.19(1). pp. 1-26. January 2000.

18. Russell, S. and Norvig, P., *Artificial Intelligence: A Modern Approach*. pp. 31-50. Prentice Hall, 1995.

19. Schneider, P. and Eberly, D., *Geometric Tools for Computer Graphics*. pp. 673-694. Morgan Kaufmann, 2003

20. Stout, B., “The Basics of A* for Path Planning.” *Game Programming Gems*. pp. 254-262. Charles River Media, 2000.

21. Sweeney, T., “UnrealScript Language Reference.” [http://udn.epicgames.com/pub/Technical/UnrealScriptReference/]. 8 August 2003.

22. “Unreal Developer Network.” [http://udn.epicgames.com/pub/Main/WebHome/]. 29 September 2003.

23. Van der Sterren, W., “Tactical Path-Finding with A*.” *Game Programming Gems 3*. pp. 294-306. Charles River Media, 2002.

24. Van der Sterren, W., “Terrain Reasoning for 3D Action Games.” *Game Programming Gems 2*. pp. 307-316. Charles River Media, 2001.

25. Young, T., “Choosing a Relationship Between Path-Finding and Collision.” *Game Programming Gems 3*. pp. 321-332. Charles River Media, 2002.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Chris Darken
   Naval Postgraduate School
   Monterey, California

4. Joseph Sullivan
   Naval Postgraduate School
   Monterey, California

5. Rudy Darken
   Naval Postgraduate School
   Monterey, California

6. Director, Army Modeling and Simulation Office
   HQDA, DCS G3, (DAMA-ZS)
   Washington, DC

7. Commander
   National Simulation Center
   Fort Leavenworth, Kansas

8. Commander
   National Training Center
   Fort Irwin, California

9. Alex Mayberry
   Executive Producer, Army Game Project
   Monterey, California

10. Christian Buhl
    Lead Programmer, Army Game Project
    Monterey, California

11. Greg Paull
Programmer, Army Game Project
Monterey, California

12. David J. Morgan
Monterey, California